Verifiable Computation in Smart Contracts

Thomas Kerber

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2017

Abstract

Recent developments in blockchain technologies have seen the popularisation of smart contracts, that is, distributed programs which store their state on a blockchain. Usage of these programs has practical restrictions, however, including both hard and soft limits on the number of calculations they can perform. This project aims to provide a proof-of-concept implementation of how techniques from verifiable computation may be used to reduce the practical costs of using smart contracts, enabling them to be used in a wider range of applications. Verifiable computation is typically seen from the perspective of a user wishing to outsource computation to a third party, but wanting guarantees that the party provides the correct results. In the case of smart contracts, this scenario is flipped, with the user wishing to perform the computation themselves, and guarantee the correctness to the rest of the network. This project provides a functioning means of utilising verifiable computation in an Ethereum fork, and determines the challenges in extending this into a practical, widely used technology.

Acknowledgements

I would like to thank my supervisor, Aggelos Kiayias, for his guidance in finding and refining the project goals.

I would further like to thank my father, Manfred Kerber, for his proofreading this report.

Finally, I'd like to thank all of my friends and family who supported me during this project, and ensured I remained motivated to the end.

Table of Contents

Chapter 1

Introduction

Blockchain technology has provided a means to create a distributed consensus about the state of a system [39]. While financial transactions were the first, and are still the most prominent application of this technology through BitCoin [25], the systems blockchains can create consensus over can be much more complex. Recently, and crucially for this project, Ethereum has implemented a blockchain which creates consensus on the state of a general purpose virtual machine, the Ethereum Virtual Machine, or EVM [39]. Ethereum allows specifying ‘contracts’, programs which manage finances, entirely without the oversight of any one party. Instead, the network as a whole ensures that the programs are executed correctly, ensuring that the contract is transparent and accountable. As a result, a contract can be trusted by a third party, even if the party trusts no particular node in the system [39].

This consensus does not come for free, however. Blockchain technologies have been struggling with scalability for a while, even BitCoin, which only maintains a ledger of transactions, has difficulties [7]. For Ethereum, where the state being stored is more complex, this is even more focal. This is because every node in the networks needs a copy of the system’s state. Further, as the protocol is built on mutual accountability, all (full) nodes must verify that the state of the system is correct [25]. For BitCoin, this is a simple matter of checking a number of hashes, signatures, and account balances. For Ethereum, things are more complex. As the state is the state of a powerful virtual machine, verifying it is done by recomputing the steps the virtual machine must take [39]. In effect, any computation performed on the EVM is performed on **all** (full) nodes in the Ethereum network. To combat this quadratic growth in computation effort required, as well as to prevent denial of service attacks, a dynamic cost is imposed on all calculations made [39]. While this solves the problem from a technical standpoint, by letting market economics decide the price of computation, it does not solve the practical problem: Some computations will not be performed, because it is too expensive to perform them.

For instance, smart contracts governed by more complex rules, such as a neural network, or state-space search, are economically infeasible. This project discusses one potential approach for alleviating the burden placed on verifying nodes, allowing for more complex calculations, and greater scalability of Ethereum. While the technology, as implemented in this project, is not ready for active use, it demonstrates a theoretical possibility and sketches the groundwork for an efficient implementation. It utilises prototype implementations of research in verifiable computation, integrating it into the existing Ethereum ecosystem. Verifiable computation provides a means to prove the correctness of a computation, such that verifying this proof is computationally less complex that performing the computation itself [29].

1.1 Contributions

My contributions to this area lie in integrating verifiable computation into Ethereum [21], specifically into the Solidity language [17] and `solc` compiler [18], as well as the EVM implementation provided by the Parity client [36]. Specifically, the following core work has been done:

- The Solidity language has been extended to support the declaration of functions supporting verifiable computation.
- The `solc` compiler has been modified to function with the extended Solidity dialect, compiling partially into verifiable code.
- The Parity EVM implementation has been extended to support a verification built-in, and a blockchain utilising this was created.
- The Parity client has been extended with a proof generation API call.

