# Access Control for Binary Integrity Protection using Ethereum

Oliver Stengele
Andreas Baumeister
oliver.stengele@kit.edu
andreas.baumeister@partner.kit.edu
Karlsruhe Institute of Technology
Institute of Telematics
Karlsruhe, Germany

Pascal Birnstill
pascal.birnstill@iosb.fraunhofer.de
Fraunhofer IOSB
Karlsruhe, Germany

Hannes Hartenstein
hannes.hartenstein@kit.edu
Karlsruhe Institute of Technology
Institute of Telematics
Karlsruhe, Germany

## ABSTRACT

The integrity of executable binaries is essential to the security of any device that runs them. At best, a manipulated binary can leave the system in question open to attack, and at worst, it can compromise the entire system by itself. In recent years, supply-chain attacks have demonstrated that binaries can even be compromised unbeknownst to their creators. This, in turn, leads to the dissemination of supposedly valid binaries that need to be revoked later.

In this paper, we present and evaluate a concept for publishing and revoking integrity protecting information for binaries, based on the Ethereum Blockchain and its underlying peer-to-peer network. Smart Contracts are used to enforce access control over the publication and revocation of integrity preserving information, whereas the peer-to-peer network serves as a fast, global communication service to keep user clients informed. The Ethereum Blockchain serves as a tamper-evident, publicly-verifiable log of published and revoked binaries. Our implementation incurs costs comparable to registration fees for centralised software distribution platforms but allows publication and revocation of individual binaries within minutes. The proposed concept can be integrated incrementally into existing software distribution platforms, such as package repositories or various app stores.

## KEYWORDS

Blockchain, binary integrity protection, revocation

**ACM Reference Format:**
Oliver Stengele, Andreas Baumeister, Pascal Birnstill, and Hannes Hartenstein. 2019. Access Control for Binary Integrity Protection using Ethereum. In *The 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19), June 3–6, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3322431.3325108

## 1 INTRODUCTION

One of the tenets of the current IT infrastructure is the execution of binaries authored by 'trustworthy strangers'. Not only does this practice carry with it the implicit assumption that the author did not include malicious functionality in the binaries, but also that the binary was not altered between creation and installation or execution. Binary integrity protection aims to prevent the latter. It consists of three key aspects: First, there is the distribution of an executable binary itself, which we will not examine in great detail in this paper. Second, there is the distribution of information that allows anyone to verify the integrity of a particular binary. And third, the recipient of both, binary and integrity protecting information, must be able to determine their authorship and decide whether or not these sources are trustworthy. It is this last aspect that introduces an access control problem.

Today, any given piece of software has an identity that transcends individual binaries. It can exist in multiple versions and on different platforms or devices but still be considered 'one software'. Access control in the context of binary integrity protection revolves around the ability to attach newly created binaries to an established software identity and thereby attesting that they are indeed a new incarnation of said software. In order for this attestation to be useful, each binary must be both uniquely identified and integrity protected. To ensure authenticity and integrity, two mechanisms have established themselves for integrity protection: signatures and hashes.

Signed binaries require a public key infrastructure (PKI) to check the validity of underlying certificates, but offer ideal availability, since the signatures are bundled together with the binaries whose integrity they protect. However, this tight coupling between binary and signature becomes an issue when it comes to revocation. Revoking a certificate can affect multiple binaries and may not even be reliable [5]. Alternatively, hashes are generally stored at a server and queried during the verification of a binary. If the server in question is offline or overwhelmed, the verification cannot be completed. Contrary to signatures, centrally stored hashes allow for very precise revocation. Thus, the two general concepts present different trade-offs between availability and granularity of revocations.

In this paper, we build on the observation that Blockchains provide the elements required for binary integrity protection in general since they themselves have to ensure the integrity of their state information. Two recent papers [2, 9] have started to explore the design space of Blockchain-based binary integrity protection and 'binary transparency'. Our contribution in this paper is twofold:

- First, we present a concept for binary integrity protection based on the Ethereum Blockchain [4] to achieve both, high availability and revocation precision.
- Second, based on this use case, we demonstrate the application of Smart Contracts as a tamper-evident, auditable, and


*SACMAT '19, June 3–6, 2019, Toronto, ON, Canada*

ACM ISBN 978-1-4503-6753-0/19/06…$15.00
https://doi.org/10.1145/3322431.3325108

always invoked access control mechanism. We show that deploying the implementation of the developed Smart Contracts in Ethereum incur costs below $ 10 for setup and below $ 1 for publication and revocation of integrity protecting information.

The paper is structured as follows: in Section 2, we introduce the use case of binary integrity protection and highlight the relevance of access control. We give an introduction to Blockchains in general and Ethereum as a Smart Contract platform in particular, and list related work. We introduce a system model in Section 3 and present our concept in Section 4. Section 5 contains an evaluation of the presented concept alongside a discussion of the previously listed related work. We conclude the paper in Section 6.

## 2 BACKGROUND AND RELATED WORK

In this section, we give a thorough introduction to underlying concepts as well as features and properties of Blockchains and Ethereum followed by a brief introduction of related work.

### 2.1 Blockchain and Ethereum Fundamentals

Binary integrity protection as well as Blockchains build on cryptographic hash functions and digital signatures. Cryptographic hash functions are a way to generate a fixed-sized output given an arbitrary large input with three key properties: the function is deterministic, so any given input always leads to the same output; it is infeasible to find two inputs that produce the same output; given an input, it is infeasible to find a second, different input to produce the same output as the first. So, given the output of a cryptographic hash function, hash for short, and a piece of data whose integrity is in question, one can execute the same hash function on the data and compare the result to the given hash. Only if they are equal is the integrity of the data ensured. In this way, a hash fulfills both the function of a unique identifier and an integrity check.

Digital signatures combine cryptographic hash functions with asymmetric cryptography. Given an asymmetric key pair with a secret key *sk*, which is kept private, and a public key *pk*, a signature is generated by signing a hash with the secret key of the signer. Any recipient of a signature can both compute the same hash function and verify the signature using the public key, compare the results and be convinced of both the integrity and authenticity of the signed data if they match. In the context of Ethereum, identities correspond to public keys and transactions are only valid if they are signed with the corresponding secret key.

