EthIKS: Using Ethereum to audit a CONIKS key transparency log

Joseph Bonneau

Stanford University & Electronic Frontier Foundation

Abstract. CONIKS is a proposed key transparency system which enables a centralized service provider to maintain an auditable yet privacy-preserving directory of users' public keys. In the original CONIKS design, users must monitor that their data is correctly included in every published snapshot of the directory, necessitating either slow updates or trust in an unspecified third-party to audit that the data structure has stayed consistent. We demonstrate that the data structures for CONIKS are very similar to those used in Ethereum, a consensus computation platform with a Turing-complete programming environment. We can take advantage of this to embed the core CONIKS data structures into an Ethereum contract with only minor modifications. Users may then trust the Ethereum network to audit the data structure for consistency and non-equivocation. Users who do not trust (or are unaware of) Ethereum can self-audit the CONIKS data structure as before. We have implemented a prototype contract for our hybrid EthIKS scheme, demonstrating that it adds only modest bandwidth overhead to CONIKS proofs and costs hundredths of pennies per key update in fees at today's rates.

1 Introduction

Distribution and verification of public keys for end-to-end encrypted communication remains a challenging problem. In terms of deployment, the most successful model has been centralized services which serve as trusted public key directories [10], such as those used by iMessage and WhatsApp. This model is also employed by security-focused messaging applications such as Signal (TextSecure), Silent Circle, Telegram or Threema. These apps additionally allow users to verify public keys manually, although experience suggests few actually do so.

These services might try to launch a man-in-the-middle attack by serving keys maliciously. That is, instead of serving Alice's true public key PK_A to Bob, the server might serve a public key PK_S for which it knows the private key. Such attacks are facilitated in centralized applications since the server typically routes and/or stores all communication for efficiency, making it is straightforward to decrypt traffic if the keys are known. Many applications also enable users to register multiple public keys to support multiple devices, making it easy to add an "interception key" which simply looks like an extra device.

An essential requirement of this attack is that the key directory interacts inconsistently between Alice and Bob. If Alice queries her own public key, it

should respond with the correct result or else Alice's device may automatically detect the attack (since it knows which public keys it has uploaded). A key server with global consistency would therefore be a significant security upgrade: as long as Alice verifies that her own entry in the directory is correct, she can be sure that she is not being attacked. Furthermore, if Bob trusts that Alice is regularly monitoring her own entry, Bob can accept whatever public keys the directory returns for Alice and trust that she will detect if an attack has taken place. While this setup will only detect attacks (without preventing them), it is far more lightweight for users than manually verifying all public keys out-of-band.

CONIKS [7] (CONsistent Identity Key Service) is a concrete proposal for a key server with consistency while also protecting users' privacy. This approach is currently being adapted by Google and Yahoo! for use in their prototype end-to-end encrypted email systems. The key data structure in CONIKS is a signed hash chain of roots of Merkle prefix trees.

Ethereum [11] is a "secure decentralized transaction ledger." Inspired by Bitcoin [9], Ethereum adds support for long-lived, stateful "contracts" with a Turing-complete scripting language. Under the hood, Ethereum uses data structures similar to CONIKS, including a blockchain with snapshots of the entire Ethereum system using Merkle Patricia trees to store the state of each contract.

These similarities are not coincidental. While the two systems were designed for very different purposes, both require a globally consistent data structure supporting efficient updates and proofs of inclusion. In this paper we show that, with minor modifications, a CONIKS directory can be "wrapped" inside an Ethereum contract in a hybrid scheme we call EthIKS. This allows it to piggyback on Ethereum's consensus protocol to prevent equivocation, potentially obviating the need for a separate gossip protocol to ensure consistency. It also enables increased efficiency for clients willing to trust the Ethereum network.

We have implemented a prototype Ethereum contract to measure the cost of EthIKS both in terms of transaction fees paid to the Ethereum network ("gas") and bandwidth overhead compared to the original CONIKS design.

2 CONIKS overview

We provide a brief overview of CONIKS here [7]. The key data structure in CONIKS is a chain of directory snapshots, or *signed tree roots* (STRs). Each STR commits to the entire directory, which is a binary Merkle prefix tree containing the current mapping from users to public keys.