Chapter 2

Background

This chapter will briefly sketch the background required to fully understand the work done within this project. It will provide a high-level view of verifiable computation, as well as a more in depth basis to the understanding of blockchains and Ethereum.

2.1 Verifiable Computation

With the recent rise in popularity in cloud computation, the question of how to ensure that the providers of cloud services are actually performing the computations they claim to be doing has arisen [34]. If a cloud service has been tasked with computing the result of a function f, how can the client be sure that the result for an input x they receive is correct? They could recompute $f(x)$ themselves and compare the values, but this would defeat the purpose of using a cloud service for the computation in the first place. It is possible, of course, that there is an easy way to check if the output is correct for some particular f. For instance, if $f = h^{-1}$ for some hash function h, f would be very difficult to compute, while h would be easy. However, this does not work for arbitrary computation, and is situation specific.

Verifiable computation proposes to solve this problem by requiring the cloud service to, instead of just computing $y = f(x)$, compute a proof p as well [29]: $(y, p) = \mathcal{P}(f, x)$, which the client can verify using a verification function $\mathcal{V}(f, x, y, p)$. The verification function $\mathcal{V}$ must be such that it returns true only if $y = f(x)$ and $(y, p) = \mathcal{P}(f, x)$. This verification stage should be computationally less complex than computing $f(x)$ by itself [29].

2.1.1 Pinocchio

Pinocchio is a tool developed by Microsoft Research to implement verifiable computation in a semi-practical way for simple C programs. This section is sourced from [28], Pinocchio's research paper. Pinocchio consists of two parts: a C to arithmetic circuit compiler, and a prover/verifier tool. For this project, only the latter is of interest. It is ‘semi-practical’, as it is the first published implementation of verifiable computation to achieve verification speeds smaller than native execution speeds for some programs. Notably, however, for many programs verification takes longer than native execution would, and even when it outperforms native execution it does not do so by a large margin. Pinocchio should therefore be considered a proof of concept, and not a finished product.

Pinocchio itself, as briefly mentioned above, does not work on native machine code, but instead operates on arithmetic circuits. These are roughly analogues to boolean circuits, however instead of carrying boolean values along the wires, they carry integers, and instead of boolean gates, gates are additions or multiplications. Its arithmetic circuits operate on 254-bit fields, which Pinocchio uses to emulate 32-bit native integers. This is part of the reason for Pinocchio performing worse than native code in some situations; computing the program via its arithmetic circuit inherently carries a large overhead from dealing with the increased word size. While verification performs worse than *native* code execution in some cases, it almost always performs better than evaluating the circuit itself. Arithmetic circuits themselves carry inherent limitations as well, as they are fundamentally a circuit, not a program. More specifically, the circuits are without feedback, and therefore have a fixed, finite evaluation time. This also means, however, that they are inherently not Turing-complete, and that not all programs can be correctly compiled to arithmetic circuits. Pinocchio’s compiler primarily deals with this problem by unrolling all loops, up to a maximum number of iterations.

Pinocchio has four stages of execution: Compilation to arithmetic circuits, key generation, proving, and verification. These must happen in order, and depend on each other. More specifically:

- Compilation depends on the input program, and produces an arithmetic circuit.
- Key generation depends on the arithmetic circuit, and generates a public proving and verification key.
- Proving depends on the arithmetic circuit, the proving key, and the circuit input, and generates a circuit output and a proof.
- Verification depends on the arithmetic circuit, the verification key, the circuit input, the circuit output, and the proof, and either succeeds or fails.

It is worth noting that due to the nature of the underlying cryptography, if the parties verifying the proof, and the party generating the keys are different, the key generating party must be trusted. This is because the key generating party could create a cryptographic backdoor, allowing it to ‘prove’ false statements. While this is not important in the traditional application of verifiable computation, as the key generating party, and the verifying party are the same, this is not the case for this project.

