Recording Remote Attestations on the Blockchain

©2023

Andrew Cousino

B.A. Mathematics, Knox College, 2004

Ph.D. Mathematics, Kansas State University 2013

Submitted to the graduate degree program in Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Masters of Science.

Perry Alexander, Chairperson

Committee members

Alexandru Bardas

Drew Davidson

Date defended: May 3, 2023

The Thesis Committee for Andrew Cousino certifies
that this is the approved version of the following thesis :

Recording Remote Attestations on the Blockchain

Perry Alexander, Chairperson

Date approved: May 3, 2023

Abstract

Remote attestation is a process of establishing trust between various systems on a network. Until now, attestations had to be done on the fly as caching attestations had not yet been solved. With the blockchain providing a monotonic record, this work attempts to enable attestations to be cached. This paves the way for more complex attestation protocols to fit the wide variety of needs of users. We also developed specifications for these records to be cached on the blockchain.

Acknowledgements

I would like to acknowledge the support of my mother, Marsha; my advisor, Prof. Alexander; and the research group that Prof. Alexander has assembled.

Contents

List of Figures

Chapter 1

Introduction

Trusting other computers on a network becomes more hazardous every day. This work is a tiny step towards reducing those hazards. Remote attestation is a way of building trust between systems on a network. An attestation request of a target from an appraiser is a sort of health checkup for the target. A cache of these checkups forms a health record of the target. This can cut down on repeat appraisals. And opens future avenues of research for trusting a system based upon its history.

For example, an administrator adding a new printer to a network can appraise the printer offline ensuring that the device booted from a secure state, with up-to-date firmware. See Perlroth (2013) for an attack on networked printers. This can form the initial checkup of the health record. Every reboot or firmware update can be appraised and have such a check up added to the health record. Making this record available for others on the network to read will allow third parties to be sure that this printer is approved. Unlike with people, these health records of systems may be made widely available.

In order to be widely available on the network, health records must be able (1) to be appended by appraisers, (2) to not be modified, (3) to not be erased, and (4) to be available to all on the network. The first three conditions form a sort of monotonicity for the records in order to allow proper functioning and to prevent covering up a bad checkup. The fourth condition is a requirement shielding the health records from a denial of service attack. A private blockchain meets all these requirements. Blockchain is monotonic, and its decentralized nature helps with the high availability requirement.

We use the blockchain as a database, not because it is trendy, but because in this scenario a decentralized ledger is advantageous to a centralized database. A central database is by its very

nature able to be modified as well as erased, allowing an attacker to alter/remove bad checkups. This makes such a database a single point of attack on the network. They are also a single point of failure for denial of service attacks. And so, the choice of blockchain is not made capriciously.

Finally, we wrote specifications for these health records in the proof assistant Coq. In order to do temporal specifications, we used Grant Jurgensen's computational tree logic library from his masters thesis (Jurgensen, 2022), which was developed inside our research group. We formally specified our health records and proved that it satisfies some monotonicity criteria. This is assuming that an adversary cannot override the blockchain consensus mechanism, e.g. a $50\% + 1$ attack. Such an attack is outside our threat model as it requires the attacker to be in control of more of the network than we can reasonably accommodate.

Chapter 2

Background

2.1 Blockchain

In 2008, the pseudonymous Satoshi Nakamoto wrote a white paper (Nakamoto, 2008) describing the cryptocurrency Bitcoin and its underlying blockchain technology. A blockchain is a decentralized, distributed, digital ledger that consists of records, or blocks, which account for transactions between multiple computers, or nodes. The originating block in a blockchain is known as the genesis block. All other blocks include a hash of the previous block, linking all the blocks into a chain, and a cryptographic signature to ensure integrity.

2.1.1 Ethereum

Note that Satoshi's Bitcoin can be viewed as a state transition system, in which the state is the ownership status of every mined bitcoin and the transition function takes a state and collection of transactions (including the mining of new coins) to produce a new state. The Bitcoin blockchain is the path through the system which starts at the genesis block and ends at the present time. This blockchain allows for a limited, stack-based scripting language. Ethereum is an open source blockchain contrived by Vitalik Buterin. Buterin and other founders created Ethereum to provide the additional functionality of allowing nodes to run immutable, Turing-complete, decentralized applications—smart contracts—on this new blockchain.