Merkle prefix trees The tree in CONIKS maps arbitrary indices[1] to data. It is a radix tree; each branch of the tree represents either a "0" or "1" in the binary representation of an index. Each leaf of the tree stores data mapped to the index represented by its complete path from the root. To reduce the length of paths in the tree, subtrees with only one non-null leaf are collapsed into a single leaf

[1] The term "key" is avoided to prevent confusion with cryptographic keys.

marked with the unique suffix of this single non-null index. The data structure is authenticated in that each non-leaf node includes the hash of its children. The root of the tree thus uniquely commits to the entire data structure, assuming the hash used is collision-resistant [8].

Private bindings To ensure privacy, CONIKS generates the index for each user's data using a verifiable unpredictable function (VUF). The CONIKS provider generates a VUF private key which can be used to deterministically derive the index for any username and provide a publicly-verifiable proof that this index was generated correctly. Furthermore, each leaf in the CONIKS tree stores a commitment to a user's data rather than the data itself. Thus, to verify a (username, data) binding in the CONIKS tree, one must verify both that that the index produced by the VUF for that username is present in the tree and that the commitment at that leaf commits to the claimed data. Without the VUF proofs or commitment randomness, the CONIKS tree reveals no information about any usernames or their data beyond the number of users in the tree.[2]

Key binding proofs To communicate in CONIKS, Alice requests Bob's key binding from the CONIKS provider as of the latest STR. The provider responds with Bob's key data, a Merkle proof of inclusion in the STR's tree root, the VUF proof of Bob's index and the randomness to open the commitment to Bob's data.

Non-equivocation To assure that all users see a consistent version of the CONIKS tree, the root is included in a chained sequence of STRs. Each STR commits to the hash of the previous version of the tree (and hence the entire history of the directory) as well as a timestamp and other metadata, and is signed by the CONIKS provider. While the CONIKS provider is able to sign two inconsistent versions of the tree, if they are ever discovered this will provide non-repudiable proof that the provider is malicious. To discourage such equivocation, the original CONIKS proposal assumes that users will participate in a gossip protocol to share STRs they have observed. It also suggests that STRs might be embedded in an external append-only log such as the Bitcoin blockchain.

Key updates and revocation By default, the CONIKS provider can change a user's key binding at any time. This enables recovery from lost or stolen keys by traditional backup authentication means such as password reset questions or telephone helplines. Optionally, CONIKS users may request that their leaf be marked with a *strict* flag meaning that updates must be signed by a designated user-controlled key. This option enables preventive (rather than purely detective) defense against unauthorized key changes, at the price of burning the username forever if the update key is lost. There is no special notion of revocation of CONIKS: the old key is simply replaced in the next version of the tree.

[2] The number of valid users in the system can be obscured by adding dummy users at random indices with random data, which will be indistinguishable from real users.

Auditing and monitoring Each CONIKS user *audits* the provider for consistency, checking that each STR forms a chain and potentially checking for equivocation with third parties (i.e. gossip). Auditing can also be done by any third-party. Each user also *monitors* their own entry in the tree for correctness based on the key changes they have actually requested from the server. If an unexpected key change occurs, the user's software should show a warning message.[3]

Efficiency considerations In the process of auditing and monitoring, each user must download every STR from the server and check that their binding is correctly included. While these checks are all logarithmic in the number of users, if STRs are issued frequently users must download and verify a large number of signatures. However, if STRs are issued slowly, the time to add a new key binding (or equivalently, revoke an old one) will be long. The original CONIKS proposal suggested STRs being issued on the order of hours, with a secondary system of auditable "promises" to include data in the next STR to enable faster enrollment (similar to signed certificate timestamps in Certificate Transparency [4]).

In development at Google and Yahoo!, promises were scrapped in favor of faster STR updating times. To mitigate the cost of verifying that a user's binding has stayed consistent in n consecutive STRs, an update count is added to each leaf, enabling users to simply verify that their update count was not incremented at in the most recent STR. However, this assumes the existence of third party auditors to verify that update counts are incremented if and only if the committed data is actually changed.

Our goal in wrapping CONIKS in an Ethereum contract is to maintain the advantages of frequent STRs, while relying on the Ethereum network to audit that update counts are incremented correctly. We also use Ethereum to gain confidence in non-equivocation.

