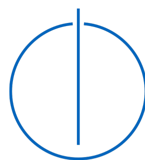
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Threat Analysis, Evaluation, and Mitigation for Smart Contracts Endorsed by TLS/SSL Certificates

Jan Felix Hoops



DEPARTMENT OF INFORMATICS

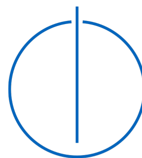
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Threat Analysis, Evaluation, and Mitigation for Smart Contracts Endorsed by TLS/SSL Certificates

Bedrohungsanalyse, -bewertung und -abwehr für durch TLS/SSL Zertifikate abgesicherte Smart Contracts

Author:	Jan Felix Hoops
Supervisor:	Prof. Dr. rer. nat. Florian Matthes
Advisor:	Ulrich Gallersdörfer, M.Sc.
Submission Date:	13.04.2021



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 13.04.2021 Jan Felix Hoops

Acknowledgments

I want to thank my advisor Ulrich Gallersdörfer for his continued support during my work on this thesis and his near-constant availability, making the task of writing a thesis during a pandemic feasible. To both Prof. Dr. Florian Matthes and Ulrich Gallersdörfer, I am thankful for the trust they placed in me and the opportunity to work on this exciting and important topic at the forefront of blockchain research. I could not have asked for a better one. Further, I enjoyed my conversations with Jonas Ebel, who was also writing a thesis on a related topic at the time and helped me gain some additional background information. I appreciate Vincent Bode, Max Schickert, and my father Jan Hoops for taking the time to help me proofreading. Finally, I would like to thank my parents for supporting me throughout my studies and being there for me when I needed them.

Abstract

Although the popularity of smart contract-based platforms, such as Ethereum, is ever-increasing, there is still no widespread measure in place to securely associate a smart contract with a real-world entity. This inability for users to verify their counterparty in a transaction leaves room for human error and malicious actors to cause significant damage. For example, an attacker could compromise a website and replace the account address listed there with one of their own. Users subsequently would send their funds to the attacker rather than the intended recipient, losing their funds irrevocably. Gallersdörfer et al. propose TLS-endorsed smart contracts (TeSC) to address this problem [25]. TeSC creates verifiable bindings between domain names and smart contracts. These bindings can only be created by the owner of a domain and its corresponding TLS certificate, allowing users to be aware of who they interact with.

In the first part of this work, we explore TeSC's impact on smart contract attacks. In this process, we identify and evaluate possible attacks against TeSC, concluding that it provides a valuable security benefit. However, one of the more dangerous possible attacks is typosquatting, a malicious practice relying on visual similarity of domain names. We address this threat by introducing a new component to TeSC, which can heuristically detect if a user is interacting with a typosquatting domain during binding verification. This allows front-end software, such as wallet software, to warn the user of the potentially unintended transaction recipient before issuing the transaction to the network. Thus, the component can reduce the chance of loss of funds.

We design, implement and evaluate this component in the second part of this work. Due to the large variety of typo-generation models used by attackers to generate typosquatting domains, it is difficult to detect them without large amounts of metrics or context information. As this component has to run client-side, it is also limited in the amount of data that can be stored and processed locally. We address these challenges by using a comparative detection approach based on string similarity. The component uses a two-stage process to maximize efficiency. First, the input domain is paired with domains chosen from an internally maintained reference list, and each of these pairs is evaluated by a neural network to detect possible cases of typosquatting. Second, possible cases are further evaluated using additional information to come to a final verdict. Only at this point does the component access any external resources, such as the domain name system. Evaluation results are promising with $f1 = 0.995$ and suggest effective typosquatting detection capability with an exceedingly low rate of false positives at 0.0000375. We do acknowledge that our test data is likely to produce optimistic results but are confident that real-world performance will not be significantly worse.

Contents

Acknowledgments **iii**

Abstract **v**

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Research Questions . . . 3
1.3 Structure . . . 3

2 Background **5**
2.1 Cryptography . . . 5
2.1.1 Hash Functions . . . 5
2.1.2 Merkle Trees . . . 6
2.1.3 Fundamentals of Asymmetric Cryptography . . . 7
2.1.4 Digital Signatures . . . 9
2.2 Internet Public Key Infrastructure . . . 10
2.2.1 Stakeholders . . . 10
2.2.2 X.509 Certificates and their Lifecycle . . . 11
2.2.3 TLS/SSL . . . 14
2.3 Ethereum . . . 15
2.3.1 The Ethereum Blockchain . . . 15
2.3.2 Ether and Ethereum Accounts . . . 17
2.3.3 Smart Contracts . . . 18
2.3.4 Ethereum Consensus Algorithm . . . 20
2.4 TLS-endorsed Smart Contracts . . . 21

3 Related Work **23**
3.1 Established TLS-related Systems . . . 23
3.1.1 IPsec . . . 23
3.1.2 DNSSEC . . . 23
3.1.3 Certificate Transparency . . . 24
3.1.4 CRL . . . 26
3.1.5 OCSP . . . 26
3.1.6 CRLite . . . 26
3.2 Blockchain Account Authentication . . . 27
3.3 Smart Contract Vulnerability Analysis . . . 28
3.4 Typosquatting Detection . . . 29

4 Analysis **31**
4.1 Use-Cases . . . 31
4.1.1 ICO Investments . . . 31
4.1.2 Digital Document Verification . . . 32
4.2 Stakeholders . . . 32
4.3 Requirements Engineering . . . 34
4.3.1 Established TLS System Goals . . . 34
4.3.2 Common Requirements of TLS Systems . . . 35
4.3.3 Requirements for TeSC . . . 37
4.4 TeSC Vulnerability Analysis . . . 40
4.4.1 Attacking Ethereum Smart Contracts . . . 41
4.4.2 Attacking TeSC . . . 45
4.5 Limitations of TeSC . . . 48
4.6 TeSC Registry Robustness . . . 49

5 Component Design **51**
5.1 High-Level Design . . . 51
5.2 Typosquatting Candidate Detection . . . 53
5.2.1 Simplifications . . . 53
5.2.2 Data Preparation . . . 54
5.2.3 Feature Selection . . . 56
5.3 Typosquatting Candidate Evaluation . . . 59

6 Component Implementation **61**
6.1 Technology Choices and Tools . . . 61
6.2 Component Integration . . . 61
6.3 Component Implementation . . . 62
6.3.1 Classifier Implementation . . . 63
6.3.2 General Implementation . . . 63

7 Component Evaluation **67**
7.1 Quantitative Evaluation . . . 67
7.2 Qualitative Evaluation . . . 69

8 Conclusion **73**

List of Figures **75**

List of Tables **77**

Bibliography **79**

1 Introduction

With the rising popularity of blockchain and smart contract-based systems, the incentives for attacking these systems are plenty. Smart contracts are versatile and fill many different roles like conducting auctions, vending tokens, or exchanging currency. And while blockchains themselves are tamper-proof, a weakness in the blockchain ecosystem is the difficulty involved in verifying the identity of an operator behind a smart contract. Consequently, the process of exchanging address information is an attractive target for attackers because (fraudulent) transactions can not be reverted. Thus, an attacker can be sure to keep all of the funds that they manage to divert to one of their wallet addresses. They could achieve this by hacking the website of an organization accepting cryptocurrency and changing the displayed wallet address to an address under the attacker's control. A prime example of this attack vector is Coindash's disastrous Initial Coin Offering (ICO) in 2017, causing $7 million in damages [65]. Another possible target for attackers is the social media accounts of big organizations and well-known individuals. This recently transpired when an attacker gained access to Twitter's employee tools and tweeted from several high-profile accounts to trick users into sending Bitcoin to a wallet address under their control [6]. The hacked Twitter accounts belonged to big companies like Apple, some well-known individuals like Elon Musk, and prominent cryptocurrency exchanges like Coinbase and Binance [56]. Unfortunately, users had no way to check if the advertised wallet addresses actually belonged to the tweeting companies and individuals. Even though Coinbase took notice within a minute of the first malicious tweets sent from other cryptocurrency exchange's Twitter accounts and blacklisted the attacker's wallet address to at least protect their own users (and other exchanges followed soon after) [5], the damages still amounted to at least $109,000. All of these attacks can be classified as front-end attacks. In total, Chen et al. estimate that these attacks have caused $24.5 million, putting this group of attacks second in terms of financial damage only superseded by application-level attacks [10]. Additional damage to the public perception of blockchain technology is immeasurable.

To address this problem, Gallersdörfer et al. propose TLS-endorsed smart contracts (TeSC) [25]. TeSC's main benefit compared to previously proposed solutions is that it leverages the TLS/SSL public key infrastructure to avoid the bootstrapping problem. It is a proposal focused on the Ethereum blockchain but could theoretically be adapted to any other blockchain featuring smart contracts. Any entity in possession of a domain and a matching X.509 certificate can create a smart contract and bind it to their fully qualified domain name (FQDN) by adding an endorsement to it. This endorsement consists of a claim and a signature. A claim contains the address of the endorsed smart contract, the FQDN it should be bound to, and the expiry date. The signature is

created by signing the endorsement with the certificate's private key. A user can rely on an off-chain verifier application that fetches the smart contract endorsement from the blockchain and all relevant certificate data from the web to determine its authenticity.

TeSC's specification is still evolving, and this thesis is intended to provide valuable research that may help guide future design decisions. To that end, we examine how TeSC impacts existing attacks against smart contracts and how TeSC itself could be attacked. As part of our threat analysis, we identify typosquatting as TeSC's final weakness. Typosquatting is a type of attack relying on similar-looking domain names like `tum.de` and `tunn.de`. An attacker could abuse this similarity to create a TeSC endorsement with the latter domain or a similar variation to trick inattentive users into sending a transaction to them instead of the legitimate smart contract associated with the former domain. Therefore we introduce a new component designed to mitigate typosquatting in the second part of this thesis.

1.1 Problem Statement

TeSC aims to solve the problem of smart contract authentication by bridging two large and constantly evolving ecosystems: blockchain and the TLS/SSL public key infrastructure of the web. This makes it a highly complex system, especially from a security standpoint. TeSC is still in development, and its effectiveness and general feasibility remain to be evaluated. For this evaluation and potential changes, there are several factors that have to be considered:

- **TeSC's trade-off** TeSC's arguably biggest strength, the avoidance of a bootstrapping phase, comes at a price. The system is potentially inheriting weaknesses from the TLS/SSL public key infrastructure, and these need to be identified and adequately addressed.
- **Unintended use-case for TLS/SSL certificates** TeSC's reliance on the TLS/SSL public key infrastructure requires cautious examination because using X.509 certificates this way was never explicitly intended. Thus the standard is not built with TeSC's use case in mind. The infrastructure was designed to handle web traffic encryption. In contrast to that, TeSC uses these certificates to authenticate smart contracts, making these certificates critical in securing irreversible financial transactions of unlimited value. That could impact the assessment of certain threats, which are acceptable in the context of the internet, but not for TeSC's.
- **Usability** As for any system, usability is vital for user adoption. However, security and usability are commonly conflicting goals. TeSC has to be as easy to use as possible, and changes or additional measures need to strive towards that.
- **Cost** Computational power on the Ethereum world computer is not free. Any security mechanisms added or changed must not compromise TeSC's economic viability.

- **Living Specification** As previously mentioned, TeSC's specification is by no means final and constantly evolving in response to new insights. This introduces additional challenges in keeping up with the specification.

This final factor was addressed by freezing the TeSC specification used as the basis for this work at the latest possible point. That way, the relevance of this thesis is preserved while facilitating the work process.

1.2 Research Questions

This thesis is driven by three overarching research questions along with some related sub-questions:

RQ1 What are actively used security mechanisms for the TLS/SSL certificate infrastructure on the web?
For accurate evaluation of TeSC and its appropriate modification and augmentation, it is necessary to characterize this system properly. We aim to pull from existing and well-established security mechanisms also related to the TLS/SSL infrastructure to set formal requirements for TeSC.

RQ2 What attacks could be performed against TeSC?
To evaluate and appropriately mitigate the threats TeSC is facing, we have to identify them. This is a multi-step process:

(a) What attacks on smart contract infrastructure exist?

(b) How does TeSC impact existing attacks?

(c) What new attacks does TeSC introduce?

RQ3 How can TeSC be augmented to improve its security benefit?
With knowledge of TeSC's weaknesses, appropriate action can be taken. The specifics largely depend on the results of **RQ2**, but some potentially relevant questions are:

(a) Can we adapt security mechanisms used in the TLS/SSL certificate infrastructure on the web?

(b) How effective are any newly added security mechanisms?

(c) How costly are any newly added security mechanisms?

1.3 Structure

The structure of this thesis is closely following the research questions. Chapter 2 presents all necessary background knowledge required to follow this thesis. This includes information on cryptography, certificate infrastructure, and blockchain technology. In

chapter 3, we provide some context by elaborating on different kinds of systems bearing similarities to TeSC. Here we address **RQ1** by looking at different security mechanisms working with TLS/SSL certificates. Also, we go over blockchain-based name systems that can be understood as TeSC's closest competitors. Additionally, state of the art on smart contract vulnerability analysis and typosquatting prevention is discussed. Next, in chapter 4, we characterize TeSC and evaluate it with a focus on security, addressing **RQ2**. As part of this, we look at use-cases, set and motivate formal requirements, and analyze possible attacks. During our analysis, we identify typosquatting as a weakness in TeSC's current design, answering **RQ3**. From here on, the remaining chapters cover the augmentation of TeSC with a typosquatting detection component. Design, implementation, and evaluation of this component are covered by chapters 5, 6, and 7. Finally, we reflect on our work and give pointers to possible future work in chapter 8.

2 Background

This thesis builds upon several existing systems and infrastructures that need to be introduced. The following sections go over all important associated concepts the reader needs to be familiar with to follow along using a bottom-up approach. It contributes to a security mechanism for Ethereum smart contracts that leverages the web's TLS/SSL public key infrastructure (PKI).

Cryptography is an important foundation for both Ethereum and the TLS/SSL PKI. The necessary fundamentals are presented in section 2.1. Next, we introduce the TLS/SSL PKI in section 2.2 and Ethereum in section 2.3. Finally, in section 2.4, we address the security mechanism this thesis contributes to, which is TLS-endorsed smart contracts (TeSC).

2.1 Cryptography

For this thesis, only four concepts from cryptography are of importance. The first one is hashing, which is relevant because it is one of the underlying technologies of blockchains and thus Ethereum. The second concept is the Merkle tree, which is also used by blockchains. The final two concepts are asymmetric cryptography and digital signatures, which are relevant for both blockchains and the TLS/SSL PKI.

2.1.1 Hash Functions

A hash function h is a function that maps any input x of finite length to an output $h(x)$ of fixed length [50]. Possible applications for this kind of function are found in a wide range of fields, from efficient data structures (e.g., hash maps) to many forms of data integrity verification. The latter application is a security-critical use of hash functions and only permits cryptographic hash functions. A cryptographic hash function must have three properties:

Preimage Resistance The function must be irreversible. That means that given an output y, it is computationally infeasible to determine an original input x, such that $h(x) = y$.

Second Preimage Resistance This property is sometimes also referred to as *Weak Collision Resistance*. It requires that, given an input x_1 and output $h(x_1)$, it is computationally infeasible to find x_2, such that $h(x_1) = h(x_2)$.

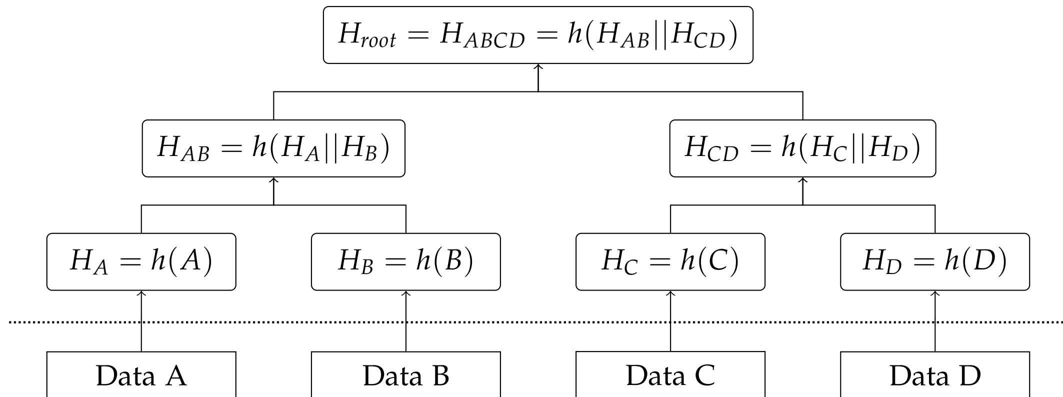


Figure 2.1: Structure of a Merkle tree.

Collision Resistance This property is also sometimes referred to as *Strong Collision Resistance*, and as that name implies, it is strongly connected to the previous property. It is stronger, because it allows the hypothetical attacker one more degree of freedom: It must be computationally infeasible to find x_1 and x_2, such that $h(x_1) = h(x_2)$.

These properties ensure that a hash produced with a cryptographic hash function can be seen as a fingerprint of a piece of data. Preimage Resistance ensures that this piece of data can stay secret if required by the application. Second Preimage Resistance and Collision Resistance prevent a piece of data from being altered without affecting the fingerprint.

There is one more property that is essential to only select security-critical uses of hash functions [24]:

Hiding The function must prevent a piece of data x from being discovered if that data is concatenated with a random value r. More formally, given $h(r||x)$, it is computationally infeasible to find x.

This last property is important for most contemporary blockchain consensus mechanisms and will be picked up again later in this chapter when introducing Ethereum.

2.1.2 Merkle Trees

Merkle trees are named after Ralph C. Merkle, who introduced the concept in 1988 [43]. A Merkle tree is a data structure used to verify the integrity of large data structures [24], such as large lists of data blocks.

The structure of a Merkle tree is illustrated in figure 2.1. In basic terms, it is a binary tree storing hashes [24]. The leaves of the Merkle tree are created by hashing a single data block each. For every subsequent level of the tree, the hashes in two nodes of the previous level are concatenated and hashed to form a new parent node. If the number

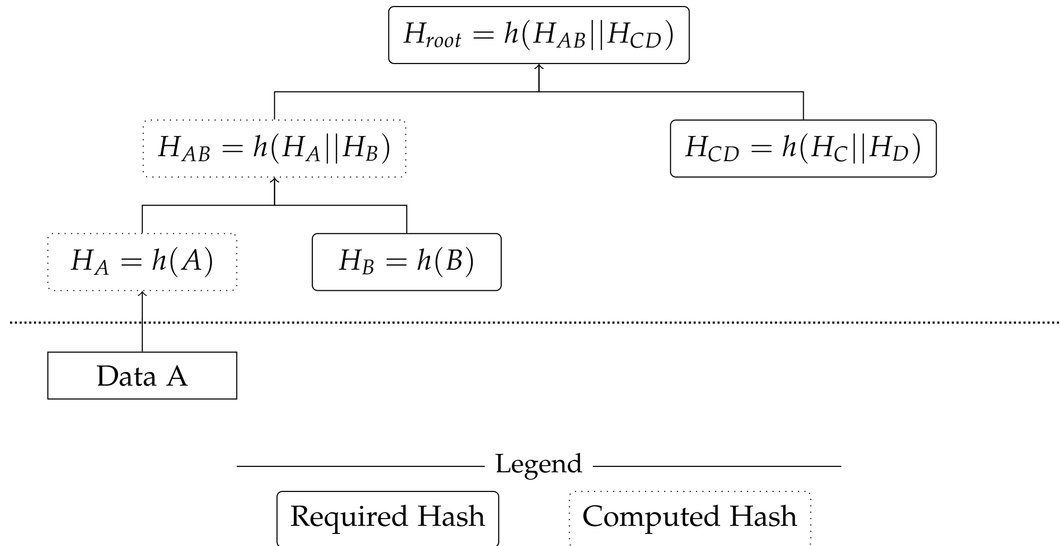


Figure 2.2: Proof of membership in a Merkle tree.

of data items and thus leaves is not a power of two, there will be cases of a node having no counterpart to form a parent node. In this case, the hash in the node is concatenated with itself to form a new parent node. This guarantees the existence of one single root node containing the *root hash*.

The root hash of the Merkle tree can be used to verify the integrity of the data structure the tree was built upon. While this could also be achieved by simply hashing the data structure in its entirety, Merkle trees have one important advantage: They efficiently provide *proof of membership*. Confirming that a certain data block is part of the Merkle tree only requires the data block in question, the root hash, and one additional hash for every level of the tree except the root. With that information, the tree root can be computed and compared to the retrieved root hash in $log(n)$ time. This process can be seen in figure 2.2 for data block A. Required data is shown with a complete outline and computed hashes with a dotted outline. The arrows indicate what data block or hash is necessary to compute the target. In addition to proof of membership, a sorted Merkle tree can also provide proof of non-membership in a similar manner.

2.1.3 Fundamentals of Asymmetric Cryptography

Asymmetric cryptography is also sometimes referred to as *public-key cryptography* [50]. In contrast to symmetric cryptography, there is no single common secret key shared by all communicating parties. Instead, every party owns a pair of keys: one being private and the other being public. As implied by the names, the private key is kept a secret, while the public key is shared with any other communicating party. The system is designed such that a message encrypted with one key can only be decrypted with the counterpart key.

In its most basic use-case, asymmetric cryptography can be used to send a secret message. This simple example assumes that public keys were shared in advance in some secure way [50]:

1. Alice encrypts a message x with Bob's public key k_{pub} resulting in the encrypted message $y = e_{k_{pub}}(x)$.
2. Alice sends the encrypted message y to Bob.
3. Using his private key k_{priv}, Bob can obtain the original message $x = d_{k_{priv}}(y)$ by decrypting the encrypted message y.

This simple example assumes that public keys were shared previously. Securing the process of sharing keys is a challenge. If an attacker manages to replace a public key with one under their control, they can read all intercepted messages encrypted using that public key. This renders a naive public key exchange at the start of a connection vulnerable to man-in-the-middle attacks. This challenge of securely distributing keys is addressed using a public key infrastructure, which is the topic of section 2.2.

In contrast to using symmetric cryptography for the exchange of secret messages, asymmetric cryptography has two significant advantages:

One Specific Recipient Only one party, namely the one in control of the private key belonging to the public key used when encrypting a message, can decrypt a message.