We intend to use a Blockchain to represent and manage software identities and binary hashes. The concept of a public, permissionless[1] Blockchain was introduced in 2008 by Satoshi Nakamoto as the underlying data structure of the decentralised cryptocurrency Bitcoin [8]. As a whole, a Blockchain stores and manages state. In the case of Bitcoin, this state relates to outputs of previous transactions and the requirements to spend them. Ethereum [4] abstracts this notion further and allows the definition of arbitrary state machines called Smart Contracts [10]. Both in Bitcoin and Ethereum, changes to the Blockchain state are executed as *transactions* which are bundled into blocks and attached to the established chain, hence the name. Most importantly, every new block references a previous block via a cryptographic hash, thus uniquely identifying it and further securing the integrity of the entire previous chain. For the sake of clarity, we will focus on Ethereum for the remainder of this section, but many of the concepts also apply to other Blockchains.

*Concensus.* Considering that Ethereum is an open system where participants can join and leave at any time, extending the Blockchain while maintaining consensus becomes yet another interesting access control problem. Whichever participant is granted the ability to create the next block is expected to execute all transactions correctly, thus ensuring the validity of the resulting state, and to not abuse his temporary position of power to gain an unfair advantage over other participants. To solve this, a process called *mining* is employed. Essentially, all potential block authors, called *miners*, compete against each other to construct the next valid block by expending a limited resource[2]. While it takes time and effort to find a block, checking the validity of a new block is fast and easy. To compensate the lucky miner for his troubles and to incentivise participation in the first place, each block comes with a reward in newly created Ether, the eponymous cryptocurrency of Ethereum. Additionally, all transactions contained in this new block pay transaction fees to the miner as a further incentive to include as many pending transactions as possible in his block. This randomised write access ensures consensus in the following way. If a miner produces a block that other miners deem invalid, they will not mine on top of it, it will become *stale*, and its author will miss out on both the block reward and any transaction fees, since the longest chain is defined as valid. During this process, miners use a peer-to-peer network to gossip any new transaction or block they receive to their peers in order to keep everyone updated. The bottom line of this construction is the ability of anyone presented with a Blockchain to validate it, starting from the so called *genesis block*, up to the most current block and arrive at the same state as everyone else without trusting anyone. In order to present a convincingly manipulated chain, all blocks between the first altered one and the current block would have to be mined again, due to the hash references linking the blocks together. This feat is made ever more difficult by the size of the peer-to-peer network maintaining the Blockchain. Thus, the Ethereum Blockchain can serve as a highly available, decentralised, and tamper-evident store of information.

*Smart Contracts.* Ethereum serves as a platform for decentralised applications through the use of Smart Contracts, which were first conceptualised by Nick Szabo [10]. Simply put, an Ethereum Smart Contract is a state machine governed by a set of functions that can be called and executed through transactions, either by other Smart Contracts or by human actors. The governing code is stored on the Blockchain in the form of bytecode for the Ethereum Virtual Machine (EVM), a virtualised execution environment that anyone can run. A public key-value database is stored alongside the code on the Blockchain where data can be written to or read from during transactions. This data can also be read from a local copy of the contract through *calls* which do not need to be recorded on the

[1] 'Public' meaning that anyone can read the contents of the Blockchain and 'permissionless' meaning that anyone can participate in maintaining and updating the Blockchain

[2] Computational effort in Proof of Work; temporary control over Ether in Proof of Stake

Blockchain and thus do not incur any fees. Control flow during transactions can depend on both information stored within the contract as well as provided information such as the public key of the caller or the current block number. It is important to note that the integrity of both the key-value database as well as the contract code is protected by the Blockchain. Consequently, the internal state of a Smart Contract can only be altered by its contract code through transactions. As will become clear later, we ensure that hashes can only be published or revoked, but not altered or replaced, by not including any corresponding functionality in the contract code. In a sense, Smart Contracts can be used as a tamper-evident, auditable, and always invoked access control mechanism. The combination of contract code and key-value database is given a unique address during deployment which can be used to reference or call it.

At this point, we will provide a brief distinction between 'tamper-evident' and 'tamper-proof' within the context of Ethereum. While anyone is able to change a local copy of the Ethereum state or the underlying Blockchain, no diligent peer or attentive user should accept it as valid. In this way, the Ethereum Blockchain and everything built on top of it is tamper-evident but achieves practical tamper-proofness through its decentralised architecture and through constant verification by all involved parties. However, since this construction can only provide probabilistic guarantees on its properties, we err on the side of caution and only rely on Ethereum being tamper-evident. This also serves as a reminder to always verify information obtained from a Blockchain.

*Transaction Fees.* Ethereum employs a peculiar mechanism called *Gas* to determine transaction fees which also serves as a defense against denial of service attacks. Every individual operation on the EVM has a certain amount of Gas associated to it [11]. Simple instructions like additions or comparisons are cheap whereas allocating permanent storage for contract code or variables is more expensive. The sender of any transaction specifies a conversion rate from Ether to Gas as well as a maximum amount of Gas that can be used. When mining for a new block, miners select pending transactions to include and execute them locally to determine both the new state of the Blockchain as well as the Gas cost of each transaction. If they succeed in mining a block, then they receive an amount of Ether corresponding to the Gas consumed and the exchange rate set by the sender of each transaction in addition to the aforementioned block reward. If a transaction consumes more Gas than its sender specified, then no state changes are applied but the miner still collects transaction fees. Ethereum enforces a total Gas limit per block which can be adjusted within certain parameters by the miners of each new block. Consequently, there is an upper limit to the complexity of transactions and since Smart Contracts are deployed via transactions, this also imposes a limit to the size of their bytecode.

## 2.2 Related Work

When it comes to binary integrity protection on Blockchains, two prior works have to be noted: Contour [2] and Chainiac [9]. While we introduce the core concepts of both works here, we will discuss them thoroughly in Section 5.