In Ethereum, the state is made up of objects called *accounts* which have 20 byte addresses. Accounts consist of four fields: (1) a counter called the *nonce*, which counts how many transactions the account has sent and act as a unique identifier for the account's transactions; (2) the account's

current ether balance, where ether is the primary cryptocurrency of Ethereum; (3) the account's contract code, if any; and (4) the account's storage, which is empty by default. There are two types of accounts: externally owned accounts which are controlled by private keys and contract accounts which are controlled by their contract code. The state transitions are direct transfers of value and information between accounts. Externally owned accounts have no code and can send messages by creating and signing transactions. Contract accounts execute their code upon receipt of a message which allows them to read and write to internal storage, send other messages, and create new contracts. Further, contract accounts have direct control over their own ether balance and their own key/value storage which allows for persistent variables.

Transactions within Ethereum are cryptographically signed data packages that store messages, sent by externally owned accounts, and contain the following information.

1. The message recipient
2. A signature of the sender
3. An amount of ether to transfer from the sender to the recipient
4. An optional data field
5. A `StartGas` value which is the maximum number of computational steps the transaction's execution is allowed to take
6. A `GasPrice` value which is the fee the sender pays per step of computation

The gas fields act to prevent denial of service attacks against the system. Messages are like transactions except that they are sent by contract addresses instead of externally owned addresses. Messages are objects that are never serialized, exist only within the Ethereum execution environment, and contain everything that a transaction does with the exception of the signature and the `GasPrice` field (Buterin, 2022).

We are using a private, Ethereum blockchain to cache the results of remote attestations in a health record.

2.2 Remote Attestation

Remote attestation is the act of creating evidence for properties of a target and transferring this evidence over a network to another party, known as the appraiser. Evidence takes the form of direct and local measurements, taken by the appraiser, of the target. An protocol for remote attestation is a cryptographic protocol involving a target, an appraiser, an attester, and possibly other parties acting as proxies of trust. Such a protocol is supposed to supply evidence which the appraiser considers authoritative while respecting the privacy goals of the target (Coker et al., 2010).

2.2.1 Copland

Copland is a domain specific language with formal semantics designed to expresses attestation protocols. A phrase, which is a term in the language, describes a single protocol.

$$
\begin{array}{rcl}
A & ::= & \mathsf{USM}\,\overline{a} \\
 & | & \mathsf{KIM}\,P\,\overline{a} \\
 & | & \mathsf{CPY} \mid \mathsf{SIG} \mid \mathsf{HSH} \mid \cdots \\
T & ::= & A \\
 & | & @_P T \\
 & | & (T \to T) \\
 & | & (T \overset{\pi}{\prec} T) \\
 & | & (T \overset{\pi}{\sim} T)
\end{array}
$$

The meta-variable P represents a place, which is where attestation protocols are interpreted. $\mathsf{USM}\,\overline{a}$ is a user-space measurement with $\overline{a}$ abstracting the details. $\mathsf{KIM}\,P\,\overline{a}$ is a kernel integrity measurement such as those done by the Linux Kernel Integrity Monitor (Loscocco et al., 2007). The CPY, SIG, and HSH are a minimal set of evidence operations which represent copying, signing, and hashing.

The phrases $T_1 \to T_2$, $T_1 \overset{\pi}{\prec} T_2$, and $T_1 \overset{\pi}{\sim} T_2$ compose two sub-phrases and dictate an execution order. $T_1 \to T_2$ specifies that T_2 follows T_1 and consumes the evidence that T_1 produces. $T_1 \overset{\pi}{\prec} T_2$

requires sequential execution also but splits prior evidence between T_1 and T_2 using the projection functions $\pi := \langle \pi_1, \pi_2 \rangle$, with T_1 pulling from π_1 and T_2 from π_2. $T_1 \overset{\pi}{\sim} T_2$ allows for asynchronous execution, again splitting evidence between the sub-phrases using the projections forming π. $@_P T$ has the place P interpret the phrase T and return the resulting evidence (Petz & Alexander, 2019).

Our remote attestations will be done in the Copland language. In other words, the nature of each checkup in a health record will be described using Copland.

2.3 Tools

The programming language CakeML and proof assistant Coq were the two primary tools used in this work. CakeML is a variant of the Standard ML programming language. The formal semantics and compiler for CakeML is verified in HOL4 proof assistant. This language, along with some foreign function calls to C, was used to talk to the Ethereum blockchain as well as implement the attestation manager. Coq is another proof assistant which utilizes the calculus of inductive constructions, a higher-order typed lambda calculus. Coq was used to model the health records and prove the model correct.