Key Compromise Mitigation Private keys never have to be shared, eliminating the risk of them being intercepted. Additionally, a party's private key being compromised only affects messages sent to that party. Messages sent by that party are still secure.

However, these advantages are not the main appeal of asymmetric cryptography. Beyond the simple exchange of secret messages, this field of cryptography enables some completely new use-cases [50]:

Key Establishment Using asymmetric cryptography, it is possible to agree upon a secret key to use for symmetric encryption in subsequent message exchange, without the need for a secure channel at any part of the process. This is desirable because symmetric encryption algorithms are generally less computationally expensive than their asymmetric counterparts.

Identification & Authentication A private key holder can prove their ownership of the key and thus their claimed identity.

Non-Repudiation It is possible to generate indisputable proof that a message was authored by a key holder and has not been altered.

The unification of the last two use-cases is represented by digital signatures, which are discussed next.

2.1.4 Digital Signatures

Digital signatures are an important tool leveraging hashes and asymmetric cryptography to provide a digital counterpart to handwritten signatures [50]. As mentioned when discussing asymmetric cryptography, a message encrypted with one key of the pair can be decrypted with the other. Instead of encrypting using the public key, signatures are produced using the private key. A simple example of this process is as follows [50]:

1. Alice encrypts a message x with her private key k_{priv} resulting in the encrypted message $y = e_{k_{priv}}(x)$.
2. Alice sends the encrypted message y, as well as the original message x to Bob.
3. Using Alice's public key k_{pub}, Bob can compute the original message $x' = d_{k_{pub}}(y)$. If $x' = x$, then the signature is valid.

Similar to the simple example for asymmetric cryptography in the previous section, it is assumed that there is some system in place to share public keys securely. Without the possibility to connect a public key to an entity, a digital signature is of no use. With this assumption, Bob can be sure of three things after the presented exchange:

Message Origin Authentication Because everyone has to keep their private key secret, and the signature can not be produced without that key, Bob can be sure that the message was written by Alice. Additionally, he can prove that fact to a third party if necessary.

Message Integrity Thanks to the hash that has to be produced by Alice, Bob can rest assured that the message was not altered in the transfer.

Non-Repudiation Bob can be sure that Alice can not deny signing this message. Once the signature is produced and shared, it can not be taken back.

In practice, digital signatures work differently than the presented simplified example. Messages can be very long, making the encryption process required to produce a signature computationally expensive and the signature itself very long. Instead, hashing can be used to improve the efficiency of digital signature schemes without losing any strength in their provided assurances. The example from before can be improved as follows:

1. Alice uses a hash function h, to hash her message x and proceeds to encrypt that hash using her private key k_{priv} producing a signature $s = e_{k_{priv}}(h(x))$.
2. Alice sends the signature s, as well as the original message x, to Bob.
3. Using Alice's public key k_{pub}, Bob can compute the originally encrypted hash $v = d_{k_{pub}}(s)$. Bob is aware of the hash function used by Alice and can compute the hash of the message $v' = h(x)$. If $v = v'$, then the signature is valid.

Thanks to digital signatures, a pair of asymmetric keys, or more specifically, a public key, can be considered an identity.

2.2 Internet Public Key Infrastructure

Public key infrastructures (PKI) are designed to solve the problem of publishing public keys [7]. As discussed previously, asymmetric key cryptography relies on the secure exchange of public keys. Digital signature schemes can only confirm that the message was authored by the owner of a specific public key, but this knowledge alone is insufficient for most applications. On the internet, for example, a client accessing an online store has to be able to confirm that it is directly communicating with the store's servers, to prevent man-in-the-middle attacks. For this purpose, a public key has to be connected to some human-readable identity. The internet PKI primarily binds a public key to a subject's name, which is a domain name in most cases. All parties involved in the establishment of these bindings are discussed in section 2.2.1. The bindings themselves are standardized to take the form of an X.509 certificate, presented in section 2.2.2. Finally, in section 2.2.3, we address Transport Layer Security (TLS), which is the protocol leveraging the internet PKI to establish secure connections for all kinds of applications.

2.2.1 Stakeholders

There are mainly three types of entities directly involved with the internet PKI [7]:

Certificate Authority The smallest and most closed-off type of stakeholder is the certificate authority (CA). CAs are the most critical part of any PKI. They are usually commercial enterprises responsible for issuing certificates. By issuing a certificate, the CA assures anyone that a specific public key belongs to a specific real-world entity. Thus, the trustworthiness of a PKI relies on the trustworthiness of the involved CAs.
In theory, anyone can operate a CA, but to be truly operational, a CA has to be widely trusted by end-users and end-user software developers. This trust is dependent on independent inspections and a spotless record.

End Entity Any end-user of the PKI is classified as an end-entity (EE). EEs generally fit into at least one of two categories: they are a certificate subject, meaning they are a customer of the CA, and/or they operate client software.
Certificate subjects can request the issuance of new certificates. Additionally, they can revoke a certificate in case their private key has been compromised.
Client users are the biggest group within EEs. For example, everyone using a web browser falls into this category. They rely on the internet PKI's certificates to establish secure connections.

Registration Authority A registration authority (RA) is an intermediary to a CA. They handle interactions with certificate owners and provide various administrative

Attribute Name	Attribute String	Example Value
common name	CN	wwwv11.tum.de
organizational unit	OU	Leibniz-Rechenzentrum
organization	O	Technische Universitaet Muenchen
locality	L	Muenchen
state or province name	S	Bayern
country	C	DE

Table 2.1: An example of an X.501 distinguished name.

services. Usually, they are responsible for authenticating EEs performing any type of request, as well as validating the request itself to some degree. Accepted requests are relayed by the RA to the CA.

2.2.2 X.509 Certificates and their Lifecycle

The internet PKI uses certificates adhering to the X.509 version 3 standard [13]. This standard defines a clear and extendable structure for a signed binding between a public key and a distinguished name (DN). DNs are standardized as X.501 and have a hierarchical structure. By themselves, they are a list of relative attribute-value pairs. With every further pair, the described entity is further narrowed down. The X.509 defines a set of standard attributes that have to be supported by any implementation. Most certificates encountered on the web use a subset of these attributes, which were originally defined as standard identifiers for the Lightweight Directory Access Protocol (LDAP) specification [67]. To further illustrate the concept, table 2.1 shows the attributes used by the certificate on `www.tum.de` in ascending hierarchical order.

An X.509 certificate consists of three required fields [13]: `tbsCertificate`, `signatureAlgorithm`, and `signatureValue`. The certificate itself is found as the value of the first field in the form of a TBSCertificate, or To-be-signed-Certificate. As the name implies, this is the raw, unsigned certificate. The latter two fields hold information about the cryptographic algorithm used to produce the signature, as well as the signature.

Table 2.2 lists the TBSCertificate fields. Among these is the DN of the issuing CA, the DN of the subject, and of course, the subject's public key. Another particularly important field is the validity window. It contains two timestamps marking the start and the end of the certificate's validity period. Giving certificates a limited validity period ensures that the length of time the CA has to maintain status information is well defined. Also, the regular renewal of certificates provides the benefit of constantly replacing the public keys in circulation with longer, stronger ones when necessary.

In its most basic form, an X.509 certificate simply is a message signed by a trusted third party. Thus, a certificate requires no secrecy and can be transmitted via untrusted channels.

Tag	Explanation
version	Indicates the version of X.509 that this certificate adheres to. Currently, there are v1, v2, and v3.
serialNumber	A positive numeric identifier for a certificate that has to be unique per CA.
signature	A data construct that contains information on the algorithm used by the CA to sign the final certificate. It has to be identical to the algorithm named in the `signatureAlgorithm` field at the root level of the certificate.
issuer	The distinguished name of the signing CA.
validity	Two timestamps defining the start and end of the validity period for this certificate.
subject	The name of the subject, which this certificate binds a public key to.
subjectPublicKeyInfo	Information on the type of public key, as well as the public key that is bound to a DN.
issuerUniqueID	An optional ID uniquely identifying the issuing CA. This is intended to distinguish two entities in case a DN is ever reused. The field is only supported since v2.
subjectUniqueID	An optional ID uniquely identifying the certificate's subject. It is the counterpart to issuerUniqueID.
extensions	Introduced in version 3. This field contains a list of extensions. These allow the inclusion of additional information into the certificate, including the permitted types of usage for the subject's public key.

Table 2.2: A list of the top-level fields contained in an X.509 TBSCertificate.

Certificate Revocation

In some instances, a certificate needs to be invalidated before the end of the validity period. One example of such a case is a subject's private key being stolen. Because certificates are circulating freely, the issuing CA (or anyone else for that matter) can not remove the certificate from circulation. Instead, a CA has to mark the certificate identified by the certificate serial number as revoked using an additional mechanism. Over time, multiple of these mechanisms have been developed and see active use to this day. These mechanisms are similar to this thesis in the way that they try to mitigate risk caused by misissued certificates and thus can be considered related work. They are examined as part of section 3.1.

Certificate Validation

The process of deciding the validity of a certificate is known as certificate validation. In its most basic form, this process consists of three aspects to check: whether the validity period has not expired, whether the certificate chain of trust is valid, whether the certificate has not been revoked, and whether the certificate is permitting the intended key usage [7]. To understand aspect two, some further explanation is necessary.

As presented so far, the internet PKI relies on trust in CAs to bind public keys to DNs. However, to validate a certificate signature, the validator needs to be able to know that the CAs public key is trustworthy. Most software relying on the internet PKI, such as browsers, overcome this hurdle by shipping with a default set of trusted CAs, provided by the developer. The members of this set are referred to as trust anchors [13] or root CAs. Usually, they are stored in the form of self-signed certificates, which are certificates where the subject and issuer fields are identical, and the certificate has been signed using the subject's private key.

As mentioned in the introduction to this section, the internet PKI is hierarchical. That means that CAs can create new subordinate CAs by signing a certificate including the appropriate extension allowing the certificate subject's private key to be used to sign further certificates. This hierarchical structure distributes the responsibility of certification between many CAs. This situation implies that not every certificate is directly signed by a root certificate. Instead, there has to be a path from the final certificate back to a root CA [13]. This path is also referred to as a chain of trust.

Figure 2.3 presents an illustrated example for certificate path validation. Each arrow indicates one certificate's subject is the issuer of the certificate pointed at, and the example assumes that every certificate's validity depends solely on its chain of trust. Only certificates $Cert_1$ and $Cert_2$ are considered root certificates and thus inherently trusted. This leads to all certificates being trusted that were either directly signed by any root certificate or have a valid path to one of the root certificates to be trusted. These are shown with a complete outline. All other certificates are untrusted and shown with a dotted outline.

Checking whether a certificate is trustworthy, whether its validity period has not

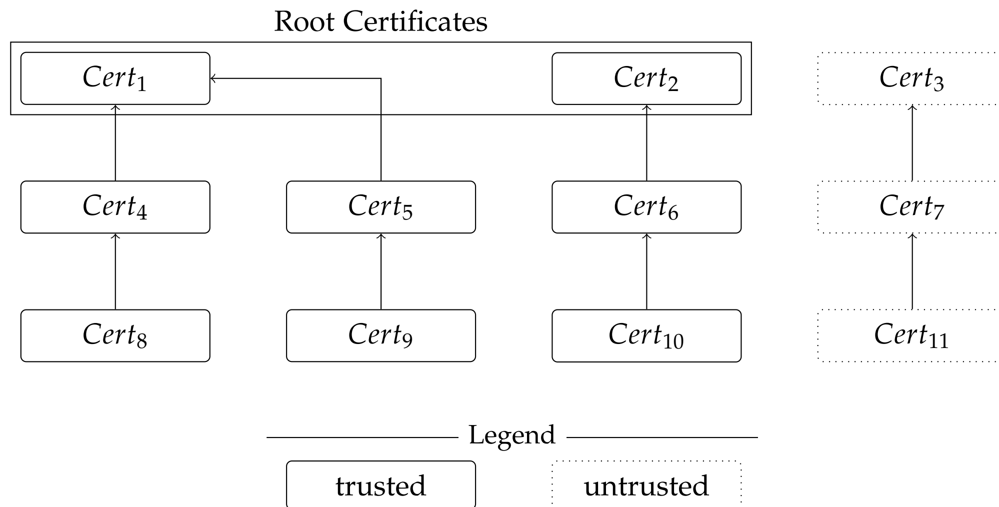


Figure 2.3: Different scenarios of certificate path validation.

expired, and whether it has not been revoked is required for every certificate of a path. This process is called path validation. Only if there exists such a path and only if all certificates on this path are processed successfully and were not revoked, the final certificate is considered trustworthy. Whether a certificate is successfully processed depends on a multitude of factors, such as the semantic integrity of the certificate and the time of validation being inside the certificate's validity period.

The final aspect is determining whether the intended key usage is permitted. Certificates can specify how a subject's public key may be used. The validator must ensure that the intended usage of the public key being validated is permitted by the certificate. Possible uses include "encipherment, signature, [and] certificate signing"[13].

2.2.3 TLS/SSL

TLS (Transport Layer Security) and SSL (Secure Sockets Layer) are protocols that enable secure communication between a client and a server through the use of the Internet PKI. SSL was developed in 1994 by Netscape. In the following years, the SSL protocol development moved to the Internet Engineering Task Force (IETF), and since 1999 they release new versions of the protocol under the name TLS. Today, TLS is widely adopted, and many application-layer protocols, such as HTTPS (Hypertext Transfer Protocol Secure), address their security requirements by building upon TLS [64]. Because TLS and SSL are closely related, for the remainder of this thesis, we will refrain from constantly referring to both TLS and SSL where necessary and instead mention only TLS.

The TLS protocol can be divided into two parts: the handshaking protocol and the record protocol. As the name suggests, the first part performs the handshake necessary to establish a secure communications channel. During this handshake, the

server authenticates itself by sending a `certificate message` to the client. This message contains the server's certificate, as well as all intermediate certificates that might be required to complete a path to a trust anchor and allow the client to successfully validate the server's certificate as described in the previous section. After a successful handshake, the record protocol is responsible for data exchange between client and server while providing privacy, data integrity assurance, and data origin authentication.

2.3 Ethereum

Ethereum is a distributed ledger, and computation platform introduced through a white paper by Vitalik Buterin in 2013 [9] and considered a second-generation blockchain. Thus Ethereum could learn from and try to improve on Bitcoin. Blockchains are an implementation of a distributed ledger. Distributed ledgers are envisioned to provide distributed, highly redundant data storage with consensus on the data while not requiring trust in any authoritative party.

Bitcoin introduced the concept of a blockchain to the world in the now-famous white paper "Bitcoin: A Peer-to-Peer Electronic Cash System" published in 2008 under the pseudonym of Satoshi Nakamoto [49]. Bitcoin implements an electronic currency of the same name that functions without any oversight. It was meant to be independent of traditional financial institutions to be free from any interference and to lower the costs associated with financial transactions. For example, Bitcoin's design makes it impossible to freeze or seize funds.

Ethereum further expands on the ideas of Bitcoin. Like Bitcoin, Ethereum is based on its own blockchain and has its own built-in currency, which is presented in sections 2.3.1 and 2.3.2, respectively. Next, we examine what is probably the most important difference distinguishing Ethereum from Bitcoin: the ability to execute program code on the blockchain. These programs are called smart contracts and are presented in section 2.3.3. Finally, in section 2.3.4, we focus on how Ethereum maintains consensus within its peer-to-peer network and how this consensus algorithm is expected to radically change in the near future.

2.3.1 The Ethereum Blockchain

A blockchain in its most basic form is a ledger of state transitions starting from an initial state. These transitions are referred to as transactions, and multiple transactions are grouped into a block. The initial block is called the genesis block. Multiple blocks chained together form the blockchain. In the case of Bitcoin, the current state of the ledger has to be computed from the initial state up by applying all of the transactions in order. While that is efficient in terms of storage space per block, it introduces disadvantages in the long term. First, most network members, usually referred to as nodes, need to download and store the entire blockchain. Second, state queries, such as querying an account balance, become more expensive because they need to be computed

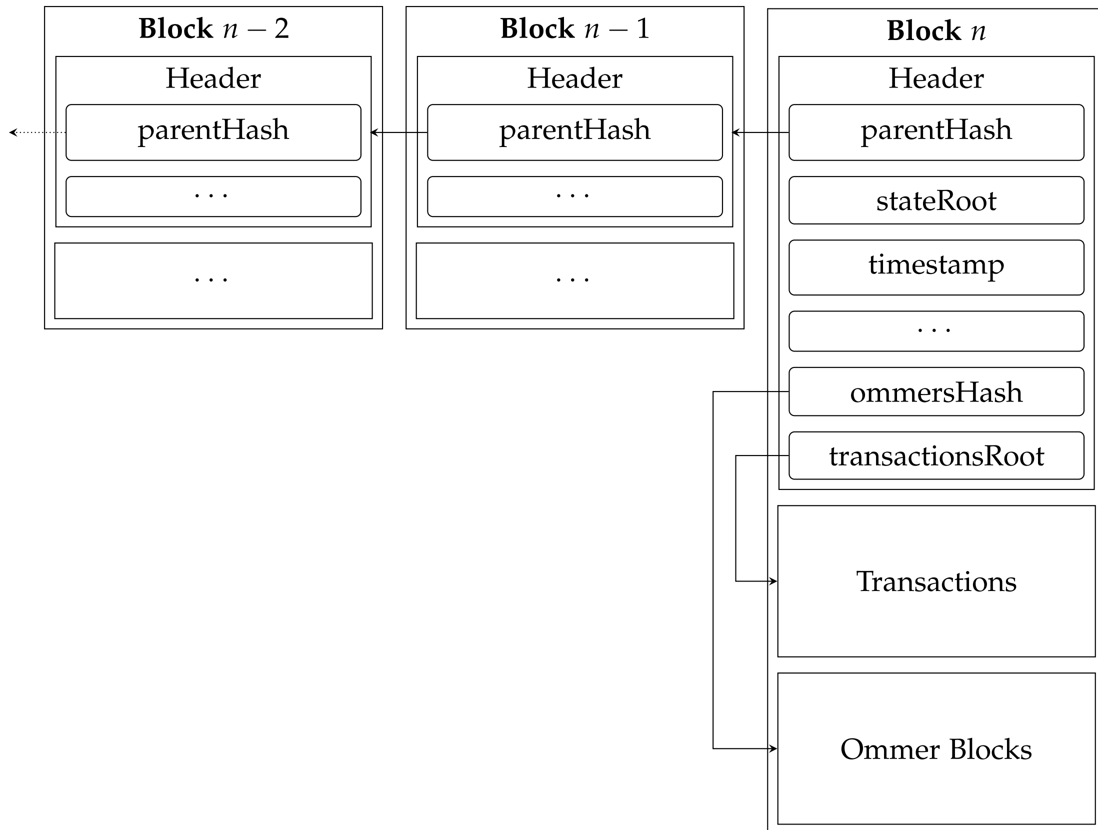


Figure 2.4: The basic structure of the Ethereum blockchain.

by iterating over all blocks. Ethereum tries to mitigate these problems by indirectly incorporating the current state of the ledger into each block.

The basic structure of a block and how it fits into the Ethereum blockchain can be seen in figure 2.4. Fields that are not required to understand the basic data structure itself have been omitted for the sake of clarity. Also, the figure uses arrows to emphasize the chain of hashes, which is explained in the remainder of this section.

Ethereum blocks consist of three major components, with the first two being part of the body and the final one being the header [66]:

Transactions Naturally, the block has to contain the transactions it is supposed to group. There is no direct limit on the number of transactions per block.

Ommer Blocks Ommer blocks are an innovation of Ethereum. It describes a block that is a sibling of one of the current block's parents within 6 generations. These blocks are relevant for the consensus algorithm. Per the specification, the current block contains only the headers of ommer blocks and is limited to a maximum of two.

Header The header captures the integrity of the block and links it to the preceding block. This link is established by including the hash of the preceding block's header in

the field parentHash. It is sufficient to solely consider the preceding block's header because the header contains hashes of the block body. The transactionsRoot field contains the root hash of a modified version of a Merkle tree built over the list of transactions. The ommersHash field simply contains the hash over the ommer block list. Additionally, the header contains some more information required for the consensus algorithm.

This chaining of hashes ensures that no old block can be manipulated. If someone attempted to alter a block, the alteration would affect the block header and thus the block hash, which would no longer match the parentHash in the succeeding block and break the chain. If someone were to alter the block's body without adjusting the header, the block's internal consistency would be destroyed, leading to the block being considered invalid and invalid blocks can not be part of any valid chain. Because all honest nodes accept the longest valid chain of blocks as the global state, the manipulated block will end up being ignored.

The header field stateRoot is another of Ethereum's innovations. As mentioned before, Ethereum differs from Bitcoin because it tracks the current global state in each block, and this is the field enabling it. It holds the root hash of a modified Merkle tree built over the entire global state. This global state consists of all account states, which will be explained in the next section. The modified Merkle tree used is cleverly designed: tree nodes can refer to a node from a previous version of the tree to designate it as a child. This allows storage space-efficient incremental changes to the global state because only nodes that are changed have to be added to storage. Standard network nodes, also referred to as full nodes, are expected to maintain a database with all tree nodes, which can be computed from the blockchain's blocks. This state tracking enables light nodes that can efficiently query the global state without storing the entire blockchain. Instead, they request the branch of the state they are interested in from a full node along with the current block header. By Merkle inclusion proof, they can efficiently validate the global state using the block header's stateRoot. This does still require the light node to have verified the block header's validity. Light nodes can choose how exactly they handle this problem, leading to various degrees of security. For example, they can choose only to validate the chain of header hashes and the consensus algorithm data contained in each header [37].

2.3.2 Ether and Ethereum Accounts

Ether is Ethereum's built-in currency and is generally shortened as ETH. One Ether can be broken up into smaller units, with Wei being the smallest and Ether the largest. These units, also referred to as subdenominations, are listed in table 2.3 [66].

Entities can own Ether. They are identified by their account, which is also sometimes called a wallet. With Ethereum being fully decentralized, identity management has to be decentralized as well. This is achieved through the use of asymmetric cryptography and, more specifically, the use of public keys as identities. Because anyone can create an

Name	Multipliers (rel. to Wei)	
	Base 10	Decimal
Wei	10^0	1
Szabo	10^{12}	1.000.000.000.000
Finney	10^{15}	1.000.000.000.000.000
Ether	10^{18}	1.000.000.000.000.000.000

Table 2.3: An overview of Ether subdenominations.

account by generating a pair of asymmetric keys, accounts can be created without any governing party.