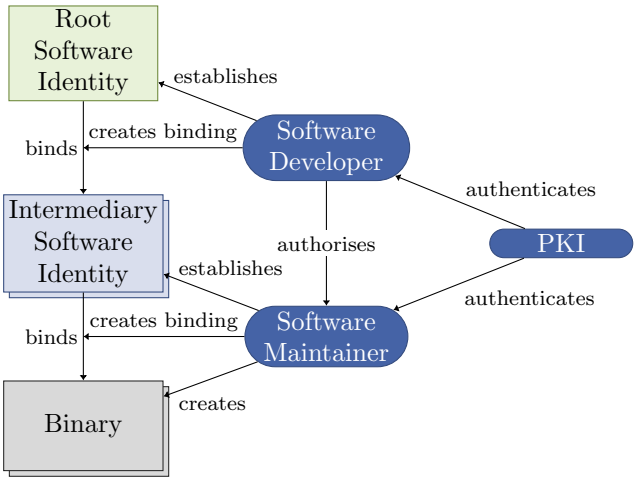


**Figure 1: The hierarchy of software identities down to individual binaries and the relationship and privileges of actors over it. The authorisation of a Maintainer is achieved through the creation of a binding between a Root and Intermediary Software Identity by the corresponding Developer.**

The main idea behind Contour is for the operator of a software distribution platform (SDP) to store cryptographic hashes of binaries in a Merkle tree [7] and publish the root hash on the Bitcoin Blockchain. Together with every binary, a corresponding proof of inclusion and a reference to the Bitcoin block is distributed, thus allowing the user to check its integrity. Note, that the metadata necessary for this integrity check cannot itself be integrity-protected as this would result in a circular dependency. A user must therefore also check the validity of the Merkle tree root hash it is being pointed to.

In contrast to Contours light-weight construction, Chainiac follows a more comprehensive approach. In addition to ensuring binary integrity, it also ensures source-to-binary correspondence through decentralised reproducible builds and binary transparency with an augmented Blockchain called 'Skipchain'. While a Blockchain like Ethereum only links blocks in one direction through hashes, a Skipchain employs references in both directions and across longer distances, similar to a skip list. Software releases are interconnected on the Skipchain in this manner to facilitate easier rollbacks and updates.

In this paper, we explore a different point in the design space with focus on precise revocation, comprehensible access control, and a light-weight architecture on top of Ethereum.

## 3 SYSTEM MODEL

In this section, we introduce a system model describing roles and assets involved in protecting the integrity of binaries. However, we first have to introduce terminology to better describe the aforementioned access control problem. As mentioned at the beginning of the previous section, statements of binary integrity are only useful if they can be attributed to a software identity under the control of a trustworthy developer. To model this access control problem, we define a *Root Software Identity* (Root SI) as an abstract

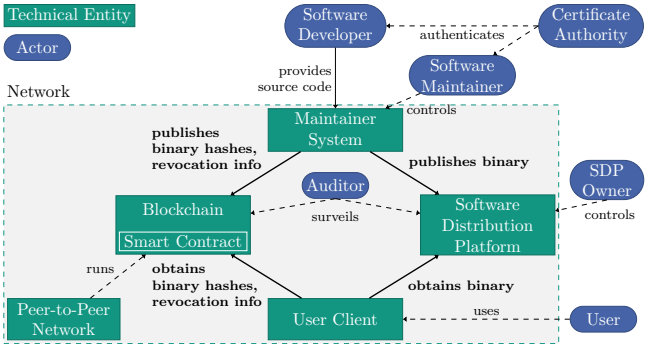


**Figure 2: The collaboration of actors and technical entities relevant for the security model when using a Blockchain to provide binary integrity protection.**

representation of a particular software, independent of any specific binary and an *Intermediary Software Identity* (Intermediary SI) as an organisational construct between a Root SI and binaries. Software Identities and binaries can be bound together as depicted in Figure 1 to form a verifiable access control hierarchy. By establishing bindings between Root SIs and an Intermediary SIs, Developers authorise Maintainers to produce and attest the integrity of binaries of their software. When validating the integrity of binaries, Clients can then traverse this hierarchy to confirm that it leads to the expected Root SI, similar to how certificates are validated by checking if they lead back to a trusted root CA.

## 3.1 Roles and Assets

In Figure 2, we show how binaries and hashes are created, stored, and accessed in order to ensure their integrity using a Blockchain and how actors and technical entities are involved. Generally speaking, we intend our system for storing and managing binary hashes to run in parallel to established distribution paths for binaries. We define the following roles, which can all be assumed by individuals, groups, or organisations:

- A *Software Developer* writes and updates the source code for a given software, establishes and manages the Root Software Identitity, and is allowed to delegate the creation of binaries.
- A *Software Maintainer* builds executable binaries from source code, computes their hashes, and distributes both via their respective pathways. He creates and manages one or more Intermediate Software Identitities which form a verifiable link between a Root SI and any concrete binary.
- An *SDP Operator* runs a software distribution platform, receives binaries from Software Maintainers and offers them for download to Users.
- *Users* verify the integrity of binaries and execute them.
- We also allow *Auditors* to hold Software Maintainers and SDP Operators accountable by pointing out discrepancies between the set of binaries available on an SDP and the corresponding store of hashes.

It is important to note that the roles of Software Developer and Maintainer are not mutually exclusive. Executable binaries can be created by the same entity that provides the underlying source code. In a similar way, multiple Software Maintainers can provide distinct binaries based on the same source code, e.g. for different operating systems or CPU architectures. This is represented by individual Intermediary SIs corresponding to these organisational units which are all bound to the same Root SI. Both Software Developers and Maintainers are identified by at least one public key and authenticate themselves through the use of the corresponding secret key. A PKI can be used to certify the correspondence between public keys and real-world identities.

In terms of assets, our system model includes the following:

- *Binaries* are the main subject whose integrity we intend to protect.
- *Binay Metadata* is attached to each binary and contains information necessary to perform verification. Its integrity is protected alongside the binary.
- *Cryptographic Hashes*, or *hashes* for short, are the means by which we intend to achieve our goal. They are generated by Software Maintainers alongside their corresponding binaries. They must be authentic, integrity-protected, and highly available.
- *Revocation Statements* are issued by Software Maintainers to signify that a specific previously published binary is no longer safe to use. These statements must be authentic and they must propagate quickly and reliably to be effective.

## 3.2 Attacker and Trust Model

The ultimate goal of an attacker is to trick users into executing a manipulated binary at least once. To that end, the attacker is able to alter benign and authentic binaries and publish them on SDPs. Additionally, the attacker is able to create and interact with Smart Contracts like any other individual. The attacker can also compromise all but one key pair of Software Developers and Maintainers.