3 Ethereum overview

While it is often described as being "like Bitcoin with a Turing-complete scripting language," Ethereum [11] is perhaps more accurately described as a *consensus computer*. Unlike Bitcoin, in which each block contains a set of transactions updating an implicit global state, each block in the Ethereum blockchain explicitly commits to the complete state of the system which includes both user accounts and *contracts*, which represent a running process in the system with code, memory state and a monetary balance. Each contract's code describes an API which users of the system can call to cause the contract to execute some code which may update its state and/or transmit money to other contracts or users. An API call is called a *transaction*. Transactions must be signed by a specific sender and may contain a payment and an arbitrary amount of data.

[3] Note that in CONIKS, warning messages are only intended when the user's own key has changed unexpectedly at the server. If their peer's keys change, this is ignored as it is assumed the peer will monitor this change themselves.

A simple example is a game of chess between strangers with a binding monetary bet. A contract representing a chess game can be sent to the network. Its code should initialize the contract state to represent an empty board and no players. Two players may then join by sending a message to the contract along with a deposit equal to the betting stake. While the game is underway, the deposits will be owned by the contract itself. Each player will then submit moves in turn, with the contract updating the board after each move and rejecting any invalid moves. When one player wins the game, the contract would then send its entire value to the winner and close.[4]

Programming Ethereum contracts correctly has already proven quite subtle [2], requiring defensive programming and extensive sanity checks to ensure no API calls can corrupt the contract state. For example, in the chess game, the contract must implement a timeout rule where players lose by default if they don't submit a move within a required time, to avoid simply stalling a lost game forever (sometimes called a "rage quit").

Contract fees The state of every contract in the system (as well as each user account) must be tracked by every miner. Every miner must also validate all transactions in every block to see that they execute each contract's code correctly and update the global state accordingly. This presents an obvious denial-of-service avenue as contracts may contain infinite loops, allocate an arbitrary amount of storage, or perform other resource-intensive computations. Thus, every instruction executed requires a fee, referred to as *gas*. Gas is the same currency used for sending value between users and/or contracts in the system; it is simply called gas when it is being used to pay for executing a transaction.

Simple instructions (e.g. addition) cost 1 gas whereas more complex instructions may cost significantly more (e.g. computing a SHA-3 hash costs 20 gas). Writing to storage is particularly costly, at 100 gas per 256-bit word. Any transaction must send sufficient gas to pay for all of the instructions executes. If a contract runs out of gas while executing a transaction, execution halts with all changes to the state undone and the gas being kept by the miners. Thus it is critical both to write programs which are efficient in their gas costs and to ensure transactions contain sufficient gas to pay for the instructions they execute.

Storage model Each block in the Ethereum blockchain contains a "Merkle Patricia tree" with the state of each contract and user account stored in the leaves. Each contract or user account is represented by a unique 160-bit address (either the hash of the user's public key or a hash of the contract's code plus a nonce). Unlike the prefix tree used in CONIKS, in Ethereum the Patricia tree is 16-ary (hexary), although the suffix-compression is similarly applied.

Each leaf contains a hash of that addresses state, including its current balance within the system. For addresses representing contracts (as opposed to simply user accounts) the state also includes the root of a Merkle Patricia tree representing that contract's persistent storage. The storage model is very simple: each

[4] In Ethereum parlance, the contract closes by calling a special `SUICIDE` opcode which enables the network to permanently delete its storage.

contract has a memory space of 2^{256} 256-bit words, each representing a 256-bit address. The contract's storage is thus a function $\{0,1\}^{256} \to \{0,1\}^{256}$. Leaves are inserted into the tree for any address with a non-zero word stored; addresses which are not in the storage tree are interpreted to have a value of 0.

This storage model makes implementing hash tables extremely simple in Ethereum: the value v associated to a key k is simply stored at the memory address $H(k)$, with the storage tree handling this efficiently under the hood. Most high-level Ethereum languages (including Solidity) expose this $\{0,1\}^{256} \to \{0,1\}^{256}$ map as a built-in type.