Accounts are referred to by their address. The address for a given account is computed by applying the Keccak hash function to the public key and taking the least significant 160 bits of the hash [66]. The resulting address is usually displayed in hexadecimal form to improve human readability.

More formally, accounts in Ethereum are defined as a mapping of an address to an account state consisting of four parts [66]:

nonce The nonce is a natural number used to keep track of the number of transactions issued by this address.

balance This is the number of currency owned by that account in Wei.

storageRoot This is the root hash from a modified Merkle tree built over the data storage associated with this account.

codeHash This is the hash of the smart contract code associated with this account. In contrast to the previous three fields, this one is immutable.

Previously speaking of entities when introducing Ethereum accounts was a deliberate choice. An account does not necessarily represent a person. There are two general types of accounts: externally owned accounts and smart contract accounts. Externally owned accounts can represent a person, company, or similar entity. These accounts have no associated code, and thus their `storageRoot` and `codeHash` fields are empty. Smart contract accounts are initialized with associated program code, also referred to as a smart contract, and that code can use storage. In contrast to externally owned accounts, these can not issue transactions.

2.3.3 Smart Contracts

Smart contracts enable running Turing complete programs on the blockchain. A smart contract account can own currency, and a smart contract can move currency, making them very powerful. From escrow accounts to gambling, they open up a multitude of applications. Contract code execution happens on the Ethereum Virtual Machine

(EVM), a virtual machine run by network nodes. Together with the blockchain, the EVM guarantees that the code is executed correctly. This creates a "world computer" that has no owner and is resistant to manipulation [24].

The EVM is running EVM code, which is a "stack-based bytecode language" [9] not unlike assembly. Apart from the stack, the EVM also has memory in the form of a theoretically infinite byte array. Data from both stack and memory is wiped after a finished computation. Additionally, there is the key-value store associated with every account that is simply referred to as storage. As this storage is part of an account and accounts form the global state, storage is persistent.

In Ethereum, method calls can be performed by both external accounts and smart contract accounts. These calls take the form of a message. External accounts may send messages within a transaction, and smart contracts may trigger messages using the EVM instruction `call`. This entails that every message may lead to more messages being sent. Messages are sent from a sender account to a recipient account, and the recipient is free to send new messages itself, meaning it can call smart contract code from any other account. A message contains the following information [9]: an amount of Ether that is transferred, optional data that takes the role of method parameters, as well as the maximum amount of gas that this call may consume.

The use of the world computer's computational resources incurs a fee, and the resource use is measured in gas. Every EVM instruction has an associated gas cost. The total amount of gas used by a message is calculated by adding up the gas cost of all executed EVM instructions and adding an additional gas fee per byte contained in the message's optional data field. Also, all gas expenses created by subsequently sent messages contribute to the total gas used by a message. These subsequent messages are one reason for every message having to set a maximum amount of gas that may be consumed. Every message is indirectly triggered by a transaction, and its sender has to pay for the used gas. If the set maximum amount of gas is reached, the transaction fails, and the sender has to pay for all of that gas because computational resources have already been expended. Gas has no set value. Instead, every time an externally controlled account issues a transaction, they must set a price in Wei that they are willing to pay per expended unit of gas [66]. If that price is too low, nodes creating new blocks may choose to ignore the transaction.

In general, development using machine code is very time-consuming, and the code is less readable. Because developing smart contracts requires an exceptionally high level of confidence in the correctness of the code, due to them dealing with potentially large sums of money and the fact that smart contracts can not be altered after deployment, they are rarely developed in EVM code. Instead, most smart contracts are written using Solidity, which is an object-oriented high-level language that can be compiled into EVM code [61]. Among other things, Solidity facilitates development by offering support for inheritance, libraries, as well as built-in data structures for accessing transaction and block data.

2.3.4 Ethereum Consensus Algorithm

The blockchain is run by a large peer-to-peer network of nodes. Each node fulfills a different role [24]:

Full Node A full node maintains an up-to-date copy of the entire blockchain and distributes issued transactions and newly created blocks within the network. Additionally, full nodes allow other nodes to download their copy of the blockchain. There is no direct incentive to operate a full node.

Light Node Light nodes try to minimize storage and network resource usage by downloading select state information only, as previously discussed in section 2.3.1. These nodes typically run software dependent on blockchain state information, such as wallet software allowing account owners to issue transactions and query their ether balance.

Miner Miners are full nodes that actively compete to be allowed to create new blocks. This process is referred to as mining and expends computational power. Miners are financially incentivized.

For blockchains, it is essential to control the pace of block creation, as well as who may create a block, to maintain consensus on the global state. There are multiple approaches to this problem, but most successful blockchains, such as Bitcoin and Ethereum, implement proof-of-work (PoW) algorithms. These algorithms rely on the miner computing some kind of token that can be used to prove a certain amount of computational power has been expended and thus authorizes them to create a new block. This is usually achieved by using a cryptographic hash function to find an input leading to a hash satisfying some condition. Because cryptographic hash functions are irreversible, miners have to do this by trial and error. Thus, any miner's chance of creating the next block is proportional to their computational power. As long as 50% of the network's computational power is controlled by honest miners, the network is secure. To incentivize miners, they get a reward per block they create, as well as the transaction fees of all transactions in that block. This block reward is decreasing over time to combat inflation.

Ethereum relies on a PoW algorithm called Ethash. It is far more complex than Bitcoins PoW because it tries to fix one problem Bitcoin experienced over time: Mining centralization. This is partially driven by the development of Application Specific Integrated Circuits (ASICs) that are significantly more efficient to mine on than standard graphics processing units (GPUs) and thus force miners without ASICs out of the network by decreasing their profitability [9]. Ethash tries to address this problem by operating on a large dataset that is regularly updated and increased in size, making available memory size and speed a major bottleneck, which impedes ASIC development [19].

Previously we stated that Ethereum determines the global state by looking for the longest valid chain of blocks. This was slightly simplified. In reality, the chain of valid

blocks that had the most computational effort put into it is selected [9]. These two chains are not necessarily the same ones because of Ethereum's ommer blocks. Ommer blocks are created when two miners publish a block almost simultaneously. One of these blocks will be abandoned, and thus all its transactions will become invalid, making the miner lose out on the block reward and the transaction fees even though they correctly mined the block. Ethereum tries to mitigate this problem and discourage miners from centralizing by paying a reduced block reward to all ommer block miners in each new block. Because of that, ommer blocks are factored in when determining the valid chain that had the most computational effort put into it.

Considering the problems of mining centralization and the exorbitant amount of energy expended for PoW algorithms, Ethereum is planned to transition to a proof-of-stake (PoS) algorithm soon [17]. This algorithm envisions that miners become validators. Validators are required to deposit ether to vote on the validity of new blocks. Correct decisions are rewarded by a percentage-based bonus on the deposit, while failure to validate or malicious behavior can lead to loss of up to the entire deposit.

2.4 TLS-endorsed Smart Contracts

Ethereum smart contract accounts, often just referred to as smart contracts, are only identifiable by their address. Secure distribution of addresses is essential yet difficult. TLS-endorsed smart contracts (TeSC) is a system designed to address this problem by binding smart contracts to domain names [25]. A domain name is easy to check for a human, and the infrastructure to certify domain names is already in place, circumventing the bootstrapping problem.

Before we discuss the specifics of TeSC's design, it needs to be noted that TeSC is still under active development. Everything presented here is based on the state of development in December 2020. TeSC's core component is the endorsement. An endorsement is stored inside a smart contract. It can be split into two sub-components: claim and signature. The claim consists of the smart contract's address, the fully qualified domain name (FQDN) of the smart contract owner, an expiry date, as well as some flags. To produce the signature, all of this information is signed using the private key of the TLS certificate used by the webserver serving the specified FQDN. This signature enables the secure binding of an FQDN to a smart contract address because only the owner of this FQDN and subsequently the certificate can generate such an endorsement.

Management of this endorsement is made possible by creating a new smart contract implementing the TeSC interface. After contract creation, the owner creates the endorsement and proceeds to save it to the smart contract by issuing an appropriate transaction. Afterward, the smart contract is ready for authentication. The TeSC registry, a global smart contract, maintains a list of smart contracts by FQDN to prevent downgrade attacks, which would otherwise be possible as TeSC is an optional security mechanism. Smart contract owners especially valuing privacy, can choose to bind their smart contract to a hash of the FQDN instead to prevent blockchain crawlers from finding their FQDN.

Of course, this option is also supported by the registry.

The last component of TeSC is the off-chain verifier. It is a piece of software responsible for verifying an endorsement found on-chain but is not running on-chain itself. In the following, we elaborate on the functionality of the verifier by illustrating the steps taken in an example. As a simple scenario, we consider a customer wanting to conduct business with an organization while securing that interaction with TeSC. We project that the verifier would typically be part of the wallet software used to issue transactions. The verifier becomes active after the customer has retrieved the organization's smart contract address (e.g., via TeSC Registry, website, etc.) and entered it into the wallet software:

1. Retrieval of the endorsement from the smart contract.
2. Retrieval of the certificate from the website contained within the endorsement's claim.
3. Verification of the endorsement using the certificate.
    a) The endorsement signature has to be produced using the certificate's private key.
    b) The endorsement's smart contract address has to be identical to the smart contract address.
    c) The endorsement's FQDN has to be identical to the organization's known FQDN.
    d) The endorsement's expiry date has not been met yet.
    e) Additionally, flags may need to be considered.
4. Validation of the certificate.

Successfully executing these steps means that the smart contract was successfully authenticated and thus is deemed safe to interact with. This covers the core functionality of TeSC.

3 Related Work

For this thesis, there are several types of related works. The focus of this thesis, TeSC, is a security system working with TLS certificates. We consider other systems in the TLS ecosystem to be related, especially with regard to system requirements. A selection of these systems is presented in section 3.1. Also, there are systems that are designed to provide the same functionality as TeSC. These are in direct competition with TeSC and are listed in section 3.2. Further on in this thesis, we concern ourselves with TeSC's impact on existing attacks on smart contracts. For some context, preexisting research on smart contract vulnerabilities is summed up in section 3.3. This thesis is concerned with a subset of smart contract vulnerabilities. Thus, we examine some existing work on the topic of smart contract vulnerability analysis in section 3.4.

3.1 Established TLS-related Systems

There are many systems in the TLS ecosystem that can be used as examples for successful security systems built on asymmetric cryptography. Apart from TLS itself, which has already been discussed in section 2.2.3, the following sections provide an overview of actively used TLS-related systems.

3.1.1 IPsec

Security Architecture for IP (IPsec) [34] is a competitor of the TLS protocol and was originally introduced in 1995 [3]. Similar to TLS, IPsec can be used with certificates to enable easy authentication of another party. In contrast to TLS, it is built upon the IP (Internet Protocol) and thus a network layer protocol. IPsec is compatible with both IPv4 and IPv6 and provides access control and authenticated encrypted data transfer. It features two operating modes: transport mode and tunnel mode. Transport mode is used to establish end-to-end encryption between two nodes. Tunnel mode can be used similarly to transport mode but is mainly intended to connect two networks by encapsulating packets for transport between them.

3.1.2 DNSSEC

Domain Name System Security Extensions (DNSSEC) [2] is a secure version of the domain name system (DNS), and in its current form, it was introduced in 2005. It is primarily intended to combat man-in-the-middle attacks on DNS queries. DNSSEC does

not use TLS certificates. However, it is very similar to the internet PKI in the way that it creates hierarchical chains of trust across DNS zones.

To secure DNS resource records (RRs), DNSSEC adds signatures enabling the verification of these records. All records of the same type within one DNS zone are bundled into a resource record set (RRset) and signed with the zone signing key (ZSK) to produce a resource record signature (RRSIG). To protect the ZSK, it is signed with a less frequently replaced and longer key signing key (KSK). The public keys used by a zone are stored as DNSKEY RRs and thus have their own RRset with RRSIG. To connect zones, parent zones have delegation signer (DS) RRs that indicate trusted KSKs for child zones. Like every type of RR, DS RRs are also bundled into an RRset and signed. A DNS resolver can use these signatures to authenticate every RR that it receives while resolving a domain name. To successfully begin resolving, the resolver needs to know one first key as a trust anchor.

In addition to authenticated RR, DNSSEC can also provide authenticated denial of existence. For that purpose, DNS servers store their RRs in order (e.g., alphabetically) and keep next secure (NSEC) RRs. An NSEC RR describes a range of non-existent RRs and is returned to a resolver once a record in that range is requested. Because every type of RR is bundled and signed, a resolver can authenticate an NSEC RR just like any other record. This is crucial to counteract denial of service attacks and censorship.

3.1.3 Certificate Transparency

Google spearheaded the development and adoption of Certificate Transparency (CT) [39] in 2013, and today is widely adopted. It was inspired by several high-profile cases of certificate authorities (CA) issuing fraudulent certificates, with the compromise of big Dutch CA DigiNotar being the most well-known case [26]. Instead of taking a preventative approach, CT is designed to combat fraudulent certificates after they have been issued. In the context of TeSC, CT is an especially interesting system because it is considered a possible source for certificate data. This is the reason that this look at CT is going into more depth compared to other presented TLS systems.

The central goal of CT is to ensure that the owner of a domain is aware of all certificates issued for that domain. A misissued certificate would thus be obvious to the domain owner and can consequently be quickly combated. That awareness is achieved through a three-component system:

Certificate Log Log servers hold an append-only list of certificates and can be operated by any interested party. This list is implemented using a Merkle Tree to facilitate proofs (e.g., inclusion or correct behavior). Everyone can submit certificates, but it is expected that CAs submit all newly issued certificates to at least one log themselves. To prevent spam, the log requires all certificates to be submitted with their certificate chain and the chain being rooted in a root certificate contained in the list of acceptable root certificates held by the log server. On successful inclusion of a certificate, the log produces a signed certificate timestamp (SCT) proving that

Endpoint	Method	Arguments	Return Data
/ct/v1/get-sth	HTTPS GET		current Signed Tree Head
/ct/v1/get-entries	HTTPS GET	start, end	Log entries (i.e. certificate data) from start to end, including start and end
/ct/v1/get-roots	HTTPS GET		accepted root certificates

Table 3.1: Certificate Transparency log server API endpoints for data retrieval.

the certificate has been logged. Additionally, everyone can query the log for proof that it is behaving correctly, proof that a certificate has been logged, or a list of newly logged certificates.

Log Monitor Every interested party can run a server constantly monitoring all log servers. For example, a log monitor could be looking for certificates issued for a domain belonging to the monitor's operator and thus detect certificates they are not aware of. Reacting to the discovery of a misissued certificate is up to the monitor's operator and not standardized. Additionally, monitors may even store copies of an entire log.

Certificate Auditor Auditors work with pieces of cryptographic information related to a log. They can be part of a TLS client (e.g., browser) and ensure that every used certificate has been included in a log by obtaining the corresponding SCT. and thus most likely is trustworthy. Alternatively, an auditor can be part of a log monitor or even a standalone service. In that capacity, the auditor constantly verifies the consistency of obtained cryptographic information and thus the correct behavior of the log.

Access to the Log servers is handled via HTTPS API. The servers themselves can be found in different ways. Google provides a list of all logs currently trusted by Chrome at `https://www.gstatic.com/ct/log_list/v2/log_list.json` [27]. In addition to its normal logs, Google also operates a log exclusively for expired certificates and one for certificates with currently not trusted root of trust.

A practice used by almost all log server operators, even though it is not part of the official standard, is temporal sharding [41]. That means that instead of operating a single log server, the operator sets up multiple logs accepting only a subset of certificates. Each of these logs is referred to as a *physical log*, and together they form one *logical log*. Usually, each physical log is assigned one year and rejects all certificates that expire after the end of that year. This practice is supposed to limit the size of logs and reduce the severity of one log server failing.

The API endpoints for all log servers are standardized, and the most important ones for data retrieval are listed in table 3.1.

At the time of writing, there exists a draft for CT Version 2.0 [40]. However, this draft has not yet been approved, which is why this section relies on the older RFC6962

[39]. The proposed changes are mainly changes to used data structures and certificate formats. The core goals and high-level design of the mechanism have not been changed.

3.1.4 CRL

Certificate Revocation Lists (CRL) [13] were first introduced with the specification of X.509 certificates in 1999 [30]. They provide a means to revoke already issued certificates before the end of their originally intended validity period. CRLs are either issued by a CA or a third party entrusted by a CA and have a scope declaring all certificates that could be included. For example, this scope could contain all certificates issued by the CA issuing the CRL. Essentially, a CRL is simply a signed list of serial numbers of revoked certificates. Because these lists are only updated at certain intervals, revocation information might be delayed. Consequently, CRLs cannot definitively answer whether a certificate is valid, only if it has been revoked.

3.1.5 OCSP

The Online Certificate Status Protocol (OCSP) [58] is another certificate revocation mechanism and was introduced in 1999 [47] as an alternative to CRLs. It specifies a service that allows clients to query the revocation status of a certificate from an OCSP responder. This active lookup has the advantage of providing revocation information in a more timely manner than CRLs, but this is achieved at the expense of privacy because the OCSP responder can collect detailed data on every domain accessed by a client. OCSP defines three possible certificate status values: `good`, indicating the certificate has not been revoked, `revoked`, indicating the certificate has been revoked or was never officially issued, and `unknown`, indicating that the responder is not able to provide revocation information for the certificate in question. This last status is mostly applicable to cases where the responder is not providing information for the issuer of the certificate. Additionally, extensions can be used to convey further information.

To minimize the overhead in network usage necessary to establish a TLS connection, OCSP was augmented by a feature commonly referred to as OCSP stapling [58]. It allows clients to indicate that they want to receive certificate status information from the server they connect to during the TLS handshake. Because an OCSP responder always signs its responses, a client can verify that the response it is presented with by a server is authentic.

3.1.6 CRLite

CRLite [38] is the newest effort in certificate revocation and attempts to combine the benefits of CRLs and OCSP. It is still in an early stage, but Firefox has included it in nightly builds at the start of 2020 [33], making future widespread adoption probable. As the name might imply, CRLite is intended to replace CRLs. However, OCSP is still intended to provide a fallback.

Conceptually, CRLite is very similar to CRLs in that it provides clients with a list of revoked certificates. There are a few key differences, though. First of all, the lists are not compiled by CAs but instead generated by third-party servers based on data from Certificate Transparency. These third-party servers could, for example, be operated by browser vendors. The other big differences lie in the improvements CRLite brings to encoding size and update frequency. The encoding is arguably the biggest advantage. Using cascading bloom filters that certificate serial numbers can be tested against, CRLite achieves data size decreases of about two orders of magnitude compared to CRLs.

3.2 Blockchain Account Authentication

TeSC addresses the problem of authenticating smart contract accounts. For this specific problem and even more generally, blockchain account authentication, there are very few proposed solutions. Before addressing these, we will examine several groups of systems that have some intersection with TeSC in terms of features and could be considered competing systems. The examined systems are not limited to the Ethereum ecosystem because while TeSC is designed for Ethereum, it should be portable to any smart contract platform.

One of TeSC's features is the association between account addresses and human-readable names. It decreases the chance of human error and improves usability. This is already implemented by some systems in productive use. Some of these implementations are very narrowly focused: For example, Coinbase, a large cryptocurrency exchange and wallet provider, allows users to send transactions by entering the Coinbase username of the receiver [11]. This is convenient for end-users, but the unfortunate reality is that this requires users to put their trust in Coinbase. Users trust that Coinbase returns the correct account address from their database for the provided username. This turns Coinbase into a single point of failure for every transaction conducted with them. An implementation like this is contradictory to the decentralized nature of cryptocurrencies and also limited to Coinbase users, making it unsuitable as a widespread, long-term solution.

Other implementations aim to provide their service across multiple blockchains. Stellar does so by establishing a new blockchain that is designed to provide a frictionless global exchange of assets [21][22]. These assets are not limited to cryptocurrencies but can include real-world financial assets like US Dollars or gold. To facilitate transfers, Stellar users have human-readable wallet addresses. However, Stellar is an entirely new blockchain that users need to adopt, and the digital assets traded there have one big caveat: apart from Stellar's built-in cryptocurrency, the Lumen, and arbitrary user-created tokens without underlying value, all traded assets there are only a representation of the underlying asset. That is true even for other cryptocurrencies. These digital assets, also referred to as tokens, are issued by Anchors, that have to be accepted by the Stellar Development Foundation. Users have to trust that these Anchors are holding all of the assets that they have issued tokens for and that they will trade in their tokens for the

underlying asset. In total, this is no suitable solution to the problem of inconvenient account addresses due to the complexity of the system and its dependence on third-party financial institutions.

Next, we turn our attention to implementations that have a significantly heavier focus on providing cross-blockchain naming. There are three big efforts to build a blockchain-based domain name system: Blockchain Naming System (BNS) [8], Crypto Name Service (CNS) by Unstoppable Domains [31], and Ethereum Name Service (ENS) [18]. BNS is built on top of the Bitcoin blockchain, while the other two solutions use Ethereum smart contracts. Because Bitcoin does not support smart contracts, BNS runs on Stacks, a smart contract enabled blockchain attached to Bitcoin [1]. Stacks uses the Bitcoin network to settle transactions and pay nodes taking part in its consensus protocol. BNS is a native part of Stacks and implemented by a smart contract contained in the genesis block. While there are some more differences between these three systems in terms of implementation, pricing, and software support, they all issue unique and strongly owned domain names to anyone who pays.

However, these properties, especially strong ownership, can also be regarded as a problem. None of these blockchain-based domain name systems root their distribution of names in the real world. Anyone can register a publicly known name, and for users, it can be as difficult to connect a name to an entity as it is to connect an address to an entity. And because of strong ownership, a company can't gain control of its name through the world's legal systems. Thus, users gain convenience, but not security.

In contrast, TeSC binds account addresses to DNS FQDNs. DNS is trusted worldwide by everyone using the internet, and most users already know the domain names of many real-world entities by heart. By not requiring users to learn a different set of names, TeSC allows this established knowledge to decrease entry barriers for users. Additionally, the domain name system usually stores information about the domain name owner, such as name and payment details at the registrar, or even publicly visible information like a contact person, making it easier to connect a domain to a real-world entity. In total, this helps TeSC to create a strong foundation to establish trust in the names bound to addresses from day one, which is not offered by other discussed name systems.