The attacker is limited in his capabilities when it comes to the Blockchain. While he can participate in the consensus mechanism, he is unable to manipulate it to his advantage or circumvent it entirely, i.e. he is unable to exceed the majority of mining power of the entire Ethereum network. Consequently, it is also impossible for him to alter previous blocks or fabricate a convincing alternate Blockchain to deceive Users. Additionally, it is not possible for the attacker to inconspicuously prevent writing to or reading from the Blockchain by any entity for an extended period of time.

For the sake of completeness, we explicitly state the inherent trust relation between a Software Developer/Maintainer and any User executing their binaries. Users trust the authors of binaries to not intentionally compromise them and cause harm to their systems. Developers extend a similar trust to Maintainers by authorising them to publish binaries of their software. In addition to this preexisting trust relation, we also require Users to trust authors of binaries to issue revocation statements in a timely and reliable fashion if a compromise is found.

In our system, Users, Developers, and Maintainers also enter into a trust relation with the peer-to-peer network maintaining the Ethereum Blockchain. Unlike the previous relation, peers are not trusted individually, but as a group. In general, this means that the majority of mining power is under the control of honest and independent miners. It is worth noting at this point that an SDP

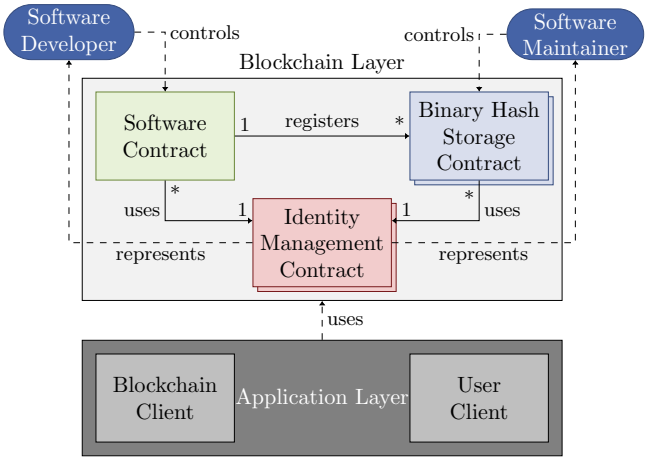


**Figure 3: Visualisation of the layer architecture of the designed concept. In addition to that, the connections between different smart contract types of the concept are depicted.**

owner is not trusted by either Software Developers/Maintainers or by Users, as any misbehaviour would immediately be noticed.

## 4 CONCEPT

With the system model in place, we now present a concept to store and manage binary integrity protecting information on the Ethereum Blockchain. The general structure of our concept is shown in Figure 3. We distinguish between the Blockchain layer, which will be the main focus of this paper, and the application layer, which we only cover briefly. We employ three kinds of Smart Contracts that will be covered in more detail in the following sections.

Common to all contracts is the concept of a *Root Owner* address. This is a special public key which can be used to recover control of a particular contract in case any of the key pairs in active use are lost or compromised. The private key of Root Owner identitites is meant to be stored offline and should only be used in emergencies. All contracts allow the present Root Owner to replace the stored public key. This can be used for both precautionary key rotation as well as for transfer of ownership.

### 4.1 Software Contract

The Software (SW) Contract (Figure 4) establishes a Root Software Identity independent of any particular version or binary. It is created and managed by a Software Developer and functions as a registry for multiple Binary Hash Storage (BHS) Contracts, keyed by their respective SDP ID. By adding references to BHS Contracts, a Software Developer establishes a binding to the corresponding Intermediary Software Identity and thereby authorises a Maintainer to publish binaries of this software on a specific SDP. In addition, this contract also contains the name of the software as human readable metadata. It is worth pointing out that the address of an SW Contract will be used to establish a 'trust on first use' association with the User. While the owner or name of a software may change and BHS Contracts may be added or removed, the address of the Software Contract remains fixed and globally unique.

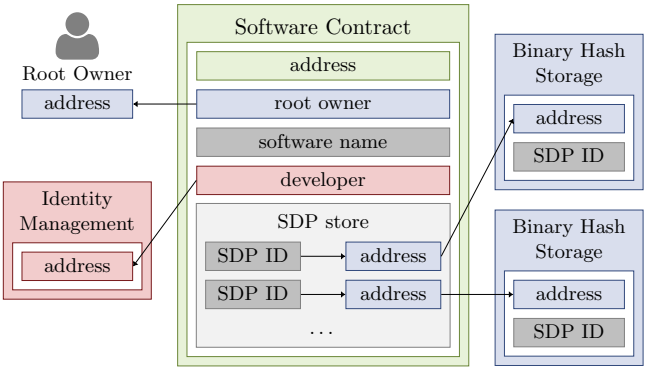


**Figure 4: Inner structure of an SW Contract and relation to other objects.**

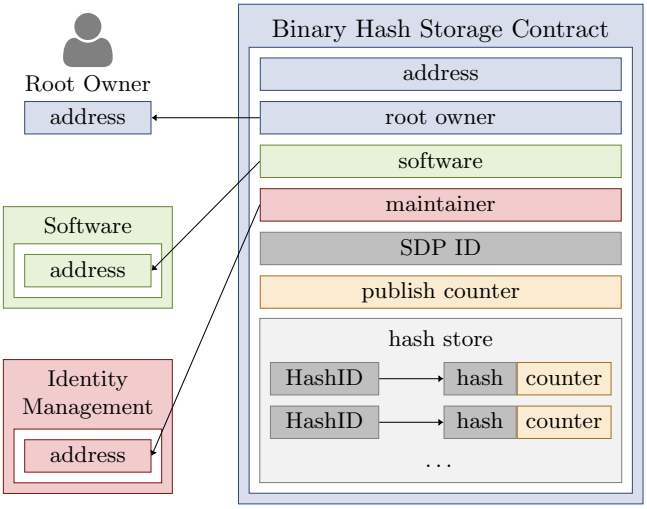


**Figure 5: Inner structure of a BHS Contract and relation to other objects.**

The interface of Software Contracts is presented in Table 1. The most important functions, `registerBHSContract` and `deregisterBHSContract`, revolve around the establishment and rescindment of bindings between Root and Intermediary Software Identities. During validation of a binary, the call `getBHSContract` will be used to check the validity of said binding.