4 EthIKS

Given the contract execution model of Ethereum, we can implement a close variant of CONIKS. The goal will be to ensure EthIKS is as secure as CONIKS for clients which ignore Ethereum completely and interact directly with the EthIKS log, while Ethereum-aware clients will gain greater trust and efficiency.

Core data structure EthIKS implements the core data structure of CONIKS, the tree mapping user indices to user data, in the persistent storage of a single Ethereum contract. This contract allows the service provider to update the tree by sending messages from a designated address. The service provider can send the contract an index i and a commitment c, which will then be stored (or updated) in the tree by simply writing the value c to memory address i. These values will be the VUF-derived private index for a given user and the commitment to the user's public-key data. A side-effect of this design is that the Ethereum blockchain will contain a record of each update to the tree, something that is not normally published in CONIKS.

Key bindings As in plain CONIKS, a key binding will include the VUF proof that an index i corresponds to a username u, as well as the randomness required to open the commitment c to the user's data. Because these values are now stored in an Ethereum contract rather than a separate CONIKS-specific prefix tree, the key binding must include the proof that the address c has the value i in the contract's persistent storage. This is simply a proof-of-inclusion for the contract's storage tree plus a proof-of-inclusion that this storage tree is currently mapped to the contract as of the most recent block in the Ethereum block chain. Notice that each Ethereum block is effectively a signed tree root (STR) in EthIKS.

Backwards-compatible proofs To maintain the normal CONIKS interface for clients which wish to ignore Ethereum, the provider still publishes signed tree roots after every update to the tree. In EthIKS the tree root to be signed is the root of the Merkle Patricia tree representing its EthIKS contract's storage after each update. These tree roots are implicitly "signed" by the Ethereum network through their inclusion in a block. The provider additionally signs the tree root, combined with a pointer to the previous version of the tree and publishes

this chain of signatures separately. For non-Ethereum-aware clients, this chain of signed roots functions exactly as in plain CONIKS.

Update frequency In CONIKS, the tree is updated (by publishing a new STR) at a provider-chosen frequency. In EthIKS, the tree can be updated in every Ethereum block. The Ethereum block frequency, targeted currently at one block per 12 seconds, is a lower-bound on the epoch length. The provider may choose to sign the tree less frequently than once per block to reduce the length of its owned chain of signed tree roots. Legacy clients would only see updated versions at this slower rate. However, Ethereum-aware clients would see new updates rapidly (and may rapidly update their own entries).

Update counts The EthIKS contract maintains an *update counter* for each index in the tree. Any update to this index's data must increments the counter; the contract's code which allows no other API for updating the tree. The benefit of this counter is that Ethereum-aware clients can be sure that their data in the tree has not changed if their counter has not changed, allowing them to skip monitoring every version of the tree and simply check the counter value in the most recent version of the tree.

User-controlled addresses EthIKS supports a comparable feature to CONIK's strict mode: each leaf in the tree has an associated owner (by default the service provider) which is the only address allowed to send updates to this leaf in the tree. Updates may include changing the owner; the provider must do this initially to create a new leaf and then transfer control to its owner if requested. Security-conscious users may request the service provider change their leaf's owner address to a public key of their choosing.

Unlike in CONIKS, users who opt for control of their own leaf can then update it directly by communicating with the Ethereum contract themselves, they no longer need to route updates through the service provider. However, if they update their data and don't send the commitment randomness to the provider, the provider can no longer answer queries about this user's public key.

Revoked usernames EthIKS also retains CONIKS' ability to permanently remove a user's data by replacing it with a special *tombstone* value. In EthIKS, tombstoned users simply have their owner set to a dummy address for which the private key clearly does not exist (e.g. the public key whose hash equals zero). Note that this is not the same as setting a user's data to be zero; this removes their data from the tree but enables this username to be later reincarnated.