3.3 Smart Contract Vulnerability Analysis

Many survey papers have been written on the topic of smart contract vulnerabilities. Still, almost all of them are limited to low-level vulnerabilities ranging from contract code to blockchain level (e.g. [20], [16], [54], [4], [28]). Only two of the papers we were able to find address high-level vulnerabilities and attacks at all ([10], [59]). This is congruent with Chen et al. stating in their 2019 paper that "existing studies focus on defending against attacks that attempt to exploit vulnerabilities in the DApp back-end (i.e., smart contracts), but largely ignore the protection of the DApp front-end (i.e., browser) and the interactions between the front-end and the back-end" [10].

In fact, the previously cited survey paper by Chen et al. was the only one we were able to find that considers vulnerabilities on all levels from contract code execution up to the user interface in depth. And while that paper is well structured and significantly more comprehensive than other papers on the topic, it is not without weakness. Vulnerabilities, attacks, and defenses are presented and put in context to each other throughout the paper. Each attack is attributed to a vulnerability, but some of these attributions are short-sighted. For example, the CoinDash hack is attributed to "broken access control" and the Enigma hack to a "weak password". It is true that these are the immediate reasons the attacks were possible. However, both of these high-profile attacks relied on one fundamental weakness in Ethereum's design: it is impossible to authenticate an account securely.

The overwhelming focus on low-level attacks, for which the vulnerabilities are easy to identify, as well as the misidentification of vulnerabilities being used to attack smart contracts, might explain how a weakness as severe as the lack of smart contract authentication has not yet been addressed in a satisfactory manner. In addition, we suspect that there is a research gap on the topic of high-level smart contract vulnerabilities in general.

3.4 Typosquatting Detection

Typosquatting originally referred to the malicious practice of registering domain names that a user could feasibly produce mistakenly while trying to type out a popular domain name. Over time the term has changed to include the registration of domain names that are visually similar but not probable to produce on accident. Several papers have explored the phenomenon and concluded that it is widespread [45][63][35][62]. A single popular domain, such as `google.com`, may be targeted by at least thousands of typosquatting domains [45].

In their 2016 survey paper on the issue, Spaulding et al. explore the different techniques used in domain generation and how attackers try to monetize typosquatting [62]. Most typosquatting domains are generated using simple techniques, such as the omission of a character, switch of two characters, or insertion of an additional character. Although less prevalent, some advanced techniques have been identified as well. Homograph attacks focus on exchanging letters for visually similar ones. Bitsquatting produces domain variations that are probable to be produced by a bit error. Soundquatting exploits similarities in pronunciation to produce typosquatting domains likely to be visited by users being introduced to a domain verbally. And finally, typosquatting cross-site scripting takes advantage of web developers mistyping JavaScript inclusions by serving malicious code at these typosquatting addresses. Existing monetization methods for these typosquatting domains include, but are not limited to, advertising, ransoming domain names, and scams. TeSC will provide a new and potentially very attractive monetization option for typosquatters.

Influential work on typosquatting detection is lacking and often focused on developing

approaches that can be used to automatically gather data for surveys instead of providing client-side protection for users. One of two papers looking very promising is written by Moubayed et al. and presents a machine learning approach to detecting typosquatting [46]. The algorithm requires only the domain in question and impressively limits used features to information inherent in the domain string, such as its length. On closer examination, however, one notices that the main dataset used does not contain any typosquatting examples. Instead, it was created to detect domains generated algorithmically used to control botnets. Unsurprisingly, it is comparatively easy to distinguish a 30 character random string from a normal domain. This sudden and unaddressed shift in subject renders the paper largely irrelevant.

The most advanced approach is introduced by Piredda et al. in their 2017 paper on typosquatting detection [52]. They also leverage machine learning and train several classifiers to detect typosquatting across all domain levels. They conclude that random forest classifiers perform best but still suffer from a high false-positive rate preventing real-world deployment. While their classifier requires just one input, being the domain in question, their approach is of comparative nature. For every original domain, one dedicated classifier is trained with features depending on that original domain. This severely impedes the scalability of the approach.

The overarching trend appears to be combating typosquatting at a low level, such as the internet service provider level [35][52]. Since no effective solution has been found yet, typosquatting still has to be considered an active threat to any system relying on domain names.

4 Analysis

This chapter's primary topic is the evaluation of TeSC's impact on the threats faced by smart contracts. But first, some foundational work is required. So far, TeSC lacks formal requirements, and they become instrumental when considering design changes and augmentations in response to our evaluation. We start out by formulating use-case examples and discussing stakeholders in sections 4.1 and 4.2. Subsequently, we analyze other systems' requirements to motivate and set requirements for TeSC in section 4.3. After that, we finally perform a vulnerability analysis on TeSC in section 4.4, which is crucial for the direction of the remainder of this work. Before ending this chapter, some thoughts on relevant challenges in TeSC's design are presented in sections 4.5 and 4.6.

4.1 Use-Cases

TeSC binds FQDNs to account addresses. Consequently, every use-case for smart contracts is a use-case for TeSC because it does not alter the capabilities of a smart contract. Instead, TeSC augments the smart contract ecosystem because interactions with smart contracts will be simpler and more secure. Simpler, because wallet software could leverage the system to allow users to choose a smart contract by entering an FQDN instead of an account address. At the very least, users no longer need to take long account addresses from multiple sources (e.g., website, email, etc.) and meticulously compare them by hand to ensure they are correct. TeSC is more secure because the binding between FQDN and account address is easily verifiable by anyone and can only be generated by the owner of that FQDN.

We will examine three different use-case examples for TeSC. First, we will look at how an end-user interacts with the system. To that end, we focus on two distinct cases: one where writing to the blockchain is necessary in section 4.1.1 and a read-only case in section 4.1.2. Front-end implementations for TeSC could take many forms, but for now, we focus on integration into the MetaMask [44] wallet software. For that reason, these use-cases involve the usage of MetaMask. With these two cases, we cover all types of intended use-cases.

4.1.1 ICO Investments

Alice wants to invest in the initial coin offering (ICO) of a company. To conduct the ICO, the company owns a smart contract account containing all of the necessary logic. Investors can pay to that smart contract to invest. To advertise this contract, the company website refers to its account address.

Alice owns an externally controlled account with some currency in it that she has set up with the wallet software MetaMask. Using a browser with the MetaMask extension, Alice can visit the company website. Currently, there are two ways to initialize the transaction process:

A) Alice copies the smart contract address listed on the company website, opens the MetaMask user interface, and pastes the address into the address field. Then, Alice proceeds to enter the transaction amount and possibly transaction data.

B) Alice presses a button on the company website that uses a remote procedure call to open MetaMask with a prepared transaction template.

Usually, most website owners will offer option B for the sake of convenience. No matter what option Alice went through, she is presented with a confirmation screen next. Apart from the previously entered transaction data, thanks to TeSC, Alice also gets information on the authentication result of the recipient smart contract, containing at least a state and the endorsing FQDN if applicable. The exact details of this state are to be determined, but a simple version could include the following ones: TeSC not supported, TeSC authentication failed, or TeSC authentication successful. Based on this information, Alice chooses whether to confirm or abort the transaction. Confirmation leads to the signed transaction being published to the network and completes Alice's investment.

4.1.2 Digital Document Verification

Alice applies to a company and supplies a digital version of her university diploma. The university owns a TLS-authenticated smart contract storing hashes of issued diplomas, allowing third parties to verify the authenticity of diplomas they are presented with. Company representative Bob is in charge of processing Alice's application and wants to verify that the diploma she supplied is authentic.

First, Bob's custom diploma verification software needs to obtain the university's smart contract account address, for example, by querying the TeSC registry contract with the well-known FQDN of the university's website. Bob additionally enters some identifier referencing Alice's diploma into his software. The software connects to a full Ethereum node to obtain all of the required data from the blockchain. After confirming that the contract is owned and endorsed by the university, the software proceeds to check the diploma hash and reports the result to Bob. If successful, Bob can be sure of the diploma's authenticity. If not, he can conduct steps deemed appropriate by the company to deal with a possibly faked diploma.

4.2 Stakeholders

Having considered use-cases, we now proceed to categorize all parties involved in the operation of TeSC and define some terminology used throughout the remainder of this

work. Stakeholders are commonly identified by the role they fill. In the context of TeSC, most stakeholders will fulfill multiple distinct roles at once. Because of this, we have chosen to group these roles in a way that we expect stakeholders to take them on typically. We only consider direct stakeholders to keep this overview manageable. TeSC, directly and indirectly, depends on many different systems, such as CT. TeSC's operation is impacting these systems in some way. That impact could be as simple as using bandwidth, making it reasonable to consider stakeholders of these systems stakeholders of TeSC. However, we choose to limit our overview to the entities coming into direct contact with TeSC during everyday operation for the sake of brevity and because they have limited influence over TeSC. The only exception to this rule is CAs because they are of critical importance to TeSC and thus have considerable influence over the system.

Smart Contract Operators

- **Smart Contract Account Owners** are entities that created a smart contract that they want to endorse. In most cases, they still exhibit some control over the smart contract, but that is no requirement. The exact functionality of the smart contract is not important in the context of TeSC. What is important is that the smart contract in question implements the TeSC interface.
- **Domain Owners** are the owners of an FQDN that they also own a TLS certificate for. Ideally, this FQDN is well-known and used to host a website for the owning real-world entity. Using the certificate's private key, they can create an endorsement and add it to the smart contract.

Smart Contract Users (Users)

- **Anonymous Smart Contract Users** interact with a TLS-endorsed smart contract without their own account. They can only perform read operations, meaning they cannot issue transactions.
- ***Ethereum Account Owners*** are in control of at least one account, enabling them to issue transactions to the network and to send or receive currency. This role is unique because not every SCU requires an account. There are use-cases working without one while still profiting from TeSC (refer to section 4.1.2).
- **Verifiers** are entities running software that performs the authentication of smart contracts. They do not require an Ethereum account because verification only requires reading data from the blockchain.

Certificate Authorities (CAs) are an integral part of the Internet PKI responsible for issuing certificates to entities after having confirmed their identity.

Certificate Transparency Log Operators are an important part of CT. TeSC leverages these logs not just for security but also uses them as certificate storage to retrieve certificate data from in case it can not be retrieved directly from a website.

The roles described above do not have to be grouped like this, but we generally expect them to adhere to this grouping. In theory, every role could be fulfilled by its own stakeholder. For the remainder of this thesis, speaking of a role representing a group refers to the entire group unless specified otherwise.

4.3 Requirements Engineering

At the time of writing, TeSC has not been formally characterized by a set of requirements. However, this is crucial to accurately assess any possible threats and subsequently address them. New security mechanisms must not compromise the integral design goals of TeSC. Thus, we will establish a set of requirements for TeSC before conducting further analysis of the system.

4.3.1 Established TLS System Goals

In the following sections, we examine several well-established TLS-related systems with a focus on the requirements set by the system's creators. This process yields a data set that is subsequently used to base TeSC's requirements upon. We believe that this dataset provides valuable insight into the selection and prioritization of requirements, given that all of the examined systems are in widespread active use.

It is difficult to find documents clearly stating system requirements. However, most papers and RFC documents introducing a system standard do provide a section explaining the fundamental goals of the standard. Relying on this information, we have compiled a brief overview:

TLS [15]

- cryptographic security (i.e., confidentiality, authenticity)
- interoperability between network software
- extensibility in terms of encryption algorithms
  - futureproofs TLS
  - does not require new security library
- efficiency in terms of computational effort and network traffic

IPsec [34]

- "provide access control, connectionless integrity, data origin authentication, detection and rejection of replays [...], confidentiality [...], and limited traffic flow confidentiality"
- meet standards of a wide variety of users
- no negative effects on participants that do not use this extension

DNSSEC [2]

- data origin authentication
- ensure data integrity
- provide proof of the non-existence of data

CT [39]

- provide means for detecting certificate misissuance
  - publish all information that is necessary to detect misbehavior of CAs without introducing the need for more trust because logs are designed such that log misbehavior can be easily identified
  - provides no hard guarantees

CRL [13]

- "meet the needs of deterministic, automated identification, authentication, access control, and authorization functions"

OCSP [58]

- provide fast revocation information for a certificate
- allow incorporation of additional status information
- allow responses to be authenticated

OCSP Stapling [51]

- faster handshakes
- efficiency

CRLite [38]

- efficiency
- timeliness
- fail-closed model (no information, no connection)
- privacy
- simple integration into existing systems
- auditability

4.3.2 Common Requirements of TLS Systems

After compiling all of those system goals, we need to condense them into a series of requirements in order to make those system goals comparable. Doing so results in the following alphabetically ordered list of twelve commonly shared requirements:

Table 4.1: This table provides an overview of common TLS-system requirements and what systems satisfy (•) or explicitly intend to satisfy (✓) them. OCSP is grouped together with its extension known as OCSP Stapling.

	TLS	IPsec	DNSSEC	CT	CRL	OCSP	CRLite
auditability				✓			✓
automation	•	•	•		✓	•	•
access control		✓			✓		
confidentiality	✓	✓					
authentication	✓	✓	✓	•	✓	✓	•
data integrity	✓	✓	✓	•	•	•	•
extensibility	✓					✓	
efficiency	✓					✓	✓
interoperability	✓	✓					✓
optional	•	✓	•		•	•	•
privacy	•	•					✓
timeliness						✓	✓

- access control
- auditability
- authentication
- automated
- confidentiality
- data integrity
- efficiency
- extensibility
- interoperability
- optional
- privacy
- timeliness

The matrix in table 4.1 illustrates how these requirements relate to the TLS-systems we previously examined. There are three possible and mutually exclusive relations between one requirement and one TLS-system:

✓ **Emphasized and Satisfied** These are requirements explicitly stated in a respective standard specification document, and this relation is indicated with a checkmark. It has to be noted that we can make no claim to the completeness of the presented matrix with respect to this relation, as there could be additional specification documents that have not been considered as part of this work.

• **Satisfied** The second one is requirements that are satisfied as a primary function of the system but not explicitly stated as a requirement have been marked with a dot. In this context, primary functionality means that the standard alone, without any extensions, or systems that could be built on top of it, is considered. In an effort

to achieve a data set as objective as possible, this relationship is only given if the requirement is satisfied without any doubt or potential for discussion.

No Relation There are, of course, pairs of requirements and systems that are in no relation. In the matrix, these are left blank.

This resulting data set gives us an indication of what requirements are important to design a successful TLS-system, but also what requirements were deemed especially critical by successful system designers. To further focus on this, we can order the requirements by how often they are *emphasized and satisfied* or *emphasized* by a system. This has been done to generate figures 4.1 and 4.2, respectively.

Figure 4.1 is already interesting on its own. We can compare the perceived importance of a requirement according to designers of successful TLS-systems to our own expectations. However, this is highly subjective due to the subjective nature of personal expectations. Instead, we are going to focus on the change in requirement importance going from figure 4.1 to figure 4.2. In the following, we discuss some select observations:

automation With a high degree of automation being very important for a user-friendly system and usability being critical for the successful adoption of a security system, its score in the first figure seems too low. This is confirmed by the drastic increase in score when looking at the second figure. Most likely, this can be explained by automation being so prevalent in the TLS sector that no one thought to emphasize the automatic nature of a new TLS-system.

optional The drastic score increase from the first figure to the second once again implies that making a new security system (i.e., TLS-system) optional is rarely a focus. Nevertheless, most systems end up being optional. We suspect that this factors in to ease adoption.

privacy Privacy is scoring surprisingly low in both figures. This could mean that designers, and people in general, do not care about privacy. More likely, there are trade-offs involved that render a focus on privacy unattractive.

authentication Consistently scoring very high across both figures, authentication clearly is critical in every way. This was to be expected, as one of TLS's most important pieces of functionality is enabling authentication.

4.3.3 Requirements for TeSC

Finally, we discuss the requirements set for TeSC. These requirements are based upon the current specification of TeSC [25] and our previous analysis of other TLS-systems. A short overview can be found at the end of this section.

Before presenting our choices, we need to address requirements that have been consciously excluded. The first of these is *availability*. While availability is important, it is largely influenced by the deployment of the system, not its design. For example, any

Figure 4.1: This diagram shows how often each of the presented requirements was explicitly stated as a design goal for one of the previously discussed TLS-systems.

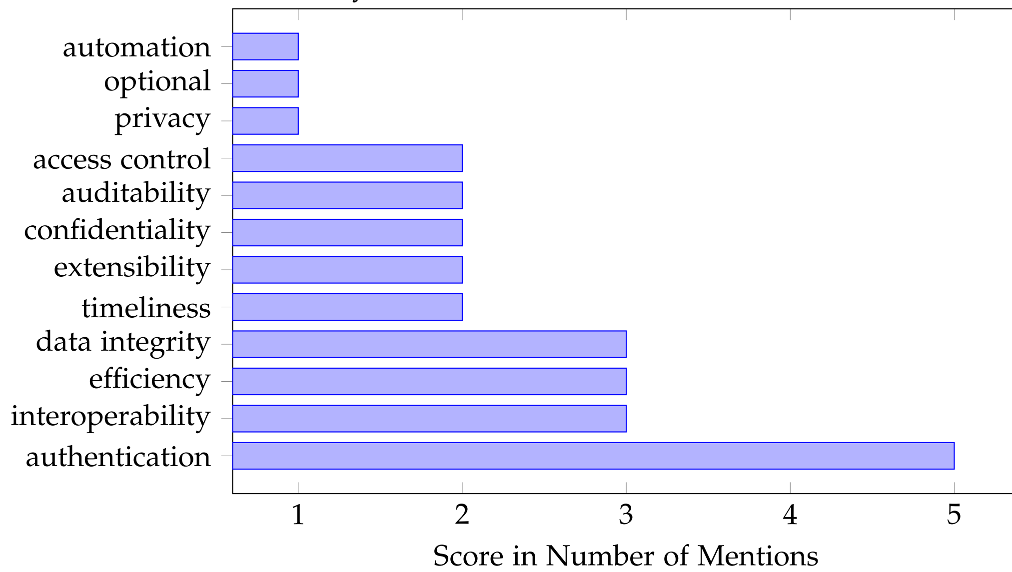
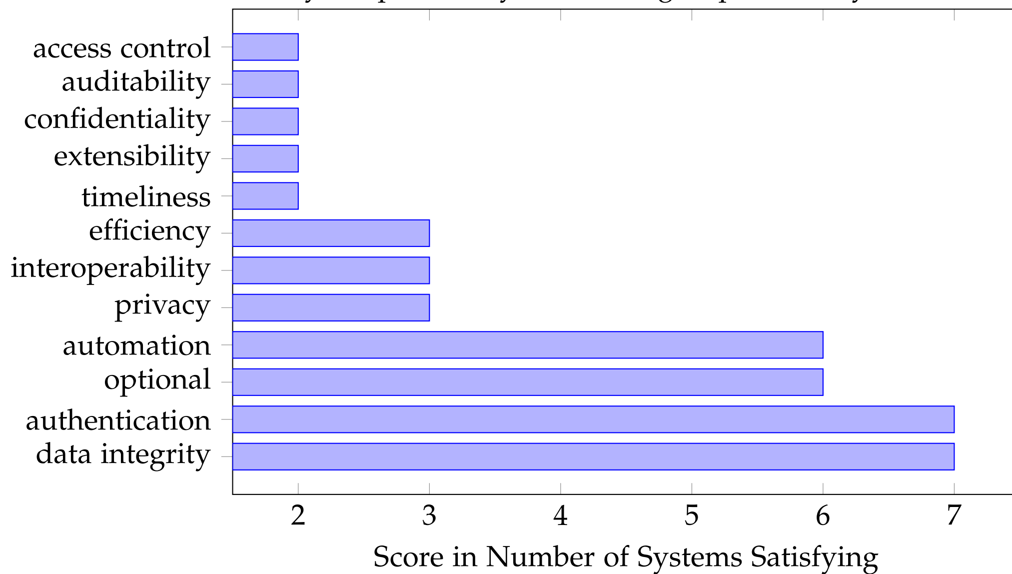


Figure 4.2: This diagram shows how often each of the presented requirements is satisfied by the previously discussed group of TLS-systems.



system can increase availability through horizontal scaling. This might also explain why availability was never mentioned in any of the TLS-system design documents that we previously examined.

Another excluded requirement is the system being *optional*. TeSC does not propose any changes on its underlying blockchain (i.e., Ethereum). It solely relies on the smart contract-functionality provided by the blockchain, making TeSC unable to influence the consensus protocol in any way. Thus, TeSC is optional by its very nature, making it superfluous to add this to the requirements.

Having concluded all necessary introductions, we will now continue by presenting and motivating the chosen requirements in-depth. TeSC's central purpose is the authentication of smart contracts, making this the first and foremost requirement (R1).

Next, we have to consider that adoption is one of the most important factors for any system. Arguably TeSC'S biggest strength is its circumvention of a bootstrapping process and thus its ability to instantly provide benefits to early adopters (R2). This benefit must not be compromised by any change to the system. To retain users, the system also needs to be comfortable to use (R3). While this is partially dependent on frontend implementations for TeSC, the right design decisions are important to pave the way. One example of this is ensuring a largely automated process, like most of the previously examined TLS-systems do. We have purposefully chosen usability instead of automation to avoid confusion because TeSC's core design requires user interaction to either manually input or check an FQDN, depending on the final implementation. The final aspect that will determine TeSC's long-term success is its economic viability, which can be seen as TeSC's efficiency requirement (R4). Computational resources are never free, and that is especially true for the use of the EVM. Due to the variable prices for Ethereum's gas, we can hardly make accurate statements about long-term cost, but we can and must strive to minimize gas usage.

Just like it is for most other TLS-systems, it is essential for TeSC's successful operation that we can guarantee the integrity of all related data (R5). This involves endorsements but also data from the TeSC registry. Furthermore, we want TeSC to be a transparent system, just like the blockchain, it is built upon. Thus, we demand that TeSC is auditable, with a focus on endorsements (R6). For once, this ensures that any attempt at attacking the system is documented permanently and enables responding appropriately. Most importantly, it introduces non-repudiation because it is always possible to exactly retrace who endorsed what contract at and for what time.