2.2 Blockchains

Blockchain technologies started with the release of BitCoin in 2009, with BitCoin rapidly rising in popularity over the past few years [8]. This section is mostly sourced from the original BitCoin whitepaper [25], except where stated otherwise. As of the time of writing, one BitCoin is valued at £1,027 [40]. With over 16 million BitCoin having been created since 2009 [32], this makes BitCoin alone a multi-billion pound technology. BitCoin follows multiple attempts at creating a secure online currency, and solves multiple problems predecessors had [23]. Most notably, previous decentralised solutions were vulnerable to double-spend attacks, in which an attacker Mallory who possessed a coin could spend it to both the parties Alice and Bob in short succession. As there was no way to tell whether a coin had been previously spent, schemes typically had to employ a centralised arbiter to prevent these attacks [23, 25]. BitCoin solved this problem by creating a distributed ledger of all transactions. This forces all parties in the protocol to (largely) agree on the sequence of events, providing a means to settle disputes in the case of a double-spend.

While it may initially seem trivial to create such a decentralised ledger, a naive solution would be vulnerable to a Sybil attack, in which an attacker could flood the network with nodes to change the consensus [10]. This would allow an attacker to rewrite history, and effectively undo an accepted transaction. BitCoin's core innovation lies in its solution to this, preventing double-spend attacks, and even providing a means to bootstrap the currency as an added bonus. BitCoin's solution to this problem lies in making entries in the ledger computationally infeasible to change after sufficient time. This is done by collecting transactions into blocks. Each block further points to the previous block, building a linked list, or chain; the origin of the term blockchain. Each block requires a party, typically known as a miner, to provide a proof-of-work. This is in the form of the hash of the block needing to be less than a protocol-specified value. This prevents a Sybil attack, as creating blocks requires computational resources. At any time, the longest chain of valid blocks is accepted by the network as the current ledger.

This protocol is secure even against an attacker with sufficient resources to generate blocks, provided he does not control the majority of block generation. This is because the attacker cannot feasibly attempt to rewrite the history of the ledger. Without the majority of hashing power, the remaining, honest miners will make more progress than the attacker. Due to this, the honest miners blockchain will be accepted by the network over the attackers. Of course, as the generation of blocks is probabilistic, in the short term the attacker could succeed, which is why the most recent few blocks are not considered fixed [20]. For transactions deeper in the blockchain however, it is almost impossible for any party to rewrite them, whatever the reason. This leaves BitCoin with a need to incentivise honest parties to mine blocks. The elegant solution it provides for this is to create a set number of coins in each block, which the miner of the block can claim for themselves. This also solves the problem of how to bootstrap the currency, without biasing the distribution toward any parties aside early adopters. This has led to accusations of BitCoin being a pyramid scheme, however there are notable differences: BitCoin does not claim it will increase in popularity or price, and provides tangible practical value outside of simply being an investment. In this sense BitCoin more closely resembles any other investment.

2.3 Smart Contracts

Smart contracts are not an entirely new concept, indeed BitCoin itself had limited support for them [26]. This section is largely sourced from the Ethereum yellow paper [39], except where marked otherwise. A smart contract is a generalisation of the transaction ledger used in BitCoin. Instead of recoding a sequence of financial transactions, the ledger is used to record the change of a general-purpose virtual machine's state This allows for enforcing arbitrary, user-defined rules for interacting with the blockchain, effectively allowing the creation of new, generic blockchain protocols on the same chain. Some examples of possible applications include auctions [30], market prediction applications [19], or, in a particularly high profile case, the creation of an entirely autonomous organisation [11]. BitCoin's support for these applications is very limited, as it is not Turing-complete [9], and as a result it cannot be used as flexibly as people would wish. Recently Ethereum, a blockchain with full smart contract support was released.

2.3.1 Ethereum

This project primarily concerns itself with the Ethereum blockchain, which was first launched in 2015 [38]. The Ethereum blockchain is the first Turing-complete blockchain, however it also has its own cryptocurrency, Ether, associated with it. Ether are, as of the time of writing, the second most traded cryptocurrency, valued at £39.82 per Ether [5]. Ether has several denominations, the smallest of which, the Wei, is worth 10^{-18} Ether.