5 Implementation and costs

We have implemented a prototype of EthIKS by modifying the prototype CONIKS implementation [7] and writing an Ethereum contract to handle the core tree updates. The EthIKS contract is contains fewer than 100 SLOC in Solidity,

operation	gas cost			
	gas	ether	mBTC	US dollar
create tree	367535	3.675	1.838	0.0036
insert new user	42042	0.420	0.210	0.0004
update mapping	12042	0.120	0.060	0.0001
delete user data	12042	0.120	0.060	0.0001
change ownership	17382	0.174	0.087	0.0002
tombstone user	17382	0.174	0.087	0.0002

Table 1: Transaction fees (gas costs) for different update types in EthIKS, along with the current price in ether (the base currency of Ethereum), millibitcoin (mBTC), and US dollars as of January 2016 exchange rates and the default gas price of 1 gas = $5 \cdot 10^{-8}$ ether. Ethereum transactions also incur an overhead cost of 21,000 gas, but this can be amortized by batching multiple updates in a single transaction so we ignore it here.

Ethereum's most popular high-level language for writing smart contracts. The contract exposes only a single API call (besides the constructor), `updateMappings` which takes a list of indices, data values, and (possibly null) addresses. Each index is updated to the new data value and its counter incremented after checking that the owner of this index is the party sending the message.

Gas costs The transaction costs (in gas) of updating the EthIKS tree through the EthIKS contract are listed in Table 1. These are based on our Solidity implementation, a hand-coded byte code might achieve better efficiency. The main cost in all operations is writing to permanent storage; the current implementation of Solidity invokes several writes at 100 gas each for every update to the tree. Still, the contract costs on the order of hundredths of pennies per update to the tree. At current rates, these costs would be significant for a large provider, which might be required to handle millions of key updates per day (costing tens of thousands of dollars in gas). However, we note that the future value of gas (and ether) is very difficult to project. The current Ethereum network (Frontier) would not handle a provider with billions of users due to limits on the size and number of transactions per block, but these limits are expected to be increased in the future as Ethereum scalability improves.

Bandwidth costs EthIKS re-uses the same construction for the VUF and hash commitment as plain CONIKS does. In our prototype implementation, we use the elliptic-curve based VUF and signature scheme (EC-Schnorr) proposed with CONIKS (CONIKS also can be implemented with RSA or BLS, but we ignore these to match the cryptography used in Ethereum). We consider two cases for EthIKS: clients which trust the Ethereum network and clients which ignore the Ethereum network (legacy clients). We simulated the same scenario used as a benchmark in CONIKS: $N = 2^{32}$ total users, $n = 2^{21}$ user updates per epoch and k epochs per day.

	CONIKS	EthIKS		
	default	legacy client	light client	full client
lookup (per binding)	1.2	4.0	7.9	7.9
monitor (per epoch)	0.7	2.6	5.0	5.2
monitor (daily)	17.6	62.4	7.9	1405
audit (per epoch)	0.1	0.1	0	60
audit (daily)	2.3	2.3	0	1400

Table 2: Client bandwidth requirements, in kB for EthIKS and plain CONIKS. Sizes are given assuming a $\approx 2^{32}$ total users, $\approx 2^{21}$ changes per epoch, 24 epochs per day, and $\approx 2^{32}$ total Ethereum addresses. A legacy client ignores Ethereum completely and simply uses the EthIKS provider's signed tree roots. A light client trusts Ethereum and relies on a third party to give it the latest Ethereum block header when needed. A full client trusts Ethereum but downloads all Ethereum block headers locally.

For legacy clients, the performance is asymptotically equivalent to that of plain CONIKS. Each binding proof requires verifying one path in the tree, one VUF, one commitment opening and one signature on the root which require a constant 192 bytes. However, Ethereum's Merkle tree structure is known to be slightly inefficient in being hexary. Assuming N total users and n updates per epoch ($n \leq N$), the binary prefix tree in CONIKS requires a path of length $\lg_2 N$ with 256 bits of data per node to reconstruct the path. By contrast, the Ethereum tree requires a path of length $\lg_{16} N = \frac{\lg_2 N}{4}$, but each node requires up to $(N-1) \cdot 256$ bits of data per node. At our simulated size, this increases the path representation from 1024 bytes to 3840 bytes, and the overall binding proof size from 1216 bytes to 4032 bytes.