Finally, we have some requirements specific to TeSC. Anytime a certificate is validated, the final result of that process depends on the root CA's trusted by the validator (R7). We would like to keep this flexibility with TeSC. This allows users with a deeper understanding of the technology to tweak the validator's root store according to their needs. For very high-value transactions, it might be desirable to use a minimal root store. In general, local validators should have as much control over and insight into their own operation as possible.

Another option for customization offered by TeSC is related to privacy. As a strong

requirement, privacy does impact usability. Nevertheless, TeSC aims to be attractive to all kinds of adopters and therefore has optional privacy features. However, we think that privacy should be completely dropped from the requirements. It complicates TeSC's design, implementation, and operation without providing many benefits. Everyone believing in the transparency of cryptocurrency has no reason to hide that they operate a smart contract. Besides that, anyone can easily obtain a list of popular domains and sub-domains and start brute-forcing hashed domain endorsements.

The requirements set here define what we see as the core of TeSC and should be preserved or even strengthened through possible design adjustments. For quick reference, they are listed in brief form here:

R1: Authentication of smart contracts is TeSC's main objective.

R2: No Bootstrapping is arguably TeSC's biggest strength in comparison to similar systems.

R3: Usability is important not to deter users.

R4: Economic Viability is key to any long-term success.

R5: Data Integrity assurance for all related data is a foundation of TeSC and enables auditability.

R6: Auditability with a focus on endorsements helps to establish trust.

R7: Local Validator Authority places the most important computations into the hands of each user.

4.4 TeSC Vulnerability Analysis

Assessing TeSC is a multi-stage process. As a basis for our analysis, we begin by examining past attacks on smart contract infrastructure in section 4.4.1. Researching how attackers can try to steal cryptocurrency is helping us understand what is preventable using TeSC and what is not. Next, we construct the full set of possible attacks on TLS-endorsed smart contracts in section 4.4.2. This involves exploring how TeSC mitigates possible attacks from the previous section and how TeSC itself may be attacked and circumvented.

To visualize our findings, we use attack trees, also known as Schneier trees [60]. In its most basic form, an attack tree shows an attack goal in the root of the tree and steps required to achieve that goal along the nodes from leaves to root. By default, edges between one parent and multiple child nodes indicate that only one of the child nodes is required to reach the parent, meaning only one step needs to be successfully executed. In other words, the default is an OR relationship. In contrast, an arch connecting multiple edges indicates an AND relationship, and in this case, all child nodes are required to reach the parent node.

Quantitative Value	Qualitative Value
1	easy
2	medium
4	hard
8	virtually impossible

Table 4.2: Initial attack difficulty assignment levels.

Schneier proposes the assignment of values to leaf nodes, which can then be propagated up through the tree [60]. Common examples include the cost and feasibility of an attack approach. For our analysis, we have chosen to introduce a singular measure we call attack difficulty. As we lack plentiful and detailed data about attacks on smart contracts or TeSC, it is not feasible to give accurate assessments of real-world values like the cost of attack. Instead, this attack difficulty measure is intended to provide a relative numerical score that indicates the difficulty an attacker has to overcome to execute a given step of an attack successfully. As such, it is a subjective quantitative measure taking into account rough estimates of cost, time, and skill required to execute a step. We initially assign these difficulties out of four discrete levels, which can be seen in table 4.2. They are not distributed linearly because we believe that one medium and one hard task should not be as difficult as one impossible task. The measure is then propagated up the tree, such that each intermediary node shows the minimum difficulty involved in reaching it.

For this work, the attack trees have been syntactically augmented. For the sake of clarity, trees are depicted in the form of directed acyclic graphs (DAGs). This avoids duplicate subtrees while preserving a clear direction of flow. Every DAG could be expanded into a tree, and for that reason, we continue to refer to our attack DAGs as attack trees.

4.4.1 Attacking Ethereum Smart Contracts

To accurately assess the impact of TeSC, we first examine attacks on smart contracts without TeSC. Most smart contract security research focuses on back-end vulnerabilities on the levels from smart contract code up to and including the blockchain. TeSC, however, is designed to address front-end protection. Thus, we already know that attacks targeting other vulnerability locations will be unaffected by TeSC, allowing us to focus on front-end vulnerabilities and attacks.

We begin our analysis by drawing from a survey paper by Chen et al. [10] that addresses a wide range of vulnerabilities, attacks, and defenses, including front-end ones. The paper classifies these front-end attacks as environment attacks and lists three high-profile attacks as examples that we will use as a starting point for the construction of an attack tree (aka Schneier tree):

CoinDash In 2017, the company CoinDash was running an ICO. Attackers managed to

breach the company website and altered the address of the ICO account that was displayed there. Subsequently, investors were sending their money to the attackers, and Ether valued at $7 Million at the time are lost [65].

Enigma Also, in 2017, the company Enigma was hacked. The CEO's email password was compromised in a previous hack and ended up in online data dumps. Because he never changed it, attackers used it to gain access to his email account and then to Slack and the company website. Using all of these channels, the attackers manipulated the company's customers into sending Ether valued at $500,000 to their account [57].

MyEtherWallet In 2018, Ethereum wallet provider MyEtherWallet suffered a major attack. Abusing a weakness in the web's Border Gateway Protocol (BGP), attackers managed to become the DNS authority on the company's website and redirected customers to a fake site [53]. The obtained customer logins were used to clean out their wallets. According to MyEtherWallet, approximately $150,000 worth of Ethereum was stolen from users [48].

In all of these attacks, the attackers share one simple goal: illegitimately obtaining currency. This goal will be the root of our attack tree. There are other potential goals an attacker might strive towards, such as denial-of-service. We deem these irrelevant for this specific analysis because TeSC most likely can not and is not designed to mitigate these kinds of attacks. When examining TeSC and its vulnerabilities, we will come back to these.

The attack tree based on these three attacks and then further augmented can be seen in figure 4.3. Possible attacks on Ethereum smart contracts can be categorized into three general types. They are highlighted in bold font, and for easier reference, they have each been assigned a Greek character. The attack on CoinDash is classified as an **address replacement attack**. This attack, denoted α, is characterized by the attacker abusing an existing transaction incentive. The attacker does so by silently replacing the advertised address with one of their own. From then on, all business partners of the targeted entity that rely on the advertised address will unknowingly send their funds to the attacker. In the case of CoinDash, that transaction incentive was the opportunity to invest in the company. The address replacement occurred when the attacker breached the company website and replaced the address displayed there. A majority of all subsequently issued transactions meant for CoinDash were sent to the attacker.

The only difficulty in address replacement attacks is the breach of an adequate communication channel. This breach is rated to be hard because only a limited set of entities will be profitable targets, and finding a suitable weakness in a limited set of communication channels, such as websites, requires skill, time, and some luck. Attackers will only have limited time before their attack becomes public knowledge. To maximize profits in that time, the ideal target needs to have a high density of received transaction value with respect to time. Additionally, targets receiving one-time transactions are more

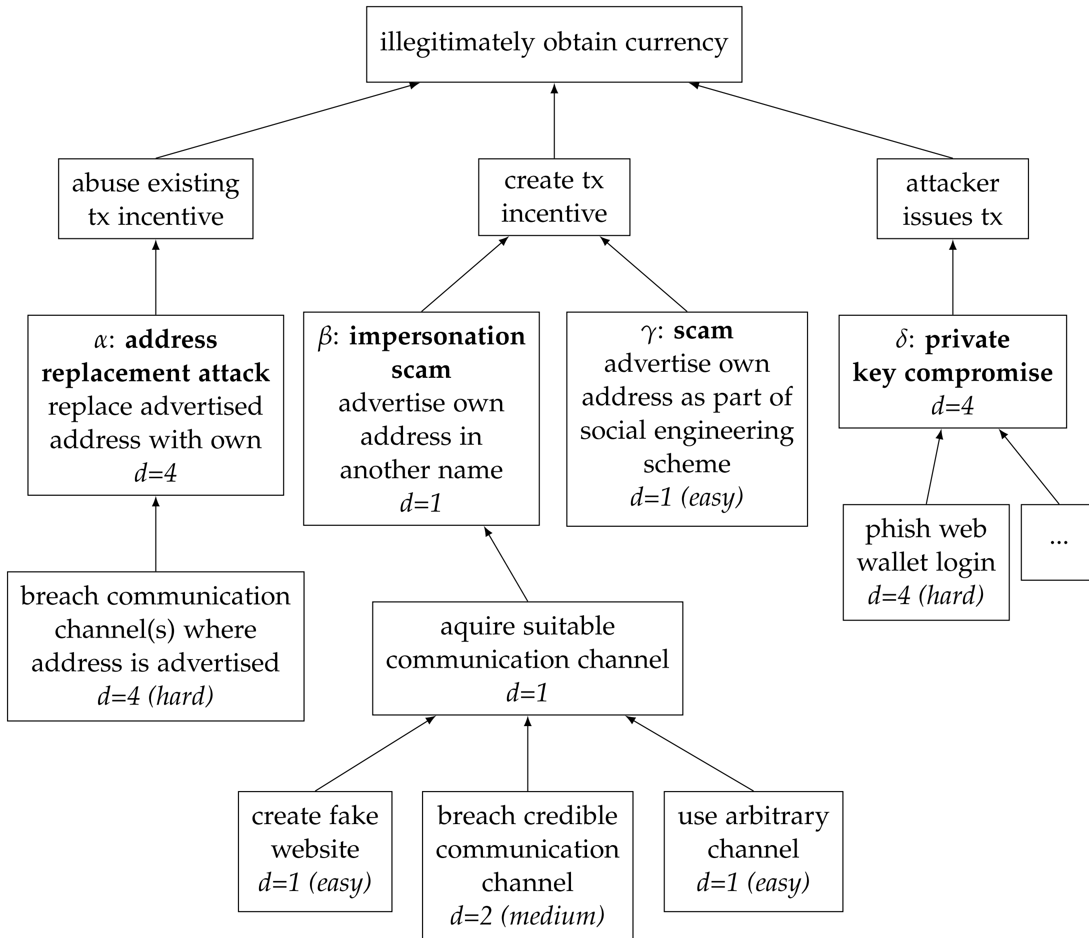


Figure 4.3: Attack tree showing front-end attacks on Ethereum smart contracts.

attractive because that increases the chance of victims not having the correct address saved anywhere. A company conducting an ICO fits this target profile perfectly.

While the attack on Enigma is very similar to the one on CoinDash, there is one difference: the attacker has to create the incentive for victims to issue transactions. This difference makes it a **scam**. More precisely, the Enigma attack is classified as an **impersonation scam**, denoted β, because the attacker assumed the identity of Enigma's CEO. The importance of that latter distinction will become apparent when looking at attacks on TeSC in the next section. In Enigma's case, the incentive used was the announcement of the beginning of a token sale that was planned by the company, but in truth, not scheduled to begin just yet. To maximize their reach, attackers breached and used multiple official communication channels of the company, including the website, Slack, and email.

Attack difficulty for an impersonation scam is very dependant on the attacker's plans. At least one communication channel is required to advertise the scam, but more can be used to increase reach. We classify possible channels into three types. The first one is a fake website, which is easy to create but needs to attract visitors in some way. Similarly, the attacker can easily use any publicly available channel, such as a fake social media profile or a forum post. This option provides increased reach but lends the scam less credibility. Alternatively, attackers can breach a communication channel of a well-known entity, such as a social media profile. This is harder than the other two options but provides both great reach and credibility. The reason that this approach is classified as medium instead of hard is the flexibility in channel and entity selection. Most scam's incentives can be tailored to any impersonated entity easily. For scams without impersonation, the attacker is completely free in their approach and can use any arbitrary channel to advertise. This is the reason for their classification as easy, but this most likely also decreases their effectiveness.

The incentive used in the scam is not part of the difficulty evaluation for two reasons: for one, it is not a technological barrier, and second, it is not as important as the channel it is advertised through. While the incentive used in the Enigma attack was very clever due to it being rooted in some truth and perfectly tailored to the impersonated entity, an incentive does not need to be nearly as sophisticated to be successful. An example is the recent cryptocurrency scam conducted via Twitter. In July 2020, an attacker gained access to several high-profile Twitter accounts via employee access [6]. The accounts were used to send a tweet each, promising to send back double the value of any received transaction. The message itself was largely the same and contained a time limit to create a sense of urgency. This straightforward incentive was sufficient to net the attacker at least $110,000 in Bitcoin [23].

The final class of attacks, denoted δ, groups all forms of **private key compromise**. During this kind of attack, an attacker gains direct or indirect access to a victim's private key and proceeds to transfer all of the victim's funds to the attacker's account. Direct key access is very unlikely but could happen if a computer containing a key file is compromised. The easier option is indirect access, for example, via a third-party web

wallet. This is exactly what happened in the case of the MyEtherWallet attack. Web wallet login details from numerous victims were collected in a phishing attack and used to drain their accounts.

The difficulty of acquiring private keys is immense because everyone owning some cryptocurrency is aware of their importance. If it were not for indirect key compromise, this attack would be rated virtually impossible. But even though it is not impossible, it is still very hard. The attack on MyEtherWallet was technologically impressive, and it would have been even harder if the attackers had decided to improve their success rate by obtaining a fake TLS certificate.

4.4.2 Attacking TeSC

Analogous to the attacks on smart contracts, we can now construct attacks on TLS-endorsed smart contracts. This new attack tree can be seen in figure 4.4. It has been purposefully constructed similarly to the previous attack tree to allow for easy comparison. Attacks γ and δ have been removed to increase visual clarity because these attacks are not impacted by the introduction of TeSC. TeSC can not and is not designed to detect scams not dependant on impersonation, and thus the attacker's approach requires no change. In the case of a private key compromise, nothing can stop an attacker from transferring all funds associated with that key.

Attacks α and β are the attacks addressed by TeSC. The former attack has become more difficult, as attackers have to undertake additional steps to prevent the victim from being warned by the TeSC verifier. Some of the approaches to do so are only possible if the victim initializes the transaction process by providing a contract address. In that case, the verifier can determine the FQDN endorsing the address but can not determine whether that domain belongs to the intended transaction receiver. In figure 4.4, these approaches can be identified by their dotted node border.

Preventing the victim from receiving a TeSC warning is possible in three ways. The first one is performing a downgrade attack. This type of attack is foreseen in TeSC's design and is supposed to be mitigated by the existence of the registry. However, if the verifier is not initiated with an expected FQDN and the provided account address does not support TeSC, then it is impossible for the verifier to query the registry for alternative contract addresses to detect a downgrade attack. Similarly, an attacker's second option is to use an externally owned account address to circumvent TeSC entirely because there is no support for endorsing externally owned accounts. This specific problem has very recently been addressed by an update to the TeSC specification introducing the idea of sub-endorsements. Any TeSC compliant contract could choose to endorse additional addresses. Through this addition, any Ethereum account address could be endorsed, and thus users could expect any account they interact with to support TeSC.

The third and last way for an attacker to defeat TeSC is to create a valid endorsed smart contract. Apart from one exception, this is very difficult, if not virtually impossible. Theft of a target's private TLS key should be virtually impossible given the appropriate handling of sensitive key material. Tricking a target into signing a malicious endorsement

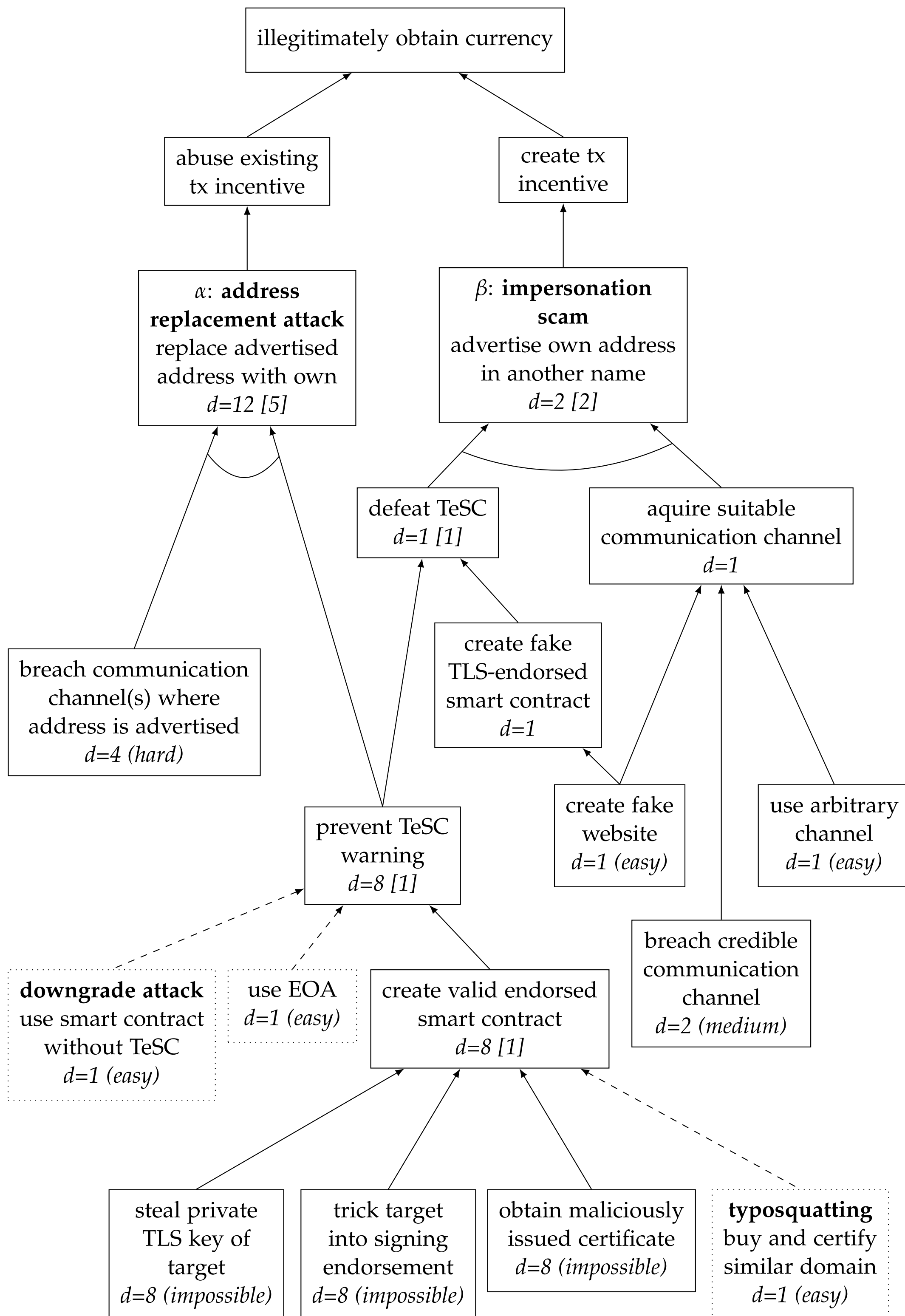


Figure 4.4: Attack tree showing relevant attacks on TLS-endorsed smart contracts.

is virtually impossible as well. There is no handshake protocol we are aware of that would allow an attacker to have the target sign arbitrary data. Thus, the only reasonable cause for a successful attack of this kind is gross human incompetence. Obtaining a maliciously issued TLS certificate is the final virtually impossible attack. While there have been cases of malicious or negligent certificate issuance in the past, the infrastructure is steadily improved and closely monitored. Unfortunately, there is one easy way to create a valid endorsed contract, although limited to transactions initiated by providing an account address only: attackers can register domains that are visually close to a target domain. This approach can be understood as a form of typosquatting. Being the legitimate owner of a typosquatting domain, an attacker can obtain a TLS certificate and create a successfully verifiable TeSC endorsement. Because no expected FQDN was provided, the potential victim has to confirm the correctness of the endorsing FQDN manually. This is prone to errors meaning an attack of this kind is likely profitable at scale.

Attack β needs to defeat TeSC as well to succeed. All of the options previously discussed in the context of attack α also apply to β. In addition, there is one more option. When creating a communication channel for the attack, the attacker can choose to go with a fake website. Preferably, a domain with similarity to the real one is chosen, but this is not strictly necessary. For this domain, the attacker can, of course, legitimately obtain a TLS certificate and use it to create a valid TeSC endorsement.

Overall, both attacks α and β have increased in difficulty because of the introduction of TeSC. Especially the former one is now significantly more difficult. However, factoring in attack paths only possible for transactions initiated with nothing but an address renders that improvement almost negligible. These values are shown within each node in square brackets for comparison. While eventually, transactions via address only might be abolished, for now, they are very common, and these flaws could seriously inhibit TeSC's real-world effectiveness and thus its adoption. Thus, downgrade and typosquatting attacks need to be mitigated.

Finally, the operational stability of TeSC needs to be considered. Attackers can theoretically circumvent the system by performing a denial-of-service attack. An attack tree for this scenario can be seen in figure 4.5. Due to clients running their own verifier, TeSC can only be disrupted by denying clients access to the necessary data. Disrupting either certificate retrieval, endorsement retrieval, or registry access is sufficient. Certificate retrieval is possible via an entity's website or alternatively via any third-party service. Because the certificate can be validated, the only harmful behavior a third-party service can indulge in is denial-of-service. Due to an attacker having to disrupt all sources of certificate data, this is virtually impossible to achieve. Endorsement retrieval is only possible via the Ethereum blockchain, and due to a blockchain's distributed nature, a denial-of-service attack is virtually impossible. Lastly, an attacker can attempt to disrupt registry access by spamming it with useless entries. While this would incur transaction fees, which are especially high for data storage operations like this, it is the most realistic angle of attack and deserves further investigation.

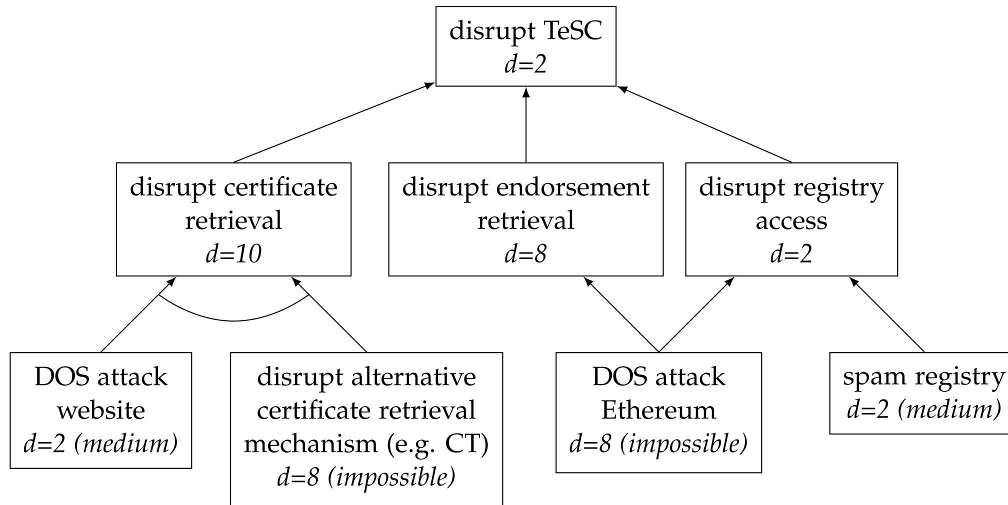


Figure 4.5: Attack tree showing denial-of-service attacks on TLS-endorsed smart contracts.