2.3.1.1 Accounts, Transactions, and Contracts

Ethereum identifies users and contracts, which will be discussed shortly, by a 160-bit hash. This has identifies the underlying account, which can either be a user account, or a contract account. Accounts can hold funds, in the form of Ethereum's native cryptocurrency, Ether. To transfer funds between accounts, or call part of a contract's API, a transaction needs to be made. Transactions must originate from user accounts, and need to be signed by the user's private key. A transaction has a value, which represents the amount transferred, an amount of so-called 'gas', which represents a transaction fee, as well as an upper limit for the computations that can be performed, as all operations performed within the scope of the transaction carry a cost in gas. This prevents over-utilisation of computational resources in the EVM, by letting their price be determined through market economics. Finally, the transaction has a data segment, which specifies arbitrary data sent to a contract.

A contract is a special account, created by making a transaction to the zero address, passing the contract's bytecode in as the transaction data. When a transaction is made to a contract, the contracts bytecode determines the actions it takes. The contract has its own storage area it can manipulate (see below), and can make transactions of its own. Contracts need to be externally prompted into this however, and cannot initiate transactions by themselves.

For instance, a contract can be set up to manage auctions, with users sending the contract funds, as well as data specifying which items to bid on. After the auction ends, users who did not place the winning bid could request a withdrawal from the contract, the winning user could request their prize, and the seller could request the winning bid. As this is governed by a smart contract, the users could inspect the source code encoding these rules beforehand, and be assured that their funds will not be stolen by a corrupt auctioneer.

2.3.1.2 Ethereum's Memory Model

Ethereum has three separate memory areas, which can be accessed from within a contract. The first is the stack, each entry of which stores 256-bit words. EVM instructions are stack-based, so the stack does not need to be explicitly read from or written to. Using the stack is free in most cases, and very cheap for operations which only manipulate the stack. However, the stack has a hard limit on its size, and elements which are too deep on it cannot be retrieved without removing the upper layers of the stack.

The second type of memory is 'memory', and refers to memory which is local to the current transaction. It is initialised to zeros at the start of the transaction, and is theoretically infinite. Any section of memory can be read from and written to in either byte-level granularity or 256-bit word-level granularity. Although theoretically infinite, use of memory must be paid with gas. The cost is proportionate to the square of the largest address used, making use of memory beyond a few megabytes infeasible for practical use.

The third type of memory is 'storage'. This is used to store data which persists between transactions, effectively defining the contract's state. It is accessed in a similar manner to memory, however the cost model is different. There are separate flat costs for setting a 256-bit word in storage depending on whether or not the word was zero or non-zero previously, and a (partial) refund for setting a non-zero word to zero.

2.3.1.3 Ethereum's Computational Model

The Ethereum virtual machine is a stack-based machine, meaning that by default operations operate on elements of the stack. For example, the `ADD` operation takes two elements off the stack, adds them together and pushes the result back onto the stack. Each operation is denoted by a single byte, and takes no additional data. The only exceptions are the `PUSH`*x* set of opcodes, push the following *x* bytes onto the stack as a single word. The instruction set can be described as a RISK instruction set, as it uses only a small number of instructions, and does not combine operations. This design was chosen to simplify the formal description of the EVM, and reduce the probability of implementation bugs [13].

In order to allow paying for transactions, Ethereum uses a concept that is called “gas”. A transaction needs to specify an amount of gas it wishes to use, and specify the price per gas it will pay. This price is an effective transaction fee, and will be payed to the miner of the block the transaction is picked up in. While the transaction author can determine the gas price, setting a low price will make the transaction less likely to be picked up by miners, as it carries a low reward to them. Gas itself is used to pay for each operation the EVM performs during the transaction. Prices range from 3 gas for simple stack operations, such as the addition of values, or the duplication of values, to 20,000 gas for setting a new value in storage. If a transaction finishes correctly, any gas left over will be refunded to the transaction author. If the gas is insufficient to finish computation, or some other error occurs during the transaction, the transaction will be aborted, and the miner will take the entire transaction fee for it. As of the time of writing, each unit of gas is valued at 20 GWei [35]. At current market prices, this translates to roughly £1 $\equiv$ 1.57Mgas [5].