### 4.2 Binary Hash Storage Contract

Similar to Software Contracts, Binary Hash Storage Contracts (Figure 5) establish an Intermediary Software Identity and function as a registry for binary hash statements for all binaries of a particular software on a specific SDP. This registry is indexed by HashID, which is also included in the binary metadata in order to establish a verifiable correspondence between binary and hash. This contract is created and managed by a Software Maintainer and legitimised by a Developer through a reference in the corresponding SW Contract, thereby binding the Intermediary SI to the corresponding Root SI.

**Table 1: Interface of SW Contract. Simple calls to retrieve the values of attributes are omitted.**

| Name | Arguments | Functionality | Allowed Caller |
|---|---|---|---|
| **Transactions** | | | |
| `changeRootOwner` | *Root Owner* (new) | Replaces the stored Root Owner address. | *Root Owner* (current) |
| `setDeveloper` | $\text{IDM}_{addr}$ | Replaces the stored Developer address. | *Root Owner* |
| `setSoftwareName` | *name* | Sets the variable `software name`. | *Developer* |
| `registerBHSContract` | $\text{BHS}_{addr}$ | Registers a BHS Contract by storing the submitted $\text{BHS}_{addr}$. The SDP ID used as storage key is obtained from the BHS Contract. | *Developer* |
| `deregisterBHSContract` | SDP ID | Deregisters the BHS Contract stored under *SDP ID*. | *Developer* |
| `updateSDP_ID` | SDP ID (old), SDP ID (new) | Changes the storage key of a BHS Contract. | *corr. BHS Contract* |
| **Calls** | | | |
| `getBHSContract` | SDP ID | Returns the $\text{BHS}_{addr}$ stored under *SDP ID*. Returns 0, if no such entry exists. | *All* |

**Table 2: Interface of BHS Contract. Simple calls to retrieve the values of attributes are omitted.**

| Name | Arguments | Functionality | Allowed Caller |
|---|---|---|---|
| **Transactions** | | | |
| `changeRootOwner` | *Root Owner* (new) | Replaces the stored Root Owner address. | *Root Owner* (current) |
| `setMaintainer` | $\text{IDM}_{addr}$ | Replaces the stored Maintainer address. | *Root Owner* |
| `registerSoftwareContract` | - | Completes the binding to an SW Contract. | *corr. SW Contract* |
| `setSDP_ID` | SDP ID | Changes the stored SDP ID and calls the stored SW Contract to update its corresponding storage key. | *Maintainer* |
| `publishHash` | HashID, Hash | Stores *Hash* under *HashID*. | *Maintainer* |
| `revokeHash` | HashID | Revokes the Hash of *HashID*. | *Maintainer* |
| **Calls** | | | |
| `getBinaryStatement` | HashID | Returns the *binary statement* consisting of (Hash, Counter) stored under *HashID*. Returns 0, if no entry exists. | *All* |

Binary hashes and revocation statements are published by a Software Maintainer via this contract. One particular implementation detail that deserves explicit mention is the inclusion of a publish counter. In Ethereum, variables are initialised to 0 and if a requested variable does not exist, 0 is also returned. In order to distinguish between a revoked hash, which we set to 0, and a hash that was never issued, we increment the publish counter and attach its value to every binary hash statement during publication. Consequently, a statement with a zeroed hash and a non-zero counter is clearly identifiable as revoked. Additionally, the counter serves as a relative time stamp and can provide limited information on the freshness of individual hashes.

The interface of BHS Contracts can be found in Table 2. Of particular note is the function `registerSoftwareContract` which is restricted to an SW Contract specified by the Maintainer and is called during the aforementioned legitimisation. The most prominent functions in general are `publishHash`, `revokeHash`, and the call `getBinaryStatement` which are used for publication, revocation, and validation of hashes, respectively.

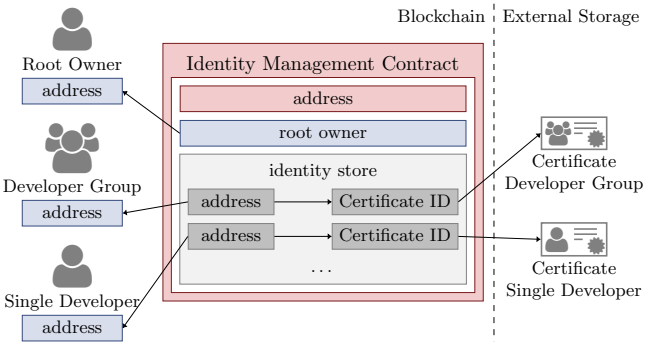


**Figure 6: Inner structure of an IDM Contract and relation to other objects.**

## 4.3 Identity Management Contract

Lastly, we employ Identity Management (IDM) Contracts (Figure 6) to allow both Software Developers and Maintainers to manage their own set of public keys for Blockchain transactions and optionally store references to corresponding certificates. To accomplish this,

**Table 3: Interface of IDM Contract. Simple calls to retrieve the values of attributes are omitted.**

| Name | Arguments | Functionality | Allowed Caller |
|---|---|---|---|
| **Transactions** | | | |
| `changeRootOwner` | *Root Owner* (new), Cert. ID | Replaces the stored Root Owner address. | *Root Owner* (current) |
| `resetIdentitySet` | - | Resets `identity store` by removing all stored identities. | *Root Owner* |
| `addIdentity` | Identity | Adds *identity* to `identity store`. | *prev. est. identity in identity store* |
| `removeIdentity` | Identity | Removes *identity* from `identity store`. | *prev. est. identity in identity store* |
| `changeCertificateID` | Cert. ID (new) | Replace the Cert. ID of the calling identity. | *prev. est. identity in identity store* |
| **Calls** | | | |
| `checkIdentity` | Identity | Returns `true` if *identity* is contained in `identity store`, returns `false` otherwise. | *All* |
| `getIdentityCertID` | Identity | Returns the stored Cert. ID of *identity*. | *All* |