4.5 Limitations of TeSC

There are some fundamental limitations inherent in TeSC's design. To create an endorsement, ownership of a TLS certificate is required. We believe that this is a reasonable requirement because nearly every entity advertising a smart contract on their website is already in possession of one. Due to TeSC's decentralized design, endorsements have to be part of an account's contract code, which entails that **only** smart contract accounts can be authenticated through TeSC. Individuals transferring currency to friends or another account of their own do not require TeSC, even though they could take advantage of it by creating and using a smart contract account. TeSC is best suited to secure smart contract accounts experiencing a high volume of transactions and transaction value from a large set of senders.

According to TeSC's current design [25], there are no cases where TeSC cannot be used apart from the aforementioned inherent limitations. There are, however, some scenarios that complicate the use of TeSC. Creating an endorsement requires the domain owner to be in direct control of the corresponding TLS certificate, which may not always be the case. For example, a domain owner using a content delivery network (CDN) does not manage their certificate(s) themself. Instead, the CDN Provider does, and thus the domain owner usually has no access to the certificate's private key. However, this problem can be circumvented by obtaining a separate certificate for the domain that is only used to sign endorsements for smart contracts. This entails that the certificate has to be retrieved via CT.

Related to this first scenario is the case of having multiple certificates for one domain. Similarly, it can be solved by using any valid certificate for the endorsement and relying on retrieval via CT. Researching the specifics of CT's design and deployment, we,

unfortunately, have to conclude that it is not viable to use CT for certificate retrieval directly. The number of certificates in circulation is huge, and because of that, log servers are designed with efficiency in mind. To serve their primary function, providing inclusion and consistency proofs, they allow the minimum possible amount of data retrieval required for monitoring and nothing more. The certificate data retrieval endpoint is designed to allow retrieval of certificate data by index range. There is no search functionality, which would be required to retrieve a specific certificate.

The search could be performed by requesting all certificate data and searching on the client, but this is infeasible due to the extreme number of certificates. For example, Google's Argon log for just the year 2020 currently holds nearly 1 billion certificates (as of January 2021). It might be possible to search a log by using the fact that all entries are in order of submission, which should reflect the date of issuance. This date, in turn, is usually close to the `notBefore` date of the validity period of a given certificate. If smart contract endorsements contained this date, it might be possible to develop a search algorithm that is feasible. However, this would be a significant undertaking and is out of scope for this thesis.

As a compromise, CT data can be accessed through third-party services such as `crt.sh`, that aggregate log data and provide search functionality. From a security perspective, this is acceptable because certificates can be validated, and consequently, there is no trust in the service's data integrity required. However, it is possible for the third-party service to withhold certificate data or deny service altogether, making a more direct certificate retrieval solution preferable.

4.6 TeSC Registry Robustness

TeSC's registry smart contract is instrumental in protecting users from downgrade attacks. Additionally, it can aid in the discovery of TeSC-supporting smart contracts and organizations. Currently, the registry's design does not incorporate any form of data deletion. This renders the registry append-only and might lead to performance problems in the future. To gain more insight into the situation, we performed some measurements, which can be seen in figure 4.6. We measured the time it takes to retrieve a number of entries from the registry at regular intervals along a logarithmic scale. Our results imply that the retrieval time grows roughly linearly, which is as good as we could have hoped.

Total retrieval times are acceptable. They do not appear exceedingly high, but it has to be kept in mind that this test was conducted under optimal circumstances with the test software running on the same machine as the test blockchain. Real-world factors like network latency and throughput limits would only worsen these results. For the foreseeable future, considering our use-cases, these retrieval times are acceptable because clients mostly rely on retrieval by domain to find a smart contract or rule out downgrading attacks. Retrieval by domain will probably return around 10 results, which should finish comfortably in under 100ms. Because deploying new smart contracts is

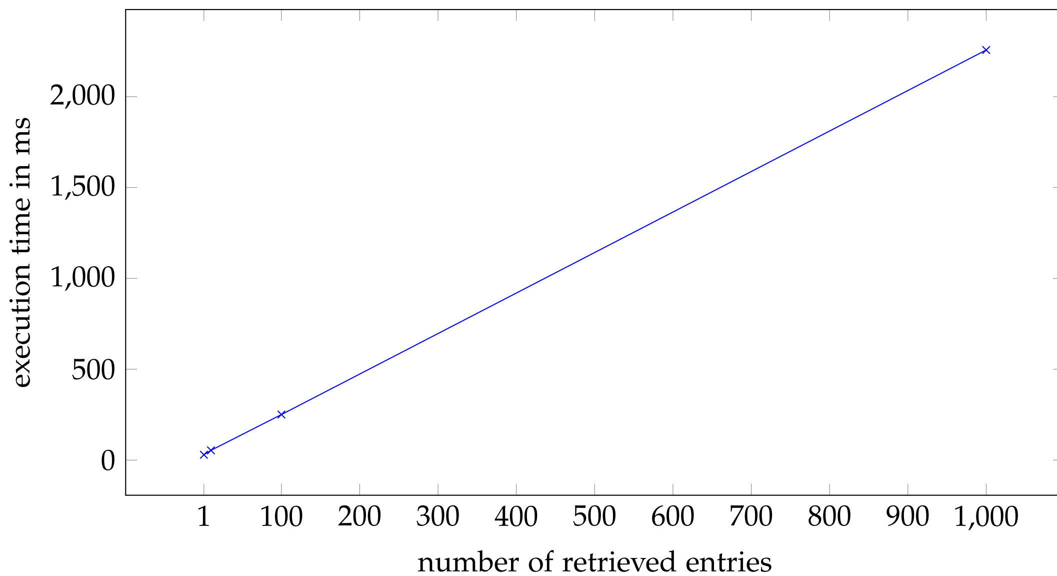


Figure 4.6: Measuring the time taken to retrieve a certain number of entries from the TeSC registry.

expensive, it might take decades of use until some domains have several hundred entries associated with them, creating performance issues. We can not accurately plan for this, as technological progress and blockchain popularity are just some of the unpredictable factors involved. Scanning the entire registry will most likely be infeasible for standard user applications, regardless of whether deleting entries is supported.

The other factor in evaluating the registry's robustness is spam. An attacker could try to fill the registry with large amounts of data to slow down and potentially completely prevent TeSC validator operation. We do not view this as a realistic scenario because this kind of denial-of-service attack is very expensive and would need to be coupled with another attack to be monetizable, adding further difficulty. To illustrate this, consider the cost of only inconveniencing users of smart contracts associated with a single domain. To do so, an attacker would need to add at least 1000 registry entries. Each one of these needs to be unique and refer to a deployed and TeSC compliant smart contract. From our testing we know, that adding one of these entries costs approximately 2 million in gas. Thus, our hypothetical attack would cost about 387,000€ assuming a gas price of 112 Gwei at the current Ether valuation. This could only possibly become a feasible attack if the Ether price were to drop dramatically. In conclusion, no urgent changes to the registry are required.

5 Component Design

After previously identifying typosquatting attacks as one of TeSC's biggest weaknesses, this chapter is dedicated to augmenting TeSC's design by a component that mitigates the risks originating from these attacks. We introduce the high-level component design in section 5.1. In the following two sections, 5.2 and 5.3, we go into greater detail on the two parts of the design.

5.1 High-Level Design

Our final goal is to detect likely cases of typosquatting as part of the TeSC verification process. For this purpose, we require a component taking in an FQDN and returning a boolean value indicating whether that FQDN is likely part of a typosquatting attack. This component is mainly intended to secure transactions initialized only by account address, and in these cases, the FQDN is read from the endorsement. In all other cases, the FQDN is provided on transaction initialization, either directly by user input or indirectly by initialization through a website, and can be passed to the validator from the wallet software. Although mainly intended for the first case, the typosquatting component is beneficial to these other cases as well because fake websites used in impersonation scams will be detected if they are sufficiently similar to an original domain.

There are two distinct approaches to the detection of typosquatting domains: isolated detection and comparison-based detection. The former decides whether a given FQDN is likely typosquatting by solely looking at features of that FQDN, while the latter decides based on comparison with other FQDNs. Isolated detection would most likely be superior in terms of runtime and memory consumption, but by definition, typosquatting is characterized by some form of string similarity, making it exceedingly difficult to obtain an accurate prediction on whether an FQDN is typosquatting without context information. For this reason, we choose a comparison-based approach.

The typosquatting component operates in two phases to limit runtime, memory impact, and the use of network resources. The first phase detects candidate pairs of two domains, of which one might be a typosquatting attack on the other. Each pair consists of the FQDN in question and one other known domain. Known domains can be taken from an internal list of well-known domains and previously verified TLS-endorsed smart contracts. It is limited to verified ones to prevent spam and ensure that the domain and a valid certificate exist. Section 5.2 focuses on that phase in detail. In phase two, which is presented in section 5.3, candidate pairs are further evaluated to determine which of the two FQDNs is most likely original. In case a single candidate pair is found with the

FQDN corresponding to user input not being the most likely original, a typosquatting attack has been detected, and a warning is issued.

Due to the considerable difficulty involved in accurately and quickly detecting typosquatting, the component will inevitably produce some false results. It is our deliberate choice to limit return information as much as possible because of the danger originating from false positives. While it would generally be helpful for users to receive the suspected original FQDN in the event of a detected attack, this could drive a user away from a legitimate smart contract towards a fake one. TeSC as a whole must never endanger users by endorsing possibly malicious actors. Thus, we decide to warn users in the event of a suspected attack to urge them to carefully check their transaction and its recipient, without steering the user's attention towards any particular other smart contract.

In accordance with the requirements of and philosophy behind TeSC, we set the following requirements for this new component:

Language Agnostic The component should serve every user well regardless of their language. If it relied on language-specific metrics, such as character frequency or matching dictionary words, a separate candidate detection model would be required for every language. This is not feasible.

Device Agnostic No assumptions about the user's device must be made. In the context of candidate detection, this is relevant for some metrics. One example is the edit-based fat-finger distance, which assumes a certain keyboard layout to measure the distance between strings [45].

Client Authority All decisions have to be made by the client locally. This ensures transparency of the decision process.

Independence The component must only use openly accessible and trustworthy data without being dependent on any third party. The component's decisions can only ever be as good as the data they are based upon.

Minimal Knowledge Base To facilitate component implementation and future maintenance, the internal knowledge base should be as minimal as possible. That means no lists of trademarks, company names, or geographical locations should be required during component operation. This also minimizes the size of a full TeSC client, saving network bandwidth for downloads and space on user devices.

Real-time Capable The component is called into action for every transaction a user initiates, and significant waiting time will deter user adoption. A typical typosquatting check should finish within one second.

5.2 Typosquatting Candidate Detection

Candidate Detection is tasked with deciding whether a pair of FQDNs is suspiciously similar. This candidate relation is symmetric, making the candidate detection process independent of intra-pair order. Because this detection has to potentially process millions of pairs for the typosquatting component to come to a conclusion on a single domain, some performance considerations have to be made. For this reason, the candidate detection process is limited to all information that can be gained from two FQDNs directly, meaning that external network resources, such as domain name servers, may not be accessed.

In their 2010 typosquatting survey, Moore and Edelman note that a large majority of `.com` domains within a Damerau-Levenshtein distance of one are typosquatting domains [45]. Thus, a simple candidate detection could be solely based on Damerau-Levenshtein distance. While this would suffice to detect the most blatant cases of typosquatting, there would be a considerable rate of false negatives. For example, `turn.de` is a typosquatting variant of `tum.de`, abusing visual similarity of characters and character groups, which may vary depending on the used font. This example has a Damerau-Levenshtein distance of two and would bypass a simple candidate detector. Simply increasing detection to Damerau-Levenshtein distances of two is not feasible because of the steep increase in false positives, especially among short domains. For example, this would mistakenly classify `tum.de` and `lmu.de` as a typosquatting candidate pair. Evidently, setting classification thresholds by hand is no feasible solution to the problem of typosquatting candidate detection.

Instead, we take advantage of machine learning to train a binary classifier automatically. However, this brings up new challenges. For easier training and especially testing, large amounts of labeled data are required. Additionally, we need to select a preferably small set of features providing all of the information required for accurate decisions. These challenges are addressed in sections 5.2.2 and 5.2.3, respectively. But first, we introduce some simplifications in section 5.2.1.

5.2.1 Simplifications

Before diving into further details about the typosquatting detection component, there are two simplifications and their consequences that need to be addressed. First of all, we assume that TeSC endorsements are signed using the highest-level domain in possession of the signer. Usually, this is a second-level domain. Some countries, such as the United Kingdom, have country-code second-level domains, and in those cases, the signer would use their third-level domain. When using the term base domain in the context of typosquatting detection, this is the domain we refer to. Analogously, FQDN refers to the fully qualified domain name consisting of the base domain and all potentially existing higher-level domains properly separated by a dot between them. For example, a smart contract belonging to TUM Department of Informatics would not be signed by `in.tum.de`, but instead by `tum.de`.

This first simplification reduces the search space considerably. However, we understand that this might complicate the setup process for TLS-endorsed smart contracts due to large organization's internal structural rigidity. Also, organizations might prefer to spread their department's smart contracts over different TLS keys, to simplify key replacement and to minimize the impact of one compromised key. Removing this simplification from TeSC would not necessarily require a complete overhaul of the typosquatting detection component. We suspect that typosquatting on lower-level domains is rare because, in most cases, only the owner of a domain is eligible to register sub-domains, and an attacker obtaining that level of access could simply redirect an existing sub-domain. As such, the signing FQDN including all its sub-domains, could be truncated accordingly before entering the typosquatting detection. However, there remains one restriction on the component, which is at least partially a consequence of this simplification: missing-dot typos can not be detected effectively. More details on this can be found in the following section on data preparation.

The other simplification made concerns domain name encoding. We assume that all domains consist solely of Latin characters. Internationalized Domain Names (IDNs) allow the use of Unicode characters in domain names [14]. This is achieved through mapping IDNs to pure ASCII domain names, allowing all established protocols dealing with domain names to continue normal operation. This allows client software like browsers to support different alphabets all over the world with minimal changes to the underlying infrastructure. Unfortunately, it also opens up creative new ways for typosquatting attacks. Some characters from other alphabets are almost indistinguishable from a Latin character and can be substituted to create typosquatting domains that are nearly undetectable for a human. By only processing ASCII domains in TeSC, these Unicode typosquatting domains become obvious to the human eye if manual confirmation is required. Additionally, this simplifies feature selection and thus the design and implementation of the typosquatting detection component. On the downside, this considerably worsens usability for users interacting with IDNs. This is currently acceptable because IDNs still have low adoption rates, with only 8,3 million registered domains in total at the end of 2019 [55].

5.2.2 Data Preparation

Contemporary real-world datasets on typosquatting are scarce, small, and usually limited to certain domains and typo-generation models due to the immense manual labor involved in their creation [52][45]. This also means that it is tough to judge how representative of real-world threats a given dataset is. For these reasons, we generate our own data using a variety of established typo-generation models on a set of well-known domains. This has the advantage of allowing us to produce large quantities of labeled data fully automatically. Moreover, there is no decrease in data quality pertaining to this specific application compared to manually curated datasets since the same methodology is used to create the base sets before manual filtering. This filtering is not required for typosquatting candidate detection because this step focuses only on whether an FQDN

is a feasible typosquatting variation of another FQDN. If and how the FQDN is currently used is irrelevant to the model because it would require external data.

The set of popular domains forming the basis of our dataset is the Majestic Million dataset provided for free online by Majestic [42]. As the name suggests, this set contains the most popular one million websites ranked according to Majestic's own algorithm, which bears similarity to PageRank. This base set is processed into a final dataset consisting of 3-tuples containing two FQDNs, as well as a binary label. Zero is used to indicate a negative pair, meaning unrelated domains, and one indicates a positive, being a case of typosquatting. This labeling is intuitive because it enables the interpretation of classifier outputs as the probability of a positive result. This can be interesting for the purpose of classifier performance analysis, but more importantly, it allows us to tweak the classifier after training by adjusting the confidence threshold.

For our prototype, the final dataset contained 400.000 entries, split evenly between positive and negative entries. Positive entries are generated by randomly picking two distinct FQDNs from the base dataset. Negative entries are created by randomly picking just one FQDN from the base dataset and creating a typosquatting FQDN based on it by applying common typo-generation models. The first group of models is sourced from a survey paper on typosquatting by Spaulding et al. [62]. All of the following typo-generation models are primarily based on human error during an input phase, but they also produce visually similar FQDNs:

Character-omission typo This typo simulates a user missing one character input and deletes one random character from the domain.

Character-permutation typo This type of typo is loosely based on a user confusing input sequencing and switching two adjacent characters. While this tends to happen more often for character pairs entered with different hands, we assume that all of these permutations happen with the same probability and perform one random switch of two adjacent characters of the domain.

Character-duplication typo This next typo is based on a human holding down a key for too long and accidentally producing two characters instead of one. One character from the domain is selected, and a copy of it is inserted directly behind it.

1-mod-inplace This final typo of the group simulates one mistake during input. One character from the domain is replaced with another arbitrarily chosen character from the Latin alphabet.

These typo-generation models have been picked to represent as many changes to a domain as possible. This automatically eliminates models relying on specific input hardware, such as keyboard key proximity. In addition to these basic typo-generation models, two more complex ones are used. These no longer rely on user input errors but instead fully maximize visual similarity:

Homograph attack This model encompasses all changes to a domain abusing visual similarity of certain characters or character groups [29]. As outlined previously, we leave out substitutions based on Unicode characters and focus solely on ASCII substitutions.

Suffix change Lastly, a very simple yet effective typo-generation model is exchanging the top-level domain of a target FQDN with another valid one. This change also extends to country-code second-level domains.

Apart from the suffix change, all of the typo-generation models mentioned so far are applied only to the base domain. All higher-level domains are excluded to avoid generating malformed FQDNs. To generate a typosquatting FQDN, up to two of these typo-generation models are randomly applied to the original FQDN, with the exception of suffix changes, which can only be applied once.

These typo-generation models cover most of the common typosquatting attacks, but there are two models deliberately left out. The first one is the missing-dot typo, which covers cases where a dot separating domain levels is left out [62]. To attack `www.tum.de`, an attacker could register `wwwtum.de`. The second model that is not used is FQDN extension. Attackers sometimes register a domain that extends the original by one thematically plausible word, often separated with a hyphen. For example, to attack `google.com`, an attacker could use `google-billing.com`. These two models are used by attackers, but they are not considered for the dataset because they are very hard for a computer to detect. To illustrate this difficulty, we consider the following pair of two legitimate domains: `munich.de` and `munich-airport.de`. The first one redirects to the website of the city of Munich, and the second one leads to the website of Munich Airport. Obviously, neither of them is a malicious typosquatting attack on the other. The only way this example is distinguishable from the previous malicious one is through context. Enabling an algorithm to make this distinction would require a dictionary per supported language at the very least and an extensive ontology of brands, landmarks, and infrastructure to be truly effective. Similar arguments are applicable to the missing-dot typos, as their detection would require extensive lists of sub-domains and dictionaries of words commonly used in sub-domains. With our requirement to create a lightweight component agnostic of language, this is not reliably detectable. One could also argue that these two typo-generation models are the least effective against a human because the significant difference in length compared to the expected FQDN is easier to notice than a one-character edit.

5.2.3 Feature Selection

The assembled dataset consisting of 3-tuples is not fed directly into the classifier for training. While the labels do not require any further processing, each pair of FQDNs is converted into a numeric feature vector. For feature selection, it is important to keep in mind that candidate detection is designed to be symmetric, implying that all used features must be symmetric. Another factor influencing available features is

the requirement to be language agnostic. For example, this eliminates metrics based on frequency of character or n-gram occurrence. In very basic terms, typosquatting candidate detection is a string comparison problem. Thus, for our features, we largely employ different kinds of string distance metrics:

Average length The average length of the two strings is included to provide the classifier with the information necessary to adjust thresholds depending on string and thus FQDN length.

Damerau-Levenshtein distance Sometimes also referred to as edit distance, this measures the minimum number of atomic edits needed to transform one string into another. The atomic edits are character removal, character insertion, character replacement, and adjacent character switch. It is the only metric that is computed over only the base domain. Excluding TLD changes from the calculation is intended to improve detection of combinations of a change in TLD and one edit to the base domain.

Normalized Damerau-Levenshtein distance Normalization is done by the maximum string length to obtain a distance measure in the range of $[0, 1]$ with a value of 0 indicating equality.

Normalized Jaro-Winkler distance This distance metric is based on matching characters being at roughly similar positions. Characters at the beginning of a string are weighted more heavily [12].

Bi-gram distance Bi-grams consider sub-strings of two consecutive characters. As a token-based metric, n-gram distances compare the sets of unique engrams found in each string. Inspired by Piredda et al., we add special start and end of string characters [52]. But instead of building feature vectors based on the occurrence of unique n-grams, we create a distance metric by calculating the Jaccard index, a set similarity index, between the two sets [32]. This yields a normalized similarity which is then inverted to obtain a normalized distance for consistency.

Quad-gram distance This distance is computed analogously to the bi-gram distance but based on sub-strings of four consecutive characters.

Normalized longest common sub-string The final metric takes the length of the longest common substring (LCS) and normalizes it over the maximum string length.