2.3.2 Solidity

Solidity [17] is currently the primary programming language targeting the EVM. Due to significant differences between the EVM and typically targeted architectures by compilers, such as the different word size, different memory model(s), as well as various new low-level operations (such as transfers), the Ethereum team decided to create a new language targeting the EVM rather than leveraging existing ones. More accurately, they created several such languages, but Solidity is by far the most widely used, and was the focus of this project. An example of a basic smart contract written in Solidity can be found at the end of this section, in Listing 2.1. Solidity describes itself as a “contract-oriented” programming language [17], and has strong similarities to a simple object-oriented language.

A Solidity file consists of multiple contract definitions, each of which consists of state variables, methods, and an optional constructor [17]. A contract does not have to be fully defined (i.e. it can have methods without a body). In this case this contract will not compile to actual code. It can still be useful for calling into other contracts, however, serving a similar purpose as an interface in some object oriented languages. Solidity has a fairly basic type system, however supports some features that are specific to the EVM or are particularly useful for smart contracts. It supports arbitrary-length arrays (although there is no provision for resizing arrays), as well as complex mappings from one type to another. The latter are particularly useful, as they allow for simple expressions of, for instance, how many funds any particular address holds. These types can be annotated as living in memory, or in storage. By default, they are placed in storage, unless they are function parameters. This can be somewhat counter-intuitive, as it also applies to ‘local’ variables in the body of a function. Further, the typing system allows for the declaration of C-style structs.

While the parallels between ‘contracts’ in Solidity, and ‘objects’ in object-oriented languages are apparent, and they serve as a useful way of understanding Solidity, the similarities can be deceiving. In object-oriented languages, objects are used as a means to split a program into modules, and objects are always something to be acted upon, hence their name. In Solidity, each contract is best viewed as its own actor; it is best to avoid the creation of new contracts if at all possible, as each creation is expensive. Further, although contracts can call into each other, contract authors (or anyone checking a contract for correctness) should be wary that the contract they are calling into may be dishonest [17]. Multiple contracts are primarily a means of modelling multiple (automated) parties, not merely dividing a piece of code into modules.

Solidity is a limited programming language; in particular it has no equivalent for traditional heap allocation, severely restricting the usage of dynamically sized data structures. Taking the limitations of the underlying platform, however, this is understandable. Further, the simplicity of Solidity reduces the chance of compiler introduced errors, which is an important bonus given the security-sensitive context.

Listing 2.1: An simple example contract building a cryptocurrency on top of Ethereum. The example is one of Ethereum’s tutorial contracts, copied from [12].

```
contract MyToken {
    /* This creates an array with all balances */
    mapping (address => uint256) public balanceOf;

    /* Initializes contract with initial supply tokens to the
       creator of the contract */
    function MyToken(uint256 initialSupply) {
        // Give the creator all initial tokens
        balanceOf[msg.sender] = initialSupply;
    }

    /* Send coins */
    function transfer(address _to, uint256 _value) {
        // Check if the sender has enough
        if (balanceOf[msg.sender] < _value) throw;
        // Check for overflows
        if (balanceOf[_to] + _value < balanceOf[_to]) throw;
        // Subtract from the sender
        balanceOf[msg.sender] -= _value;
        // Add the same to the recipient
        balanceOf[_to] += _value;
    }
}
```

2.3.3 The `solc` compiler

There are multiple implementations of Solidity compilers, targeting different platforms [17]. For this project, the `solc` compiler [18] was used, as this is the original reference implementation of Solidity. It is a command line compiler, and compiles Solidity code into EVM binaries and JSON ABI specifications for each contract within the code. Furthermore, it has useful options for outputting intermediate outputs, such as the AST, and annotated EVM assembly. The compiler is written in C++, and uses a standard multi-pass model. Parsing is written by hand; it was not generated using a parser generator.

Chapter 3

Verifiable Computation in Ethereum

This project aims to provide a proof-of-concept design and implementation [21] of using verifiable computation to decrease the amount of computation performed on-chain for smart contracts. It does this by modifying the Solidity language and compiler to allow functions to be compiled to arithmetic circuits instead of EVM bytecode, as well as modifying Ethereum itself to support a built-in 'verify' function. The latter is required for a functioning proof of concept, however this project also sketches a design for implementing verifiable computation entirely within the active Ethereum blockchain.