2.4 Literature Review

Bampatsikos et al. (2019) came up with a way to for a prover, or attestation target, with limited computational power to prevent a denial of service attack by a malicious verifier, or appraiser. The idea is that in mission critical applications, an IoT device can use remote attestation to protect itself and its users from attack by acting as a prover to another, more computationally powerful machine which acts as an appraiser, whose job it is to attest to the health of the prover. However, it may be possible to overwhelm the simple prover with a multitude of requests from the appraiser. In order to prevent a malicious appraiser from launching a denial of service attack against the target, the authors purpose that the appraiser pay a fee on the public Ethereum network for every request. This prevents such attacks by making them prohibitively expensive.

In Javaid et al. (2020), the authors purpose to utilize blockchain technologies to act as a distributed database of trusted IoT devices. The nodes of the blockchain not only mines for new blocks but also appraises the IoT devices—which are the provers—and registers new devices. One of the main contributions is the use of physically unclonable functions (PUFs) which act as roots of trust for the IoT devices which likely lack a hardware root of trust such as a Trusted Platform Module (TPM). Devices which are able to provide valid proof of their proper state are put on a trusted list, while those that cannot provide proof are put on a restricted list or even dropped from the network.

Our work does not require the use of blockchain for its cryptocurrency purposes, nor does it require blockchain nodes to perform special tasks like device registration and verification. Instead, we focus upon devices with sufficient computational power and use the blockchain only as a distributed database to store results of attestation requests. This means that not only can a verifier validate the most recent attestation request of a prover, but also validate any and all of the prover's past requests. This work opens up new discussions about how a device can be judged based upon its history.

2.5 Software Bill of Materials

- CakeML version 1322
- ganache version 7.1.0
- GCC version 11.3.0
- geth version 1.10.17
- nodejs version 12.22.9
- solidity version 0.5.16
- Truffle version 5.5.14

- web3.js version 1.5.3

Chapter 3

Design and Implementation

3.1 CakeML

3.1.1 Communicating with the Blockchain

One can talk to the Ethereum blockchain using JSON remote procedure calls over HTTP. A very basic C FFI shim for TCP/IP communication is used in front of the CakeML code as shown in figure 3.1. Using algebraic data types to represent HTTP requests and responses, we can read and write HTTP over the network. When reading responses, parser combinators are employed during the interpretation. The body of the HTTP messages are JSON encoded data, which is again represented in CakeML by algebraic data types, with parser combinators assisting in the reading as well. This minimal C shim shrinks our implementations exposure to attacks and other bugs to more acceptable levels. The parser combinator code as well as the code to parse HTTP messages and the JSON bodies was tested by trying it out on a handful of examples that should parse successfully and a handful of examples that should fail to parse. The code to format or print HTTP messages and the

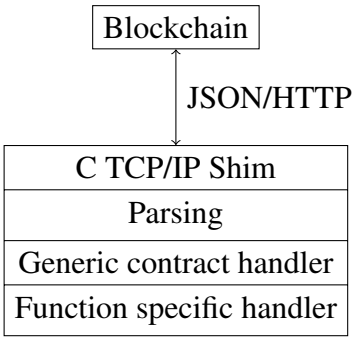

Figure 3.1: Communication with blockchain using CakeML

JSON bodies was tested by visually inspecting the output of a handful of examples and by piping the results to the blockchain to see whether an error resulted.

An example POST query and response is posted below which will call a smart contract method that creates a transaction on the blockchain.

```
HTTP POST query:
{   "jsonrpc": "2.0",
    "method": "eth_sendTransaction",
    "params": [{
        "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
        "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
        ...,
        "data": "0xd46e8dd6..."
    }],
    "id": 1
}
Response:
{   "id":1,
    "jsonrpc": "2.0",
    "result": "0xe670ec64..."
}
```

The fields of the JSON objects are described below.

- `method` tells the blockchain software what type of call this will be.
- `from` is the address of the caller, and who is responsible for the gas fees.
- `to` is the address of the callee which in our case is a smart contract.
- `data` contains the method of the smart contract to be called as well as the value of the parameters in contract ABI form.

- `result` has the results of the method call in contract ABI form.