These specific features were chosen mainly to cover as many different kinds of string distance metrics and by how clearly they show correlation with the labels. Not every metric used might be absolutely necessary, but we are confident that some subset of these metrics is adequate for the task, and finding this subset will be automatically handled during training. Figure 5.1 shows three example histograms illustrating the distribution of values for different features separated by label. The average length does not exhibit

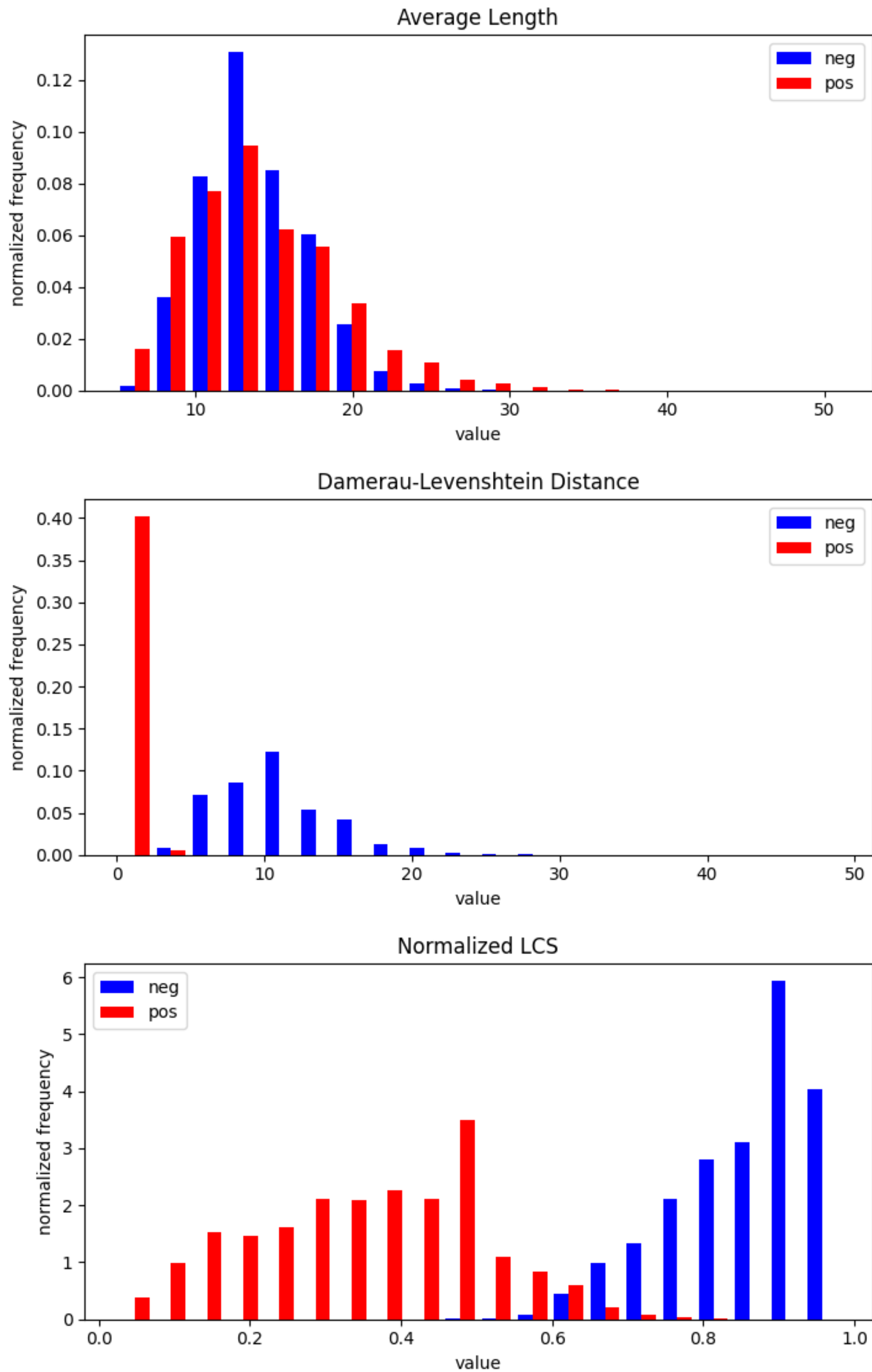


Figure 5.1: Histograms of select features. Data is separated by labels and the positive label indicates a case of typosquatting.

any meaningful separation. It is intended to provide performance improvements in combination with other features. In contrast, Damerau-Levenshtein distance provides good separation, which is to be expected considering the typo-generation models used to generate the dataset. To enable the classifier to achieve even better separation, several other features providing decent separation on their own, such as normalized longest common substring, have been included.

5.3 Typosquatting Candidate Evaluation

Once a typosquatting candidate has been detected, it needs to be further evaluated to determine if a case of typosquatting has been found and which FQDN is the original one. Through several checks leveraging additional data, the algorithm dismisses all candidates that are most likely false positives. These checks can either be successful, indicating a false positive or inconclusive. A candidate is only considered a confirmed attack if all checks can not safely dismiss it, meaning all checks are inconclusive. Thus, the checks are ordered to start with the most simple ones to minimize runtime. In order of execution, they are:

1. Checking if both are on the list of known popular FQDNs.
2. Checking if both FQDNs resolve to the same IP.
3. Checking if both FQDN's TLS certificates have significant overlap.
4. Checking if the FQDN being currently verified is older than the other candidate FQDN.

The first check is trivial. If both FQDNs are part of the verifier's list of popular FQDNs, they are almost certainly not used in a typosquatting attack. Next, if both FQDNs resolve to the same IP address, none of them is attacking the other. Most likely, one is a defensive registration of the other, meaning that the owner of an FQDN preemptively registered similar versions and redirected them to their original site.

If all checks up to this point have failed, the TLS certificates of both FQDN's are compared. It is possible that both FQDN's are covered by the same certificate. In that case, a typosquatting attack can be safely dismissed. If that is not the case, certificate subjects can be compared. If and only if the certificates are extended validation certificates, subjects can be compared. These certificates require extended background checks into the subject's identity and thus establish a binding between a real-world entity and an FQDN. That binding can be interpreted as a transitive affiliation relation, and consequently, matching certificate subjects are grounds for a candidate's dismissal.

Finally, the DNS registry is queried to compare both FQDN's original registration dates. As all other checks have been exhausted, the older FQDN is considered original and the other to be typosquatting. If that original is not the FQDN which is currently

in the process of verification, the typosquatting module has found a likely attack and terminates with the appropriate warning.

One further check was considered yet rejected. Accurate and up-to-date metrics about an FQDN's relevance would aid in improving the typosquatting candidate evaluation. Examples for such metrics include the number of Google search results or Google's number of indexed sites for an FQDN. This could especially help correctly classifying candidates consisting of FQDN's of small or medium popularity. There might exist some cases where two FQDNs are detected as candidates but belong to two similarly popular yet distinct entities. The current candidate evaluation is not well equipped for such cases. Still, we have chosen to exclude such metrics because it is irresponsible to place this much trust into a private company, especially when securing a purposefully distributed system. There is no insight into their algorithms, and the accuracy of received data can not be verified.

6 Component Implementation

A prototype implementation of TeSC has been extended with the component designed in the previous chapter. The preexisting prototype is implemented in the form of a *Node.js* service with API access. Section 6.1 will go into detail about technology choices made considering the preexisting prototype. Then we focus on how the new typosquatting detection component is integrated into the prototype in section 6.2. Finally, section 6.3 elaborates on the internal workings of the component.

6.1 Technology Choices and Tools

The prototype uses a large variety of different technologies and tools. While most of these are replaceable, some are key to the implementation and were chosen purposefully. The first and central one of them is the choice of programming language. Because the prototype is written in *JavaScript*, it is natural to develop the new component in JavaScript as well. This guarantees a uniform code base and facilitates maintenance, as well as possible future modifications. The testing framework Mocha is used for unit tests, and we extend these tests to cover the typosquatting component too.

For the classifier, a well-optimized and thoroughly documented library is required. As we have decided to develop in JavaScript, the library needs to be available for that language and ideally be distributed via the node package manager. This language requirement already limits the amount of machine learning libraries significantly. Documentation for most of the remaining ones is sparse at best, and some are fully implemented in JavaScript, making them rather slow. There is only one library fit for our use case. *TensorFlow* has a JavaScript variant called *TensorFlow.js*. This variant currently has a more limited feature set compared to the base TensorFlow library but is capable of everything required for the typosquatting classifier. Additionally, it brings great versatility allowing the use of different library back-ends with identical feature sets. Developers can choose between a JavaScript implementation, native C++ bindings, or even Nvidia CUDA support. This enables the pursuit of optimal performance on development workstations while ensuring compatibility with a wide range of systems for production use.

6.2 Component Integration

Similar to existing components, such as components for certificate retrieval and validation, the new typosquatting component is largely self-contained. There are two

major points of contact with the overarching verifier. First, the component needs to obtain FQDNs that are associated with previously verified endorsements. Second, the component has to be integrated into the verification flow. That first point of contact is intended to supply the typosquatting candidate detection with relevant additional FQDNs because the component can only ever detect cases of typosquatting if it is aware of the original domain. However, as discussed before, the TeSC registry could potentially grow to an enormous size. For that reason, even the creation of a cache for these FQDNs on verifier startup could be infeasible. We solve this issue by growing a cache organically with every performed verification. This is feasible and most likely does not affect detection capability significantly or adversely, as most attackers are expected to target popular domains, which will be included in the component's internal list of domains.

The typosquatting check is integrated into the verification flow as the very last step for two main reasons. Typosquatting detection is comparatively slow, and every other previous check failing terminates verification early and thus reduces average execution time. Moreover, typosquatting detection and, more specifically, typosquatting candidate evaluation make some assumptions about the input FQDN. Those assumptions, such as its existence and having a valid TLS certificate available, are conveniently covered by previous checks.

The final part in the typosquatting component integration is deciding on a return value policy. The prototype uses a 2-tuple consisting of a `boolean` and a `string`. That first value indicates if the verification was successful, while the second contains additional status information. If no typosquatting attack is detected, no changes to present return values are made. If one is detected, we return `false` as verification value. While the endorsement was verified from a technical standpoint, and there is a small chance of a false positive for typosquatting detection, we feel that it is important to unambiguously communicate that a fail state was reached. Additionally, an error message is returned, explaining that an attack was detected. As discussed earlier, that message must not contain additional information and consequently is static. This makes it easy to parse for front-end software, which could use it to distinguish between more than two verification states if deemed beneficial from a user experience design perspective.

6.3 Component Implementation

The classifier used to detect typosquatting candidates is the core of the implementation. Building an adequate classifier with optimized parameters is vital for the component's success. Thus, we first focus on the creation of an adequate classifier with optimized parameters in section 6.3.1. Then, we continue to present the general internal implementation of the component in section 6.3.2.

6.3.1 Classifier Implementation

We begin building the classifier by outlining the basic constraints. As mentioned before, we build a neural network. The number of network inputs is dictated by the seven features it needs to take in. As discussed earlier, one output neuron producing values in a range of $[0, 1]$ is required. This exact range is achieved through the use of a `sigmoid` activation function, which also helps gravitate output values towards either end of the range. As it is not known whether all features are relevant, the network's first densely connected hidden layer uses a `relu` (rectified linear unit) activation function. In combination with per node biases, this should allow the network to easily eliminate the influence of possibly irrelevant features.

This leaves two open questions: how many more hidden layers should be used in total, and how many neurons per layer? Assuming that all additional hidden layers are equal in size and use a standard `sigmoid` activation, this can be answered experimentally. Ceteris paribus, networks with a different number of hidden layers were evaluated. Then, the same experiment was performed, but with varying layer size instead. For transforming the classifier output into a binary class label, the naive threshold of 0.5 was used because some networks at this stage are still producing outputs clustering around this value, meaning higher thresholds make it harder to distinguish minor differences between networks. The results are illustrated in figure 6.1. Tests for each value are based on four runs using the entirety of the shuffled training set three times each. The presented $f1$ score is the median score of the four runs.

Concerning hidden layers, numbers from 2 up to 8 were tested. The resulting scores are very close to each other, but keeping in mind that we try to improve on the already great separation provided by only considering Damerau-Levenshtein distance, these improvements are valuable. The diagram indicates that there is a point where adding more layers becomes counterproductive. Because the apparent maximum is at a total of 5 hidden layers, the final model will use that number of layers.

Determining the optimal number of neurons per layer is more difficult. The problem space is larger and execution time for tests becomes significant. To counteract this, values are tested with a step size of two. Looking at the results, initially, a roughly linear relation between the number of neurons per layer and network performance can be observed. However, the curve seems to flatten out at around 42, but the data is too noisy to be exact. Nevertheless, 42 will be the number of neurons used for the final model.

This final model is trained for 8 epochs going over the full randomized training set for each epoch. For operation in the typosquatting component, the classification threshold is raised to 0.995 in order to limit false positives.

6.3.2 General Implementation

While the general component implementation closely follows the design specification, some optimizations have been made. One such optimization can be found directly at the beginning of a component invocation. The internal list of popular FQDNs can be

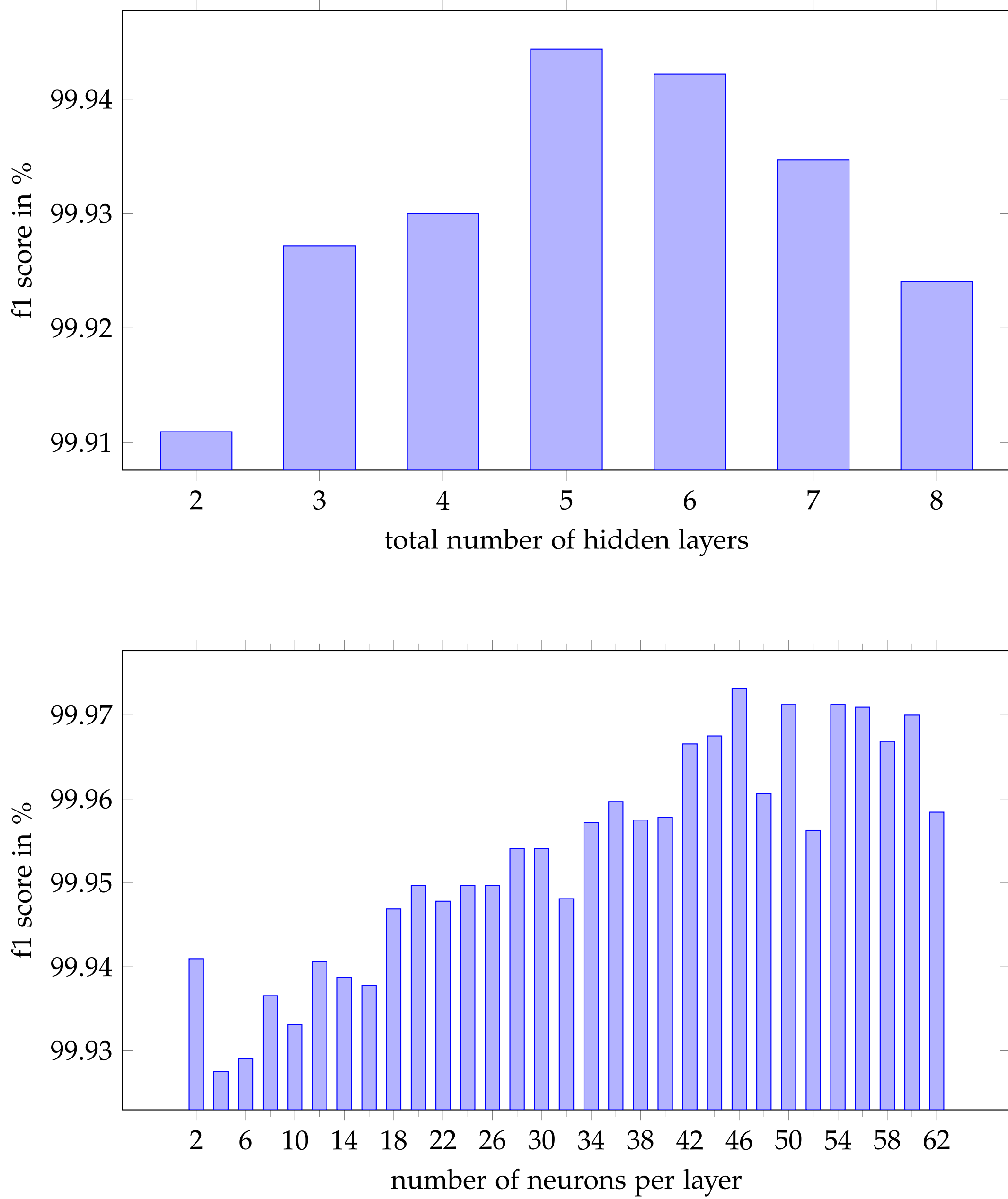


Figure 6.1: Evaluation of the two main network parameters, ceteris paribus. This data is helpful in the construction of an optimized network.

leveraged here to quickly conclude that any input FQDN is benign if it is on that list.

Next, the input FQDN is paired with domains from the internal list and from the registry cache. In order to reduce execution time, it is assumed that all pairs with a difference in length greater than 5 are not typosquatting. Removing these pairs from consideration early reduces upcoming expensive computations. The remaining pairs are converted into feature vectors and then run through the classifier in multi-threaded batches. Batching is intended to decrease required memory and optimize execution time. Depending on the hardware, the classifier will run more efficiently if it receives batched data compared to singular entries. Additionally, the component can halt execution and return after finding one suspicious domain pairing and confirming it. For that reason, all discovered candidate pairs are quickly evaluated to avoid further unnecessary candidate detection.

Coming to a conclusion, the component generates a 2-tuple result similar to the one produced by the TeSC verifier. It includes a boolean status code, which is sufficient to interpret the result of the check. In addition, it also contains a longer message with further details, naming the suspected original domain in the case of a detected attack. This aids in the creation of detailed unit tests and is great for debugging purposes. networks

7 Component Evaluation

After having designed and built the typosquatting component, this chapter is dedicated to evaluating the final product. We have taken several performance metrics, which we present and interpret in section 7.1. We then close this chapter with section 7.2 discussing some qualitative factors.

7.1 Quantitative Evaluation

The core part of the typosquatting component and a decisive factor in its performance is the classifier used for candidate detection. Table 7.1 shows some key metrics on the classifier's performance. For comparison, the table also shows the performance of a classifier of the same layout trained to rely only on Damerau-Levenshtein distance, as this feature provides the best class separation out of all the used features. As can be seen, the classifier performs very well in all commonly used metrics [36]. Accuracy measures the number of correct classifications in relation to all classifications and makes no distinction between the two different types of errors, which are false positives and false negatives. Precision measures the ratio of true positives out of all items classified as positives, while recall is the ratio of true positives to actual positives. As such, precision puts an emphasis on minimizing false positives and recall on minimizing false negatives. The high F1 score, an aggregate of precision and recall with equal weight, indicates superb performance regarding the overall detection of positives. For our model, performance with regards to false positives is at least as important as reliable detection of positives. For that reason, we also included false positive rate as a metric, which we,

Metric	Damerau-Levenshtein Classifier	Final Classifier
Accuracy	0.9886	**0.9953**
Precision	0.9949	**1.0000**
Recall	0.9822	**0.9906**
F1 Score	0.9885	**0.9952**
False Positive Rate	0.004993	**0.00003750**

Table 7.1: Performance evaluation of the final classifier used for typosquatting candidate detection and a comparable one relying solely on Damerau-Levenshtein distance. All metrics are rounded to four significant places and the best result in a metric is highlighted in bold text.

Input Characteristic	Approximate Execution Time
popular FQDN	1ms
typosquatting FQDN	500ms
unpopular FQDN	3500ms

Table 7.2: Execution time measurements for single runs of the typosquatting detection component. Time depends heavily on input. All measurements have been taken on a system featuring an AMD Ryzen 3900X and 32GB of RAM.

of course, strive to minimize in contrast to the other metrics presented so far. The final classifier is able to almost eliminate any false positives on our test set.

Comparing the final classifier to a simple Damerau-Levenshtein distance classifier, the final one is superior in every one of the presented performance metrics. At first sight, this advantage is slim. However, the real-world implications of this advantage are significant. To illustrate this, we reconsider a previous example: `tum.de` and `lmu.de`. The simple classifier falsely classifies this as a positive, but the final classifier classifies it correctly without sacrificing recall.

Execution time is the other crucial performance metric. To assess this, we have timed example runs of the entire component, not just the classifier. These measurements are listed in table 7.2. Execution time varies greatly depending on the input. Popular FQDNs can be found in the internal list of known FQDNs very quickly, completing execution almost instantly. If the input is a typosquatting FQDN, it has to be compared to known domains. Execution time now depends a lot on how popular the targeted domain is because popular ones are tested first. The worst-case in terms of execution time happens if the FQDN is unpopular and not typosquatting. In that case, it has to be compared to all known FQDNs to rule out typosquatting, which takes several seconds. On average, the present execution time is acceptable.

With that worst-case run in mind, we focus on classifier performance once more. As previously stated, the improvement over a simple classifier is significant, even though the metrics may not clearly express that at first sight. The magnitude of this improvement is best conveyed by showing its impact on a run of the full typosquatting component. Table 7.1 shows performance metrics for the classifier itself, but the classifier is not just used once per component run. Depending on the individual run, more than 1 million classifications may be performed. For the sake of a rough calculation, we now look at a run checking an FQDN not on the list of popular domains and assume the classifier has a total of 1000 domains available for comparison, which is far below the real value but produces more manageable values. One false positive is enough to mistakenly conclude that the FQDN is typosquatting if we briefly disregard the existence of candidate evaluation. Thus, the probability of a false positive in the discussed run can be calculated for the simple and final classifiers as follows:

$$p_{error,simple} = 1 - (1 - 0.004993)^{1000} \approx 0.9932$$
$$p_{error,final} = 1 - (1 - 0.00003750)^{1000} \approx 0.0368$$

In this specific case, the component using the simple classifier is virtually guaranteed to be mistaken, while the other one is likely to be correct. Bearing in mind that this calculation is based on experimental data and the scenario is just one of many, it still makes a strong point underscoring the necessity of even the smallest improvement in classifier performance. Even the so far disregarded candidate evaluation can not fix this. In its current form, it is primarily focused on filtering out defensive registrations. Receiving two unrelated FQDNs because of a classification fault, the evaluation will conclude that the one included in the popular domain list is original and conclude that the input FQDN is part of a typosquatting attack, leading to the component mistakenly raising the alarm.