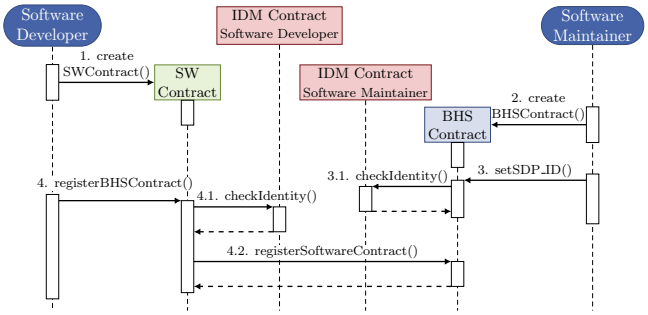


**Figure 7: Sequence diagram showing the initial setup of Smart Contracts. Both IDM Contracts were previously established by their respective owners and provided as references during the creation of the SW and BHS Contract.**

each IDM Contract contains an identity store of authorised public keys. Before any restricted action is executed on BHS or SW Contracts, the inclusion of the actors public key in the respective IDM Contract is checked through the call `checkIdentity`. Both Software Developers and Maintainers create Identity Management Contracts and store references to them in their respective other contracts. It is important to point out that identity management on Blockchains is an area of research in itself with many different approaches [1, 3, 6] that could serve as replacements for IDM Contracts.

The interface of IDM Contracts is shown in Table 3. In addition to the management functions `addIdentity` and `removeIdentity`, the most relevant function is the previously mentioned call `checkIdentity` which is used whenever the authorisation of a particular actor is determined.

## 4.4 Process Flow

In this section, we will examine the primary functions of the presented concept in more detail, starting with the setup.

*Setup.* An overview of the setup is depicted in Figure 7. A Software Developer begins by creating an Identity Management Contract and adding her public keys to it. Next, she creates a Software Contract, thereby establishing a permanent Root Software Identity, and adds a reference to her IDM Contract in order to grant herself the necessary privileges. She can then transfer the Root Owner secret keys to offline storage for safekeeping. Meanwhile, a Software Maintainer creates a BHS Contract to establish an Intermediary SI with a similarly referenced IDM Contract of his own creation. This BHS Contract contains both a reference to the SW Contract he intends to be bound to as well as an identifier of the Software Distribution Platform that he intends to upload binaries to. Once these four contracts have been created, the last step is for the Software Developer to authorise the Maintainer to create and distribute binaries by adding a reference to the BHS Contract, thereby forming a verifiable binding between the Root and Intermediary SI.

*Publication.* To publish a binary, a Software Maintainer chooses or generates a unique and unused identifier and adds it to the binary metadata along with the address of the corresponding BHS Contract. Then, he computes a hash of the binary and metadata, and sends it as part of a signed transaction to the BHS Contract via the Ethereum peer-to-peer network. Before adding the hash, the Ethereum network executes the contract code which ensures the availability of the chosen identifier and the authorisation of the transaction author. Once the hash has been added to the BHS Contract, the Maintainer can publish the binary to the corresponding SDP. The hash identifier constitutes a verifiable binding between the Intermediary Software Identity, represented by the BHS Contract, and the binary.

*Validation.* To validate a downloaded binary, the User first needs access to a validated copy of the current Ethereum state, or at least of the contracts corresponding to his binaries. He either maintains and updates this state himself by passively joining the peer-to-peer network or he must obtain it from peers. The User then inspects the metadata included in the binary to find both the identifier and the BHS Contract address which allows him to obtain a reference to an SW Contract in addition to a hash which he compares to the one he calculates himself, thereby verifying the binding between binary and Intermediary SI. He then validates the binding between the Intermediary and Root SI by checking for a corresponding reference from the SW Contract back to the BHS Contract. If this is

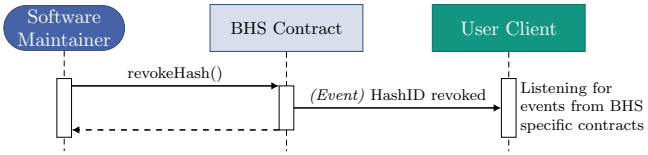


**Figure 8: Sequence diagram showing the revocation of a HashID.**

the first time the User validates a binary of this particular software, he also stores the address of the SW Contract. Otherwise, the User checks if the found SW Contract is the same as his previously stored one before accepting the binary as valid. This last step is worth emphasising. Since an attacker can deploy his own contracts, we need to establish a 'trust on first use' relation between Users and SW Contracts to protect against counterfeit contracts. While the Software Developer in control of the SW Contract may change, its contract address remains constant and serves as a trust anchor for future validations.

*Revocation.* Revoking a binary begins with the responsible Software Maintainer issuing a transaction containing the respective identifier to the Ethereum network (cf. Figure 8). It is worth noting that we consider the way in which the Software Maintainer learns of a reason for this revocation out of scope for this paper. In addition to the necessary HashID, additional information like a URL could be attached to the transaction, although we did not include this feature in our implementation. A User passively listening to the peer-to-peer network can check both the authenticity of the transaction and the authorisation of its sender to immediately take action in case it affects him. Similarly, the network peers execute the corresponding contract code which zeroes the hash after performing checks similar to those during publication. Clients that were offline during the revocation should catch up to the current Ethereum state as soon as possible to learn of any revocations to their used binaries in order to react appropriately. Note that a Developer may also collectively revoke all binaries of a particular Maintainer by deregistering the corresponding BHS Contract from his SW Contract, thereby lifting the binding between the Root and Intermediary Software Identity.

### 4.5 Alternative Approach

Creating one BHS Contract instance per software and SDP has the advantage of keeping each binary hash statement, and thus the cost of their publication, as small as possible in exchange for more frequent one-time deployment costs for each contract instance. An alternative approach would aim to reduce the number of contract instances in exchange for larger hash statements. This trade-off occurs because each hash must be associated with a particular software and Maintainer. In the concept described above, this relation is established through the interlinking of contract instances, whereas the alternative approach would have to store the necessary references alongside each hash. Depending on the implementation of this alternative approach, it could also incur additional storage costs to retain access control information about which Maintainer is authorised to publish binaries of which software. Regardless, we can only provide a glimpse into the available design space with our implementation and a careful examination of the expected usage is necessary to strike the right balance between all available trade-offs.