The contract application binary interface (ABI) format is a way to encode the values on which smart contracts operate. It is a 32 byte wide encoding with variable length values such as strings prefixed by their length. As an example, encoding the pair `("hello world", 29)` as a `(string, uint256)` tuple, one would start with the string would be prefixed by its length—11 or `0xb`—and then be converted into UTF-8 format padded with zeros in the least significant bits, then the number would be converted to its hexadecimal equivalent `0x1d` padded with zeros in the most significant bits. Finally, these 32 byte wide hexadecimal strings are concatenated together in the following order: the number of bytes from the start of the result to the start of the length of the string `"hello world"`—64 or `0x40` bytes, the encoded value of the unsigned integer, the encoded length, and the encoded string. In the end, we get the following byte string.

```
0x0000000000000000000000000000000000000000000000000000000000000040
  000000000000000000000000000000000000000000000000000000000000001d
  000000000000000000000000000000000000000000000000000000000000000b
  68656c6c6f20776f726c6400000000000000000000000000000000000000000000
```

In order to call a function with signature `foo(string, uint256)` with this particular tuple value, we would take the hash of the signature using the Keccak-256 hash function, take the most significant 4 bytes from the hash, and prepend them to the ABI encoded tuple yielding `0x24ccab8f00...00`.

As the JSON messages from the blockchain address the particular smart contract function to be called, we have a CakeML function to send and receive messages directed at each smart contract function. However, we capitalize upon the remaining, common structure of these messages with generic CakeML functions which are called by the more specialized functions. Again, refer to figure 3.1 for a diagram outlining this process. Both generic and specific CakeML functions return a `Result` data type which encapsulates correct, expected return values in an `Ok` constructor and all other return values in an `Err` constructor, short for “Error”, which in these cases wraps a string-valued error message. As there are multiple ways that communicating with a blockchain can

fail, returning an error message seemed to be the more useful for future users of this work. The code to encoding and decoding the Ethereum ABI was tested by communicating with the blockchain and getting the expected results.

3.1.2 Health Record

The health record for a node on a network is stored on the blockchain as a JSON object. It has the following fields.

- `appraiserId` a byte-string representing the identifier of the appraiser. We plan on having this be a hash of the appraiser's public key.
- `phrase` a JSON representation of the Copland phrase.
- `result` a JSON value representing the result of the appraisal.
- `signature` an optional byte-string holds the appraiser's signature of this stringified JSON object.
- `targetId` the identifier for the target of the attestation, again another byte-string.
- `timestamp` an integer value for the timestamp from when the appraisal was completed.
- `targetPubEncKey` a byte string representing the target's public encryption key.
- `targetPubSignKey` a byte string representing the target's public signing key.

At the moment, the `result` is a simple boolean value indicating whether or not the appraiser approved of the Copland phrase for the target. Right now, we are using simple unsigned integers for the identifiers. The signing algorithm used is RSA with SHA-512. This code that reads and writes health records to the blockchain was tested by writing a record and reading the same record back and comparing the two.

3.1.3 Attestation Manager

The attestation manager is implemented as a server listening for attestation phrases to evaluate and returning the evidence generated. A client looks for a fresh checkup in the health record on the blockchain. If a fresh record is found, the client goes no further. Otherwise, it sends a fixed phrase to the attestation manager, receives the evidence for the phrase, evaluates the evidence, and puts this checkup on the blockchain. In this case, the client is also serving as an appraiser of evidence due to the simplicity of our research group's demos. This code was tested by other members of our research group using code they had written to do other aspects of the particular demo.

3.2 Coq

3.2.1 History

The Coq History module is a generalization of and specification for health records. It defines a "history" of values with type T as a list over a boolean decidable type T, which are types equipped with a boolean equality operator as well as a propositional equality operator which agree with one another. Histories match what the smart contract permits: appending to a list, reading values from the list, and clearing the list entirely. They do not model the access control mechanisms of the smart contract, which permit certain authorized blockchain accounts to modify the list.

```
Definition history := list T.t.
Definition history_add (x: T.t)(h: history) := cons x h.
Definition history_peek (h: history) :=
    match h with
    | nil => None
    | cons x h' => Some x
    end.
Definition history_empty (h: history) := h = nil.
```

```
Fixpoint history_find (x: T.t)(h: history) :=
    match h with
    | nil => false
    | cons x' h' =>
        if T.eqb x x'
        then true
        else history_find x h'
    end.
```