It would be interesting to test the typosquatting component's performance. We expect slightly worse performance than on our test set because of typo generation models this classifier is not built for and the introduction of unpopular domains, which are most likely to produce false positives. Consequently, a dataset for further tests would need to include benign FQDNs that are popular but also unpopular ones. Also, it would include typosquatting domains targeting popular and unpopular FQDNs. Critically, all domains in the set need to exist and possess valid and retrievable TLS certificates. Another important part of creating the data set is deciding whether it is viable to optimize and thus test the component for typical operation. That would entail having test data with realistic proportions of different input groups. We assume that in typical operation, most domains checked will be popular ones, which are easier to classify correctly.

The creation of this dataset would be a thesis on its own, and that, unfortunately, limits our testing capabilities. So far, the component has only been tested as part of the prototype's unit tests. Tests specifically for the component test around 25 inputs created by hand, and all of these tests pass. Yet, it can not be ruled out that these tests contain bias of any form and further component evaluation is necessary.

7.2 Qualitative Evaluation

This new component is designed to assist a user in discovering malicious actors. It is not designed to and can not replace a user's vigilance. With that in mind, there are some further non-numeric factors to consider in order to evaluate the newly built typosquatting component. First off, there are limitations to revisit. The component is not able to detect typosquatting at higher-level domains, as these domains are not processed at all. It is also not designed to detect any form of typosquatting attack that relies on prepending or appending a word to an existing domain, as this would require context information and complicate the detection process significantly. Fortunately, the component is still able to detect some portion of these attacks. This seems to be

Requirement Name	Compliance	Elaboration
Language Agnostic	✓	no language-dependent metrics
Device Agnostic	✓	no device-dependent metrics
Client Authority	✓	completely client-side
Independence	✓	only relies on open infrastructure
Minimal Knowledge Base	✓	only uses set of popular reference domains
Real-time Capable	✓	typical execution time reasonably low

Table 7.3: An overview over previously established requirements for the typosquatting component and its compliance with them.

dependent on domain length. While `wwwtum.de` is not detected, `wwwwikipedia.org` is detected correctly. Limiting the component's false positive rate appears to have caused the component to only detect short domains as typosquatting if they are extremely similar. This explains why these kinds of attacks can only be detected on longer domains and presumably only if the added word is not too long in relation to the total domain length.

A limitation inherent in any comparative detection approach is that typosquatting can only be detected if the original domain is known to the detection component. While our component can not eliminate this problem, the extensive internal list of popular domains should cover a majority of cases. Additionally, the problem is mitigated by also pulling domain data from previously successfully verified TeSC endorsements. This allows the component to become aware of new domains without requiring a software update. Considering the immense difficulty of the underlying problem, the component is performing well and is easy to implement. The only maintenance required is periodically updating the internal list of popular domains, and the component works equally well on domains of every language, provided Latin characters are used.

As a final aspect, we evaluate compliance of the final prototype implementation with previously established requirements. The first set of requirements to be considered are the component's own ones. A brief summary can be seen in table 7.3, indicating that all requirements are met. The component is language and device agnostic due to its choice of metrics, and client authority is at the core of its design. We consider independence given because the component minimizes usage of external data, and all used data is retrieved from core infrastructure, such as DNS. The component's knowledge base is minimal, as it only contains a set of popular domains for reference. And reference data is strictly necessary for any comparison-based detector. Real-time capability is worthy of discussion. We consider this fulfilled because typical execution times introduce no significant delay a user would feel. As part of TeSC, the component obviously has to comply with TeSC's requirements as well. An overview of these is presented in table 7.4. The only requirements affected are authentication, usability, and local validator

Requirement ID	Requirement Name	Compliance	Elaboration
R1	Authentication	✓	slightly improved
R2	No Bootstrapping	✓	not affected
R3	Usability	✓	only slightly slower verification
R4	Economic Viability	✓	not affected
R5	Data Integrity	✓	not affected
R6	Auditability	✓	not affected
R7	Local Validator Authority	✓	preserved

Table 7.4: A look at the requirements of TeSC and how the new typosquatting component is compliant with them.

authority. Authentication is strengthened because of the newly introduced ability to detect typosquatting, usability slightly suffers due to inconvenient worst-case execution time, and local validator authority is preserved. In short, the new component does not violate TeSC's core design requirements while providing a security benefit.

8 Conclusion

In this thesis, we identify, evaluate and mitigate threats to TLS-endorsed smart contracts. Identified significant threats to TeSC are focused on two areas: remaining downgrade attacks and typosquatting. This first threat is largely a question of design philosophy. One could argue that with the inclusion of sub-endorsements, TeSC users should expect every transaction to support TeSC. On the one hand, this is reasonable and could also exert pressure on smart contract owners to quickly adopt TeSC. On the other, it could be divisive, creating a parallel ecosystem consisting of only TeSC users.

Also, as part of our threat evaluation, we have focused on several related topics. First, we examined requirements of TLS-related systems, such as the Online Certificate Status Protocol, to help set and motivate formal requirements for TeSC. Here we advocate for removing TeSC's optional privacy feature because it complicates the design without providing adequate benefit. When looking at existing research on smart contract attacks, we identified a research gap concerning front-end attacks on smart contracts. We also briefly explored TeSC's options for certificate retrieval. We concluded that Certificate Transparency's log servers do not provide feasible access to certificates and that a third-party API service may be used without compromising TeSC's security because retrieved certificates can be validated. Finally, we also examined TeSC's registry as a possible weak point. Our findings indicate that its current design is adequate because data retrieval for common scenarios is sufficiently fast and denial-of-service attacks are very costly.

In this thesis, out of the two initially mentioned two significant threats, we focus on the technical challenge, namely typosquatting. We designed, implemented, and evaluated a new component for the TeSC verifier combating typosquatting. This component is designed to be as simple as possible, universally applicable, and to minimize false positives. Our preliminary evaluation confirms high suitability for the detection of typosquatting attacks at an F1 score of 0.995 and an impressively close to zero false positive rate of 0.0000375. This detection performance is paid for in execution time. Typical execution times are acceptably low, but worst case times can reach up to a couple of seconds, introducing noticeable delay for users. Still, we believe that the presented component is able to provide a valuable service to users and that execution times can be improved with further efforts.

Looking back at our initial research questions, all of them have been answered throughout this thesis, and the results can be summarized briefly as follows:

RQ1 What are actively used security mechanisms for the TLS/SSL certificate infrastructure on the web?
As part of our work, we have examined IPsec, DNSSEC, CT, CRLs, OCSP, and CR-Lite. Gained insight was used to guide the process of setting formal requirements for TeSC that can be referred to for further evaluations and design revisions.

RQ2 What attacks could be performed against TeSC?
TeSC can be attacked in three main ways. An address replacement attack abuses an existing transaction incentive and reroutes funds to an attacker's account by changing an advertised account address. Impersonation scams rely on assuming a trustworthy identity to manipulate victims into sending money to an attacker. Finally, attackers can try to disrupt TeSC's by performing denial-of-service attacks.

RQ3 How can TeSC be augmented to improve its security benefit?
TeSC can profit from protection against typosquatting. This mitigates an easy variant of address replacement attacks and also some impersonation scams, provided that the attacker uses a typosquatting domain.

For the future, there are some challenges left that were discovered or discussed but not solved in this thesis. Obviously, the remaining downgrade attacks should be addressed in some way, even if only by elaborating on the philosophy behind TeSC. Reliable certificate retrieval is another topic that would benefit from further exploration. There might be better solutions than reliance on third-party services aggregating CT data to improve resistance against denial-of-service attacks.

Additional work should be conducted on the TeSC prototype software. Especially verifier and front-end implementations are important for demonstrating TeSC's feasibility. They also provide the basis for future iteration. Clean and well-structured code will make that easier and faster. Working on the prototype also helps to discover implementation-specific questions that are still open. One such example is the matching of domain names. Deciding on what exactly constitutes a match between endorsement FQDN and certificate FQDN is not trivial and has a significant impact on the entire TeSC ecosystem.

Finally, it would be interesting to explore how Extended Validation Certificates could be leveraged by TeSC. While the majority of the TLS infrastructure relies on users being able to connect a domain to a real-world entity, these certificates can be viewed as a direct binding between TLS key and real-world entity.

In conclusion, this work presents an augmentation to TeSC's design and provides valuable insights for TeSC's further development. We are positive that TeSC can provide significant security and convenience benefits to users and, in its final form, has the potential to become the prevalent system for blockchain account authentication.

List of Figures

2.1 Structure of a Merkle tree. 6
2.2 Proof of membership in a Merkle tree. 7
2.3 Different scenarios of certificate path validation. 14
2.4 The basic structure of the Ethereum blockchain. 16

4.1 Histogram of explicitly stated requirements. 38
4.2 Histogram of satisfied requirements. 38
4.3 Attack tree showing front-end attacks on Ethereum smart contracts. . . . 43
4.4 Attack tree showing relevant attacks on TLS-endorsed smart contracts. . 46
4.5 Attack tree showing denial-of-service attacks on TLS-endorsed smart contracts. 48
4.6 Measuring the time taken to retrieve a certain number of entries from the TeSC registry. 50

5.1 Histograms of select features. Data is separated by labels and the positive label indicates a case of typosquatting. 58

6.1 Evaluation of the two main network parameters, ceteris paribus. This data is helpful in the construction of an optimized network. 64

List of Tables

2.1 An example of an X.501 distinguished name. 11
2.2 A list of the top-level fields contained in an X.509 TBSCertificate. 12
2.3 An overview of Ether subdenominations. 18

3.1 Certificate Transparency log server API endpoints for data retrieval. . . . 25

4.1 TLS-systems and their most common requirements. 36
4.2 Initial attack difficulty assignment levels. 41

7.1 Performance evaluation of the final classifier used for typosquatting candidate detection and a comparable one relying solely on Damerau-Levenshtein distance. All metrics are rounded to four significant places and the best result in a metric is highlighted in bold text. 67
7.2 Execution time measurements for single runs of the typosquatting detection component. Time depends heavily on input. All measurements have been taken on a system featuring an AMD Ryzen 3900X and 32GB of RAM. 68
7.3 An overview over previously established requirements for the typosquatting component and its compliance with them. 70
7.4 A look at the requirements of TeSC and how the new typosquatting component is compliant with them. 71

Bibliography

[1] M. Ali. *Stacks 2.0: Apps and Smart Contracts for Bitcoin*. 2020. URL: https://gaia.blockstack.org/hub/1AxyPunHHAHiEffXWESKfbvmBpGQv138Fp/stacks.pdf. (accessed: 03.12.2020).

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. *DNS Security Introduction and Requirements*. RFC 4033. http://www.rfc-editor.org/rfc/rfc4033.txt. RFC Editor, Mar. 2005.

[3] R. Atkinson. *Security Architecture for the Internet Protocol*. RFC 1825. http://www.rfc-editor.org/rfc/rfc1825.txt. RFC Editor, Aug. 1995.

[4] N. Atzei, M. Bartoletti, and T. Cimoli. "A survey of attacks on Ethereum smart contracts." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2017. ISBN: 9783662544549. DOI: 10.1007/978-3-662-54455-6_8.

[5] B. Bambrough. *Exclusive: Twitter Hackers Could Have Stolen A Whole Lot More Bitcoin*. 2020. URL: https://www.forbes.com/sites/billybambrough/2020/07/19/exclusive-twitter-hackers-could-have-stolen-a-whole-lot-more/?sh=6de04c492f84. (accessed: 09.09.2021).

[6] BBC. *Major US Twitter accounts hacked in Bitcoin scam*. 2020. URL: https://www.bbc.com/news/technology-53425822. (accessed: 25.02.2021).

[7] M. Benantar. "The Internet public key infrastructure." In: *IBM Systems Journal* (2001). ISSN: 00188670. DOI: 10.1147/sj.403.0648.

[8] *Blockchain Naming System Documentation*. URL: https://docs.blockstack.org/build-apps/references/bns. (accessed: 18.02.2021).

[9] V. Buterin. *Ethereum White Paper*. URL: https://ethereum.org/en/whitepaper/. (accessed: 03.12.2020).

[10] H. Chen, M. Pendleton, L. Njilla, and S. Xu. *A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses*. 2020. DOI: 10.1145/3391195.

[11] S. Coelho-Prabhu. *Send crypto more easily with Coinbase Wallet*. 2020. URL: https://blog.coinbase.com/send-crypto-more-easily-with-coinbase-wallet-c90a0c84927f. (accessed: 13.02.2021).

[12] W. W. Cohen, P. Ravikumar, S. E. Fienberg, et al. "A Comparison of String Distance Metrics for Name-Matching Tasks." In: *IIWeb*. Vol. 3. 2003, pp. 73–78.

[13] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. http://www.rfc-editor.org/rfc/rfc5280.txt. RFC Editor, May 2008.

[14] A. Costello. *Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)*. RFC 3492. RFC Editor, Mar. 2003.

[15] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. http://www.rfc-editor.org/rfc/rfc5246.txt. RFC Editor, Aug. 2008.

[16] A. Dika and M. Nowostawski. "Security Vulnerabilities in Ethereum Smart Contracts." In: *Proceedings - IEEE 2018 International Congress on Cybermatics: 2018 IEEE Conferences on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, iThings/GreenCom/CPSCom/SmartData/Blockchain/CIT 2018*. 2018. ISBN: 9781538679753. DOI: 10.1109/Cybermatics_2018.2018.00182.

[17] *Ethereum Docs: PROOF-OF-STAKE (POS)*. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/. (accessed: 15.12.2020).

[18] *Ethereum Name Service Documentation*. URL: https://docs.ens.domains/. (accessed: 18.02.2021).

[19] *Ethereum Wiki: Ethash*. URL: https://eth.wiki/en/concepts/ethash/ethash. (accessed: 15.12.2020).

[20] Y. Feng, E. Torlak, and R. Bodik. *Precise Attack Synthesis for Smart Contracts*. 2019. arXiv: 1902.06067.

[21] S. D. Foundation. *Intro to Stellar*. URL: https://www.stellar.org/learn/intro-to-stellar. (accessed: 18.02.2021).

[22] S. D. Foundation. *Stellar Documentation*. URL: https://developers.stellar.org/docs. (accessed: 18.02.2021).

[23] S. Frier and K. Mehrotra. *Twitter Hack Hits Obama, Biden, Musk in Bitcoin Scam*. 2020. URL: https://www.bloomberg.com/news/articles/2020-07-15/elon-musk-bill-gates-appear-to-have-twitter-accounts-hacked. (accessed: 25.02.2021).

[24] U. Gallersdörfer, P. Holl, and F. Matthes. *Blockchain-based Systems Engineering – Lecture Slides*. Oct. 2020.

[25] U. Gallersdörfer and F. Matthes. "TeSC: Mind the Gap between Web and Smart Contracts." In: (2020).

[26] Google. *Certificate Transparency Website*. URL: https://www.certificate-transparency.org/what-is-ct. (accessed: 23.10.2020).

[27] Google. *Certificate Transparency Website: Logs*. URL: https://www.certificate-transparency.org/known-logs. (accessed: 29.10.2020).

[28] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani. “Smart Contract Vulnerability Analysis and Security Audit.” In: *IEEE Network* (2020). ISSN: 1558156X. DOI: 10.1109/MNET.001.1900656.

[29] T. Holgers, D. E. Watson, and S. D. Gribble. “Cutting through the Confusion: A Measurement Study of Homograph Attacks.” In: *USENIX Annual Technical Conference, General Track*. 2006, pp. 261–266.

[30] R. Housley, W. Ford, T. Polk, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*. RFC 2459. http://www.rfc-editor.org/rfc/rfc2459.txt. RFC Editor, Jan. 1999.

[31] U. D. Inc. *Unstoppable Domains: Architecture Overview*. URL: https://docs.unstoppabledomains.com/domain-registry-essentials/architecture-overview. (accessed: 18.02.2021).

[32] P. Jaccard. “The distribution of the flora in the alpine zone. 1.” In: *New phytologist* 11.2 (1912), pp. 37–50.

[33] J. Jones. *Introducing CRLite: All of the Web PKI's revocations, compressed*. 2020. URL: https://blog.mozilla.org/security/2020/01/09/crlite-part-1-all-web-pki-revocations-compressed/. (accessed: 11.02.2021).

[34] S. Kent and K. Seo. *Security Architecture for the Internet Protocol*. RFC 4301. http://www.rfc-editor.org/rfc/rfc4301.txt. RFC Editor, Dec. 2005.

[35] M. T. Khan, X. Huo, Z. Li, and C. Kanich. “Every second counts: Quantifying the negative externalities of cybercrime via typosquatting.” In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 135–150.

[36] M. Kubat. *An introduction to machine learning*. Springer, 2017.

[37] F. Lange, G. Ballet, and A. Toulme. *Ethereum Wire Protocol*. URL: https://github.com/ethereum/devp2p/blob/master/caps/eth.md. (accessed: 13.12.2020).

[38] J. Larisch, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. “CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers.” In: *Proceedings - IEEE Symposium on Security and Privacy*. 2017. ISBN: 9781509055326. DOI: 10.1109/SP.2017.17.

[39] B. Laurie, A. Langley, and E. Kasper. *Certificate Transparency*. RFC 6962. RFC Editor, June 2013.

[40] B. Laurie, A. Langley, E. Kasper, E. Messeri, and R. Stradling. *Certificate Transparency Version 2.0*. Internet-Draft draft-ietf-trans-rfc6962-bis-34. http://www.ietf.org/internet-drafts/draft-ietf-trans-rfc6962-bis-34.txt. IETF Secretariat, Nov. 2019.

[41] V. Lynch. *Scaling CT Logs: Temporal Sharding*. URL: https://www.digicert.com/blog/scaling-certificate-transparency-logs-temporal-sharding/. (accessed: 29.10.2020).

[42] Majestic. *The Majestic Million*. URL: https://majestic.com/reports/majestic-million. (accessed: 27.02.2021).

[43] R. C. Merkle. "A digital signature based on a conventional encryption function." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 1988. ISBN: 9783540187967. DOI: 10.1007/3-540-48184-2_32.

[44] MetaMask. *METAMASK: A crypto wallet & gateway to blockchain apps*. URL: https://metamask.io/. (accessed: 18.01.2021).

[45] T. Moore and B. Edelman. "Measuring the perpetrators and funders of typosquatting." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2010. ISBN: 3642145760. DOI: 10.1007/978-3-642-14577-3_15.

[46] A. Moubayed, M. Injadat, A. Shami, and H. Lutfiyya. "Dns typo-squatting domain detection: A data analytics & machine learning based approach." In: *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2018, pp. 1–7.

[47] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 2560. http://www.rfc-editor.org/rfc/rfc2560.txt. RFC Editor, June 1999.

[48] MyEtherWallet. *A MESSAGE TO OUR COMMUNITY - a response to the DNS HACK of April 24th 2018*. 2018. URL: https://medium.com/@myetherwallet/a-message-to-our-community-a-response-to-the-dns-hack-of-april-24th-2018-26cfe491d31c. (accessed: 25.02.2021).

[49] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." In: (2008).

[50] C. Paar and J. Pelzl. *Understanding Cryptography*. Springer, 2010.

[51] Y. Pettersen. *The Transport Layer Security (TLS) Multiple Certificate Status Request Extension*. RFC 6961. http://www.rfc-editor.org/rfc/rfc6961.txt. RFC Editor, June 2013.

[52] P. Piredda, D. Ariu, B. Biggio, I. Corona, L. Piras, G. Giacinto, and F. Roli. "Deepsquatting: Learning-based typosquatting detection at deeper domain levels." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2017. ISBN: 9783319701684. DOI: 10.1007/978-3-319-70169-1_26.

[53] L. Poinsignon. *BGP leaks and cryptocurrencies*. URL: https://blog.cloudflare.com/bgp-leaks-and-crypto-currencies/. (accessed: 14.01.2021).

[54] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss. *Security Analysis Methods on Ethereum Smart Contract Vulnerabilities - A Survey*. 2019. arXiv: 1908.08605.

[55] I. W. Report. *IDN World Report: Key Metrics*. URL: https://idnworldreport.eu/key-metrics/. (accessed: 27.02.2021).

[56] J. J. Roberts. *Scammer behind massive Twitter hack has made only $109,000—so far*. 2020. URL: https://fortune.com/2020/07/15/twitter-hack-accounts-hacked-elon-musk-bill-gates-joe-biden-kanye-west-bitcoin-who-is-hacker-how-much/. (accessed: 09.09.2021).

[57] J. Russell. *Hackers nab $500,000 as Enigma is compromised weeks before its ICO*. URL: https://techcrunch.com/2017/08/21/hack-enigma-500000-ico/. (accessed: 14.01.2021).

[58] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 6960. http://www.rfc-editor.org/rfc/rfc6960.txt. RFC Editor, June 2013.

[59] S. Sayeed and H. Marco-Gisbert. "On the Effectiveness of Blockchain Against Cryptocurrency Attacks." In: *Proceedings of the UBICOMM* (2018).

[60] B. Schneier. "Attack trees." In: *Dr. Dobb's journal* 24.12 (1999), pp. 21–29.

[61] *Solidity Documentation*. URL: https://docs.soliditylang.org/en/v0.7.4/. (accessed: 15.12.2020).

[62] J. Spaulding, S. Upadhyaya, and A. Mohaisen. "The landscape of domain name typosquatting: Techniques and countermeasures." In: *Proceedings - 2016 11th International Conference on Availability, Reliability and Security, ARES 2016*. 2016. ISBN: 9781509009909. DOI: 10.1109/ARES.2016.84. arXiv: 1603.02767.

[63] J. Szurdi, B. Kocso, G. Cseh, J. Spring, M. Felegyhazi, and C. Kanich. "The long "taile" of typosquatting domain names." In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 191–206.

[64] S. Turner. "Transport layer security." In: *IEEE Internet Computing* (2014). ISSN: 10897801. DOI: 10.1109/MIC.2014.126.

[65] J. Wieczner. *Ethereum: CoinDash ICO Hacked, $7 Million in Ether Stolen*. 2017. URL: https://fortune.com/2017/07/18/ethereum-coindash-ico-hack/.

[66] G. Wood. "Ethereum: a secure decentralised generalised transaction ledger." In: *Ethereum Project Yellow Paper* (Dec. 2020).

[67] K. Zeilenga. *Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names*. RFC 4514. http://www.rfc-editor.org/rfc/rfc4514.txt. RFC Editor, June 2006.