Ethereum-aware clients can save greatly on monitoring costs by only receiving updated paths when their data actually changes (or at a sampling frequency of their choice). If clients are tracking all of the block headers in Ethereum, this requires downloading about 200 bytes per 12 seconds or 1.4 MB per day. They might also get this data for selected blocks only by querying one or more trusted sources. The complete proof will also require proof that the EthIKS tree is correctly included in the current block, the size of which will depend on the number of contracts in existence. Currently this is less than 1000 so this proof is relatively short, but it might be considerably larger in the future. To be conservative, we assume 2^{32} Ethereum contracts exist, requiring an additional 3840 bytes of data.

We combine these numbers in Table 2 comparing a user in a plain CONIKS system, a legacy user in an EthIKS system, an Ethereum-aware user in EthIKS willing to trust a third party to deliver the current Ethereum block header (a light client) and an EthIKS client which downloads all Ethereum block headers locally. Note that this requires a large amount of bandwidth (1.4 MB per day) but might be useful for other purposes.

6 Concluding discussion

Our analysis shows that EthIKS is a natural extension of CONIKS: it is simple to implement and can be used by legacy clients with minor modifications and only a small performance overhead compared with CONIKS. This performance overhead would be reduced to near-zero by the adoption of a more efficient binary Merkle prefix tree by Ethereum; the adoption of a hexary tree has already been recognized as a regrettable design error [1] that may be fixed in future versions.

For Ethereum-aware clients, superficially additional bandwidth must be used to track the chain of Ethereum block headers. However, this might be useful on its own or be outsourced to a third party. These clients gain a significant advantage over plain CONIKS: keys can be updated very rapidly (bound only by Ethereum's 12 second block generation time). These updates are also independent of the service provider for user-controlled bindings. Furthermore, these clients gain the full security of the Ethereum consensus protocol against equivocation of the provider's state or corruption of the update counters. Overall, this greatly simplifies the service as promises to update are no longer needed (due to the fast update time) and a separate gossip protocol can be eliminated.

The idea of building a naming system on top of Ethereum or other cryptocurrencies is not new. Namecoin [5] was the first formal fork of Bitcoin, designed to provide a distributed naming system, and the simplicity of implementing Namecoin in Ethereum (requiring only a few lines of code in the simplest form) has even been used as a "Hello world!" teaching example of Ethereum programming. However, Namecoin has struggled to gain any significant use, with nearly all registered names currently held by squatters [3] and no clear economic model for assigning valuable names. It also offers no privacy for users, making it difficult to retrofit to existing communication services. CONIKS (and in turn EthIKS) addresses these problems by assuming a centralized service, which controls the assignment of names and maintains privacy by managing a secret VUF key to obscure name-key bindings. However, the central provider in CONIKS is not fully trusted to avoid inserting spurious keys or equivocating about the state of the system. This is prevented by public monitoring and auditing.

Our contribution is EthIKS, which improves on this design by leveraging the Ethereum network to do this checking. Assuming Ethereum proves to be a secure consensus computer in practice [6], EthIKS can enable greatly improved efficiency for clients willing to trust the integrity of Ethereum, while enabling normal CONIKS-like operation for legacy clients. We have implemented this and shown that it is possible today for small providers, costing hundredths of pennies per update to the tree. While the current network may not scale to large providers requiring millions of updates, our work shows that it is asymptotically efficient and therefore possible as the Ethereum network itself scales.

References

1. Ethereum Design Rationale. `https://github.com/ethereum/wiki/wiki/Design-Rationale`, 2016.
2. Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. A Programmers Guide to Ethereum and Serpent, May 2015.
3. Harry Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of Namecoin and lessons for decentralized namespace design. *Workshop on the Economics of Information Security (WEIS)*, June 2015.
4. B. Laurie, A. Langley, E. Kasper, and Google Inc. RFC 6962 Certificate Transparency, Jun. 2013.
5. Andreas Loibl. Namecoin. `namecoin.info`, 2014.
6. Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.
7. Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Michael J. Freedman, and Edward W. Felten. CONIKS: Bringing Key Transparency to End Users. In *USENIX Security*, August 2015.
8. Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *ACM Conference on Principles of Programming Languages (POPL)*, January 2014.
9. Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `http://bitcoin.org/bitcoin.pdf`, 2008.
10. Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure Messaging. In *IEEE Symposium on Security and Privacy*, May 2015.
11. Gavin Wood. Ethereum: A secure decentralized transaction ledger. `http://gavwood.com/paper.pdf`, 2014.