Using the computational tree logic (CTL) developed in our research group for Coq, we proved that immediately after appending a value to a history either the history contains this value or is cleared (`history_found_next`); that whenever a value can be found in a history, at the next moment in time, the same value can be found in the history or the history is cleared (`history_found`); and that a value added to a history stays in the history until the history is cleared (`history_kept`). The last goal is a sound and complete model for the smart contract because the transitions that modify the history list in the smart contract are exactly one transition that appends a single item to the list and another that clears the list entirely.

```
Inductive history_step: relation history :=
| history_add_step:
    forall (h: history)(x: T.t), history_step h (history_add x h)
| history_clear_step:
    forall (h: history), history_step h nil.

Instance history_step_transition: transition history_step :=
    { trans_serial := history_step_serial }.

Lemma history_found_next (h: history)(x: T.t):
    tentails
```

```
        history_step
        (history_add x h)
        (AX (tlift
                (fun h' =>
                    history_find x h' = true \/
                    history_empty h'))).

Theorem history_found (h: history)(x: T.t):
    tentails
        history_step
        h
        (timpl
            (tlift (fun h' => history_find x h' = true))
            (AX (tlift
                    (fun h' =>
                        history_find x h' = true \/
                        history_empty h')))).

Theorem history_kept (h: history)(x: T.t):
    tentails
        history_step
        (history_add x h)
        (AW
            (tlift (fun h' => history_find x h' = true))
            (tlift (fun h' => history_empty h'))).
```

The CTL library works on paths indexed by natural numbers through a state transition system. The `tentails` predicate takes 3 arguments: a relation for the transition system, an initial state, and

a CTL proposition which holds over the transition system. `tlift` lifts a proposition in Coq to one in this CTL system. `timpl` is the implication in the embedded CTL language. The temporal quantifiers used in the above goals are `AX` and `AW`. `AX` takes one CTL formula and states that it holds in every state which immediately follows. `AW` takes two CTL formulas and acts as an "until"—*AW* ϕ ψ states that ϕ holds at every reachable state until ψ holds. So either ϕ can hold for all reachable states or ϕ holds until ψ holds with no overlap.

3.2.2 Cryptography

The cryptography module models keys as an enumerated inductive type which can either be a symmetric, public, or private key. Each key is indexed by a natural number, and has an inverse key—symmetric keys are their own inverses, public keys have a corresponding inverse private key, and similarly for a private key.

```
Inductive key: Set :=
| symmetric : nat -> key
| private : nat -> key
| public : nat -> key.

Definition kinverse (k: key): key :=
    match k with
    | symmetric _ => k
    | private k' => (public k')
    | public k' => (private k')
    end.
```

Data over a type T can either be a value of type T, a cryptographic key, the result of encrypting another data value with a key, the result of hashing data, or a pair of data values. Signing a data value consists of pairing the value with an encrypted hash of the value. It was already proved that

only the inverse of the signing key can verify the signature (`sig_check`), and that if the process of checking the signature of a data value against a key succeeds, then that data value was the result of signing another value with the inverse key (`sign_only`).

```
Inductive data (T: Type): Type :=
| value : T -> data T
| kval : key -> data T
| kapply : key -> data T -> data T
| hash : data T -> data T
| pair : data T -> data T -> data T.

Definition sign {T: Type}(d: data T)(k: key): data T :=
    pair d (kapply k (hash d)).

Definition check {T: Type}(d: data T)(k: key): Prop :=
    match d with
    | pair d' s' =>
        match s' with
        | kapply k' s'' =>
            k = kinverse k' /\ s'' = hash d'
        | _ => False
        end
    | _ => False
    end.

Theorem sig_check {T: Type}(d: data T)(k1 k2: key):
    k1 = kinverse k2 ->
    check (sign d k1) k2.
```

```
Theorem sign_only {T: Type}(d0: data T)(k0: key):
    check d0 k0 ->
    exists (d1: data T)(k1: key),
        sign d1 k1 = d0 /\ k0 = kinverse k1.
```

3.2.3 All Together

A health record in Coq starts with a boolean, decidable type T. This type is passed into the cryptography `data` type, which we showed is again a boolean, decidable type. Then the cryptographic data is wrapped by the history module. The `pair` construct in the cryptography type allows us to create records of arbitrary length. The types of the record's fields being a boolean decidable type gives us the ability to store any basic, practical types that we wish. In other words, we have certainly modeled the health records which are being written to the blockchain, along with more general data types.