The most substantial part of the work done in this project consists of modifications to the `solc` compiler, and the Parity Ethereum client. As both are fairly large pieces of software, a significant amount of the work involved in this project was growing sufficiently accustomed to both codebases to make modifications to them at various levels. In both cases, the modification covered changes to many parts of the software; they were not changes that involved only a specific module or part of the software's functionality.

3.1 High-Level Design

To begin with, it is important to note the limits of verifiable computation, at least as implemented in Pinocchio. Not only is it not Turing-complete, but it also has the practical limit of not having I/O [28]. While this is an obvious observation, it is important to note, as especially in the realm of smart contracts, I/O is necessary for most practical applications. In particular, if a transaction involving a smart contract never changes any state, there is no reason to make the transaction, as the result can simply be computed off-chain. Solidity even has provisions for this: Functions labelled 'constant' cannot modify state, and are simply evaluated off-chain when executed [17]. However, while this forbids modifying state, it imposes no restrictions on *accessing* state, forbidding only the output side of input/output. As such, 'constant' functions provide insufficient restrictions for using verifiable computation.

A potential solution to this problem is to model the blockchain state σ as an input to the function, and the modified state σ' as an output. While this is a reasonable approach for formal reasoning, it is infeasible for verifiable computation. Not only is σ dynamically sized, making it impossible to model in a circuit, but the overhead of treating the entire state as an input would be immense. This project therefore takes the approach that only parts of a contract will have verifiable computation applied to them, with the rest of it executing as normal. The task of separating the 'I/O' of the contract, and the functional parts falls to the programmer. The compiler will enforce that the functional parts adhere to restrictions, such as not reading from, or writing to state variables.

Conceptually, we now have a program which comprises of multiple functions, some of which we want to apply verifiable computation to, and some of which we do not. But what does it mean to 'apply verifiable computation' to a function? The first thing to realise is that there are *two separate answers*. We either want to evaluate the function *and* generate a proof, or *not* evaluate the function, and instead *verify* a proof. Since the latter is what will occur on-chain, and will therefore be executed by more users, we will consider it first. Modelling the function with the same signature as originally is infeasible, as the verifying stage will require the proof of computation, as well as the actual output of the function itself. Instead of a call $y = f(x)$, we therefore need a call $y = f''(x, p, y)$, where p is the proof. Note that Pinocchio requires two additional parameters for verification: The arithmetic circuit, and the verification public key. These are both implicit to f. It is important to note, that this places the responsibility of generating the public key in the hands of the contract compiler. As such, the original creator of the contract **can place backdoors into it**. This problem is further discussed in Section 4.3.

If we consider the verification function to be a function $\mathcal{V}$ on the input x, the proof p, the output y, the arithmetic circuit f_{arith}, and the public verification key f_{kv}, returning 0 if verification fails and 1 if verification succeeds, then we define f'' as follows, where 'error' refers to the transaction being aborted:

$$f''(x,p,y) := \begin{cases} y & \text{if } \mathcal{V}(x,p,y,f_{arith},f_{kv}) = 1 \\ \text{error} & \text{otherwise} \end{cases} \tag{3.1}$$

So on-chain, we want the function $y = f(x)$ to be replaced with $y = f''(x, p, y)$. A call to f must now also supply a result, and the proof of said result, to correctly execute. Assuming a transaction calls f directly, this is straightforward, as p and y are calculated off-chain, they can just be passed in directly. However, as f is purely functional, there is no purpose in calling just f on-chain. Instead, f will, in useful scenarios, be called by another function g. g will typically access some state on the blockchain depending on the output of f. For example, g may be a function to distribute rewards depending on a complex algorithm encoded in f. As the signature of f is different, however, calls to it in g need to be rewritten to use the new signature. Furthermore, g itself needs to receive the proof and output of f as inputs. More precisely, as f, or other functions using verifiable computation maybe called multiple times in g, g needs to be able to receive multiple proofs and outputs of these functions as inputs.