## 5 EVALUATION AND DISCUSSION

It is interesting to see how each component of the Ethereum ecosystem serves multiple purposes in achieving the goal of a decentralised solution for binary integrity protection. In addition to being a highly available, public, tamper-evident database storing integrity protecting information, the Blockchain also functions as an immutable, auditable and always invoked access control mechanism via Smart Contracts. Similarly, the high global availability and fast communication speed of the underlying peer-to-peer network is not only necessary for its primary function of maintaining and extending the Blockchain but also compliments our concept as a reliable and machine-readable communication channel for time-sensitive messages as in the case of revocations.

### 5.1 Design Choices

We chose to base our implementation on Ethereum due to its prevalence amongst Smart Contract-capable Blockchains. This in turn allows us to determine realistic costs for an actual deployment. It is worth noting that, while we use certain features of Ethereum (like Events), the presented concept should be transferable to comparable platforms with minor changes.

In a similar way, we opted for a permissionless, and therefore 'trustless', environment since it places more constraints on the presented concept. Consequently, transferring the concept to a permissioned environment with semi-trusted participants is possible by relaxing or removing certain aspects and safeguards.

### 5.2 Security

Looking closer at possible attacks, we can demonstrate how the presented concept based on Ethereum helps to prevent Users from executing unsafe binaries. To briefly restate attacker capabilities defined in Section 3.2, he can compromise authentic binaries and publish them, deploy his own Smart Contracts in order to pass off manipulated binaries as authentic, and he can attempt to influence a Users view of the Blockchain.

Simply altering a binary fails during the verification by the User. Establishing their own set of Smart Contracts on the Ethereum Blockchain, adding the hashes of compromised binaries to them, and altering the metadata of manipulated binaries accordingly would succeed if it was the first time a User downloads a binary of this particular software. If the User had previously stored the address of the trusted SW Contract, a mismatch would be detected during validation and the binary would be rejected. Seeing how these fraudulent Smart Contracts would have to be in plain sight of the entire ecosystem, diligent observers might raise awareness before any damage could be done. Additionally, this approach puts an economic burden on the attacker for the deployment of his contracts, as will become clear in the following Section. It is important to note that a supply-chain attack would succeed, even in the case of a previously established trust relation between Users and SW Contracts. However, the proposed concept was not intended to prevent this kind of attack. Instead, the concept enabled the

responsible Software Maintainer to quickly revoke the binary in question in order to both limit further damage and to help affected Users recover.

Methods for manipulating a User's view of the Blockchain generally fall into one of two categories. In Split-View attacks, the attacker attempts to present the User with a convincing alternate Blockchain, and in Freeze attacks, the attacker tries to prevent the User from receiving updates. The Ethereum ecosystem includes countermeasures for both of these attacks. On average, a new block is added to the Ethereum Blockchain every 15 seconds. If an attacker manages to prevent any new blocks from reaching a User in order to execute a Freeze attack, it should raise suspicion within minutes. Additionally, Ethereum's highly distributed nature makes this kind of attack very difficult to execute in practice. This also applies when attempting to block a Maintainer from issuing a revocation. An attacker trying to avoid suspicion by producing and presenting an alternate Blockchain to the User in order to execute a Split-View attack would have to match the mining power of the Ethereum peer-to-peer network, which would require a prohibitively large economic investment by the attacker.

## 5.3 Costs and Scalability

As explained in Section 2.1, deployment and execution of Smart Contracts on Ethereum incur transaction fees that must be payed to the miners who include the corresponding transaction in a new block. The cost of deploying a Smart Contract mainly depends on the storage space for the EVM bytecode, whereas the cost of execution depends on the individual operations that the EVM performs. In order to investigate the economic impact of the presented concept, we implemented the necessary Smart Contracts in Solidity (version 0.4.18) and compiled them to EVM bytecode using the Remix IDE with version 0.4.24 of its Solidity compiler. Using the Ganache test framework, we deployed and executed the contracts on a locally hosted Ethereum Blockchain to accurately determine the costs of various operations. While Ether is the eponymous cryptocurrency of Ethereum, transaction fees are calculated in *Gas* and the originator of any transaction sets the exchange rate of Ether to Gas as well as a maximum amount of Gas to be expended. Using the daily averages from the 1st of February, 2019[3], we base the cost estimation on an exchange rate of USD$ 107.1 per Ether and a Gas price of ETH $13.23 \times 10^{-9}$. In order to fully comprehend the reliability of the following cost analysis, it is of vital importance to realise that these exchange rates are influenced by both market forces as well as network utilisation. A little more than a year ago, on the 13th of January, 2018, the price of Ether spiked at USD$ 1 385.02 while the Gas price was at ETH $63.72 \times 10^{-9}$ on that date.

In Table 4, we give a concise overview of the monetary costs that the primary functions of our concept invoke. It is important to note how often these functions are expected to be executed. SW Contracts are deployed only once per software project and BHS Contracts are deployed once for each Software Distribution Platform on which binaries of a particular software are published. Meanwhile, IDM Contracts are only deployed once per authorised party, unless preexisting contracts can be used, and they can be reused for more than one SW or BHS Contract.

[3]https://etherscan.io

**Table 4: Cost of operations (exchange rates USD to Ether and Ether to Gas as of 1st of Februrary, 2019)**

| Operation | Cost (USD) |
|---|---|
| Deploying SW Contract | $ 2.365 |
| Deploying BHS Contract | $ 2.314 |
| Deploying IDM Contract | $ 1.612 |
| Publishing a Hash | $ 0.112 |
| Revoking a Hash | $ 0.051 |

Let us take for example an individual assuming both the roles of Maintainer and Developer of three distinct software projects, each with monthly updates released to two different SDPs. The one-time setup costs in this scenario would consist of deploying one IDM Contract, three SW Contracts, and six BHS Contracts, so $ 13.335 in total. The annual costs for publishing six hashes per month would be $ 8.064. Considering current fees for developer accounts at Apple[4] ($ 99 per year), Google[5] (one-time $ 25), and Microsoft[6] (one-time $ 19), the costs of our scheme are quite reasonable.