Chapter 4

Conclusion

We have developed health records for remote attestations to be put on a private blockchain. This allows a node on the network to see the entire attestation history of one of its neighbors. The analysis of such a history is now an open research question. Also, we proved properties for these health records that establish a kind of monotonicity. Another thing left for future work is the scalability of this health record system. We have only tested this on a network with a handful of systems on it.

But the lowest hanging fruit of further research is implementing certificate-style attestation flexible mechanism Helble et al. (2021). In this system, shown in figure 4.1, the appraiser of evidence is separated from the client into its own process or even its own node on the network. The client would then check the blockchain for a fresh health record, signed by the appraiser. If no such record is found, the client sends their Copland phrase to the attester. This attester performs the prescribed measurement, generates the corresponding evidence, and sends the evidence onto the appraiser. The appraiser will evaluate the evidence, place a new health record onto the blockchain containing their evaluation, and hand their evaluation back through the attester to the client.

This work on certificate-style attestation is already under way. And it will be using this work on blockchains in order to store appraisals in a monotonic way.

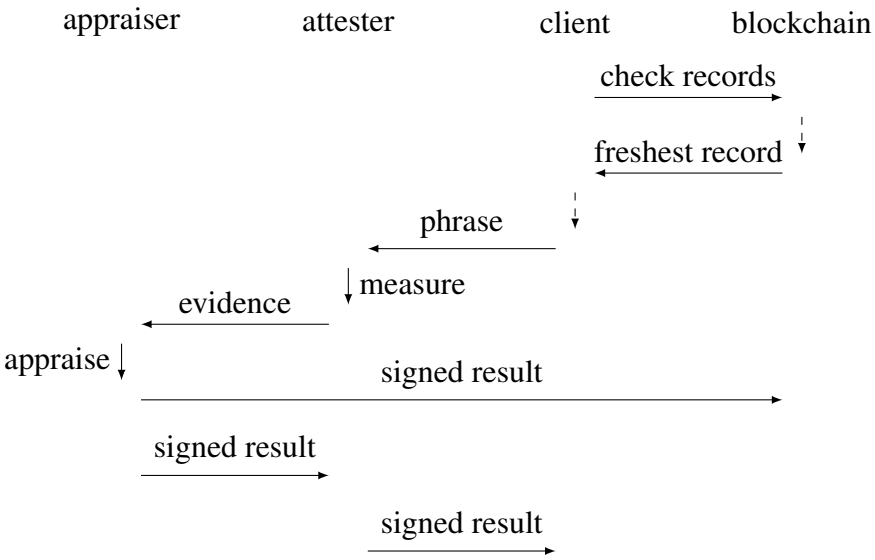

Figure 4.1: Certificate protocol

References

Bampatsikos, M., Ntantogian, C., Xenakis, C., & Thomopoulos, S. C. A. (2019). Barrett blockchain regulated remote attestation. In *IEEE/WIC/ACM International Conference on Web Intelligence - Companion Volume*, WI '19 Companion (pp. 256–262). New York, NY, USA: Association for Computing Machinery.

Buterin, V. (2022). *Ethereum whitepaper*. Technical report, Ethereum Foundation.

Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., Ramsdell, J., Segall, A., Sheehy, J., & Sniffen, B. (2010). Principles of remote attestation. *International Journal of Information Security*.

Helble, S. C., Kretz, I. D., Loscocco, P. A., Ramsdell, J. D., Rowe, P. D., & Alexander, P. (2021). Flexible mechanisms for remote attestation. *ACM Trans. Priv. Secur.*, 24(4).

Javaid, U., Aman, M. N., & Sikdar, B. (2020). Defining trust in iot environments via distributed remote attestation using blockchain. In *Proceedings of the Twenty-First International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, Mobihoc '20 (pp. 321–326). New York, NY, USA: Association for Computing Machinery.

Jurgensen, G. (May 2022). A verified architecture for trustworthy remote attestation. Master's thesis, University of Kansas.

Loscocco, P. A., Wilson, P. W., Pendergrass, J. A., & McDonell, C. D. (2007). Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, STC '07 (pp. 21–29). New York, NY, USA: Association for Computing Machinery.

Nakamoto, S. (2008). *Bitcoin: A peer-to-peer electronic cash system*. Technical report, Bitcoin Project.

Perlroth, N. (2013). Chinese hackers infiltrate new york times computers. *New York Times*.

Petz, A. & Alexander, P. (2019). A copland attestation manager. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*, HotSoS '19 New York, NY, USA: Association for Computing Machinery.