To solve this, we use an array of proofs and output values. This is a single byte array, of a sequence of proof, output tuples. Each proof, output tuple is proceeded by its length in bytes, and each proof and output is also preceded by its respective length. Both functions that use verifiable computation (such as f above), as well as functions that call the former (such as g above), will be rewritten to take this proof/output array as an additional input, as well as an index into it. Furthermore, their return values are rewritten to return a modified index as an additional output.

The rationale behind using a simple array for keeping proofs is simplicity. For example, an alternate design would be to use a mapping $(f,x) \to (p,y)$, and while this would also improve performance if $f(x)$ were calculated multiple times, it has the disadvantage of requiring modelling a map fully in memory, as well as performing map accesses regularly. As they accesses were primarily written by hand in EVM assembly, a byte array is a far simpler structure. A possible objection is that the order of calls to verifiable functions, or indeed even the values passed to them, could be different between when the proofs were generated, and when the transaction is verified, due to some state of the blockchain which is used as input having changed. In this case, the proofs will fail to verify, as they do not receive the input matching the output and proof. This behaviour is a necessary side effect of utilising verifiable computation, and is discussed further in Section 4.7. However, it is important to note that this will never result in a transaction which should not verify being accepted, and only in transactions which were correct being rejected if they are no longer correct. As such, the integrity of the system is not endangered.

The actual verification itself is outsourced to a call to an external contract. This theoretically allows implementing verification fully on the active Ethereum blockchain, as the contract could simply be a port of Pinocchio's verification stage. However, as this would require porting a large part of Pinocchio entirely for the EVM, this was considered beyond the scope of the project. For this project it was instead decided to create a functional proof-of-concept, for which purpose a modified Ethereum blockchain, with a built-in function for handling verification calls, was used instead. This built-in calls the Pinocchio binary internally, something which would not be possible within an unmodified Ethereum context, as the EVM has no provisions for executing native code [39].

For the proof generation phase, it would appear at first that we would need a separate structure: We need no additional inputs to a function, but want the generated proof to be an additional output. Furthermore, instead of calling an external verification contract, we want to execute a Pinocchio's proof generation locally. A contract call makes little sense in this scenario, as the proof generation is an off-chain activity. Fortunately, proof generation is achievable without generating separate proof generation code, simply by changing the semantics of the above verification-oriented rewrite, as explained below.

The proof array input is obviously of no use for proof generation, as we are generating, not consuming proofs. While we could record generated proofs in it, this would require approximating the required space ahead of time, as the array cannot be dynamically resized. As proof generation is off-chain, we can take liberties with the computational model. To return the proofs from the computation, we therefore simply add a state

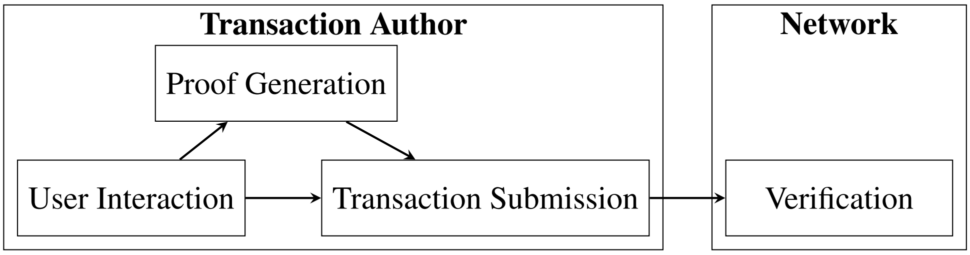


Figure 3.1: The high-level data flow of proof generation and verification in Ethereum

for generated proofs, which is initialized to empty at the start of proof generation, and appended to by each specific proof being generated. If we assumed the generic case, of an external contract being used for verification, it may be difficult to tell during execution what is the actual ‘verification’ stage, and what is ‘normal’ execution, as both are implemented in the EVM. However, in our proof-of-concept, verification corresponds to a specific built-in. As such, it is a simple matter to change the functionality of this built-in during proof generation. Instead of verifying a proof, the built-in will instead generate one, and append it to the proofs’ state.