Next to the monetary aspects, there are also functional considerations when it comes to the scalability of the presented concept. Since reading from the Blockchain can be done on a local copy by each User, this should not present a limiting factor. However, establishing contracts and executing transactions to publish and revoke hashes raises more concerns in this regard. Ethereum imposes an upper limit on the amount of Gas that can be expended within any given block. Generally speaking, this limits our concept to about 100 hash publications per block in a best case scenario. Depending on current network utilisation and transaction fees, the time from submission of a transaction to its inclusion in the Blockchain can range from minutes to hours. However, Users can receive transactions containing newly published or revoked hashes through the peer-to-peer network and validate them locally even before miners include them in the Blockchain, thereby mitigating delays due to network congestion. In this way, the network serves as an immediate, global notification service while the permanent record on the Blockchain allows temporarily offline Users to catch up later. Furthermore, the use case at hand is highly asymmetrical with every hash being written once but read significantly more often, which also aligns perfectly with the cost and performance characteristics of Ethereum. With these limitations in mind, an incremental integration of the presented concept into existing software distribution processes is not only possible, but preferable.

One point that has gone largely unexplored until now is the overhead on the side of Users. A full history of the Ethereum Blockchain and a copy of the entire current state takes up several gigabytes of disk space. This is somewhat excessive, considering that Users within the context of the proposed system would only care about a limited portion of the Ethererum state, namely the part that includes contracts related to their installed software. In theory, Users could obtain and maintain this partial state from the Ethereum

[4]https://developer.apple.com/programs/how-it-works/
[5]https://play.google.com/apps/publish/signup/
[6]https://docs.microsoft.com/en-us/windows/uwp/publish/account-types-locations-and-fees

peer-to-peer network in order to perform binary validation with a minimal storage footprint. Alternatively, they could employ the light client protocol to delegate this task to a number of full nodes. In any case, Users would not be required to perform mining or any other computationally intensive task in order to take part in the proposed system.

### 5.4 Comparison to Related Work

Looking back at the two related works introduced in section 2, we can clearly distinguish the presented concept and point out trade-offs. While Contour [2] is superior in terms of transaction costs and storage space requirements, it sacrifices transparency and any way to revoke previously published hashes. Due to the Merkle tree construction, it is possible for the SDP owner publishing the root hash to the Bitcoin Blockchain to hide hashes within the tree and present valid proofs of inclusion to selected users. An outside observer would have to reconstruct the entire Merkle tree from individual proofs of inclusion and compare it to the available binaries to discover such a deception. Furthermore, Contour lacks any way for a user to ascertain who is even authorised to publish binaries of a given software or who generated the stored hashes in the first place.

Chainiac [9] goes beyond securing binary integrity and also includes precautions to ensure code-to-binary correspondence and a transparent update log. Similar to the presented concept, Chainiac also supports the rotation of public keys to combat or recover from compromise but it remains unclear whether or not individual binaries can also be revoked. Although great measures are taken to prevent any external cause for such revocation, security critical bugs, for instance, are still a possibility and due to the transient nature of Skipchain cothorities it might not be possible to reliably and persistently alter the release log to reflect revocations. This comprehensive set of features comes at the cost of significant implementation costs and the bootstrapping of an entirely new Blockchain system, whereas the presented concept is easily deployed on the established and growing Ethererum ecosystem.

In summary, the proposed concept is positioned between Contour and Chainiac both in terms of features and complexity with the distinctive feature of enabling timely and precise revocation of previously published binaries.

## 6 CONCLUSION

In this paper, we presented a concept based on Smart Contracts to provide integrity protection for software binaries that supports timely, pinpoint accurate, and machine readable revocations to protect users from executing untrustworthy binaries or give them actionable notice in case they already executed said binaries. We accomplish this without introducing an additional trusted party. The costs of operations incurred by the proposed concept was derived based on Ether to USD conversion rates and shown to be comparable to registration fees of centralised software distribution platforms. Additionally, we gain tamper-evident, auditable, and always invoked access control in the form of Smart Contracts which are enforced through the consensus mechanism of Ethereum.

The design, implementation, and evaluation of a user client application to interact with the Ethereum Blockchain in order to validate binaries is ongoing work in addition to general improvements and refinements. Furthermore, the presented concept can be used as a blueprint and applied to similar use cases. While the concept at hand decentralises the storage of integrity protecting information, its creation and revocation is still in the hands of singular authorities. Future work could look at decentralising these aspects as well.

To conclude, we see Smart Contract enabled Blockchains, most notably Ethereum, as a very well suited framework for providing binary integrity protection and possibly for other information dissemination problems that require verifiable access control and provenance as well.

## ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research within the framework of the project KASTEL_ISE in the Competence Centre for Applied Security Technology (KASTEL).

We would like to thank the anonymous reviewers for their feedback and comments.

## REFERENCES

[1] Mustafa Al-Bassam. 2017. SCPKI: A Smart Contract-based PKI and Identity System. In *BCC '17 Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. ACM, Abu Dhabi (United Arab Emirates), 35–40. https://doi.org/10.1145/3055518.3055530

[2] Mustafa Al-Bassam and Sarah Meiklejohn. 2018. Contour: A Practical System for Binary Transparency. In *DPM 2018, CBT 2018 - Data Privacy Management, Cryptocurrencies and Blockchain Technology*, Vol. LNCS 10436. Springer, Barcelona (Spain), 373–389. https://doi.org/10.1007/978-3-030-00305-0_8

[3] Sarah Azouvi, Mustafa Al-Bassam, and Sarah Meiklejohn. 2017. Who Am I? Secure Identity Registration on Distributed Ledgers. In *DPM 2017, CBT 2017 - Data Privacy Management, Cryptocurrencies and Blockchain Technology*, Joaquin Garcia-Alfaro, Guillermo Nacarro-Arribas, Hannes Hartenstein, and Jordi Herrera-Joancomartí (Eds.), Vol. LNCS 10436. Springer, Oslo (Norway), 373–389. https://doi.org/10.1007/978-3-319-67816-0_21

[4] V. Buterin and Ethereum Community. 2013. A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper

[5] Robert Duncan. 2013. How certificate revocation (doesn't) work in practice. https://news.netcraft.com/archives/2013/05/13/how-certificate-revocation-doesnt-work-in-practice.html

[6] Sebastian Friebe, Martina Zitterbart, and Ingo Sobik. 2018. DecentID: Decentralized and Privacy-Preserving Identity Storage System Using Smart Contracts. (2018).

[7] Ralph C Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 369–378.

[8] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. , 9 pages. https://bitcoin.org/bitcoin.pdf

[9] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX, 1271–1287.

[10] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (Sept. 1997).

[11] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).