The overall process is described by Figure 3.1. On the end of a transaction author, the actions taken would be to first execute the contract in proof-generation mode with a specified set of inputs, and then create a transaction with the same inputs, and the generated proof/output array. This transaction is then sent to the network as normal, which proceeds to verify it. The proof generation, and additional parameters and return values generated by the system can be hidden from a users view, although they must still be performed locally.

3.2 ABI/API Specifications

The functioning of the system outlined above requires new calling conventions between functions, as well as conventions for encoding various parameters, such as the outputs of arithmetic circuits, arithmetic circuits themselves, and the proof array. This section will formally specify the formats of each.

3.2.1 Function API/ABI Changes

As discussed above, each function which either uses verifiable computation itself, or calls functions that do, is given two additional input parameters, the proof/output array, and the index into it, as well as an additional output parameter, the new index into the proof/output array. In terms of Solidity source code, this is expressed as a rewrite adding parameters ‘`__proof_data`‘ of type `bytes`, and ‘`__proof_idx`’, of type `uint32`, as well as adding the return parameter ‘`__proof_idx_return`’ of type `uint32`.

For instance, the Solidity function in Listing 3.1 would be rewritten to have a signature such as the one in Listing 3.2.

Listing 3.1: A simple function before parameter rewriting.

```
function foo(int32 a, int32 b) returns (int32) {
    // ...
}
```

Listing 3.2: A simple function after parameter rewriting.

```
function foo(int32 a,
             int32 b,
             bytes __proof_data,
             uint32 __proof_idx)
         returns (int32, uint32 __proof_idx_return) {
    // ...
}
```

Solidity compilers further generate an 'ABI specification', a JSON specification of the signature of a function, used to specify the entry points of a contract [17]. The same applies to this JSON specification, the format is just slightly different. For the above snippets of code, the corresponding JSON specifications would be as seen in Listing 3.3, and Listing 3.4 respectively.

Listing 3.3: The JSON ABI specification of a simple function before parameter rewriting.

```
{
    "constant": false,
    "inputs": [
        {"name": "a", "type": "int32"},
        {"name": "b", "type": "int32"}
    ],
    "name": "foo",
    "outputs": [
        {"name": "", "type": "int32"}
    ],
    "payable": false,
    "type": "function"
}
```

Listing 3.4: The JSON ABI specification of a simple function after parameter rewriting.

```
{
    "constant": false,
    "inputs": [
        {"name": "a", "type": "int32"},
        {"name": "b", "type": "int32"},
        {"name": "__proof_data", "type": "bytes"},
        {"name": "__proof_idx", "type": "uint32"},
    ],
    "name": "foo",
    "outputs": [
        {"name": "", "type": "int32"},
        {"name": "__proof_idx_return", "type": "uint32"}
    ],
    "payable": false,
    "type": "function"
}
```

3.2.2 External Verify Call ABI

In Ethereum, external function calls take and return a fixed number of bytes as arguments, specified by the caller [39]. Effectively, an external function call is therefore passed a byte array for input, as well as (writable) one for output. The input to the verify function this project specifies consists of four byte arrays concatenated together: one encoding input parameters (see Section 3.2.4), one encoding the arithmetic circuit (see Section 3.2.3), one encoding the public key (as directly output by Pinocchio), and one encoding the proof/output tuple (see below). Each of these byte arrays consists of a 256-bit word denoting the length, followed by the corresponding number of bytes of data. The proof/output tuple is encoded similarly, as a byte array encoding the proof (as directly output by Pinocchio), followed by a byte array encoding the output parameters (see Section 3.2.5). Again, the byte arrays have the same length-value format.

The output of the verify function is a byte array encoding the output parameters again. The encoding is slightly different to that of the input parameters (see Section 3.2.5). The rationale for introducing this superfluous echo is that it makes modelling the proof generation case easier, as the call can directly return the computed outputs. Again, the byte array is preceded by a 256-bit word encoding the number of bytes following it. If the verification fails, zero will be returned for this word. Although technically this clashes with the correct return for a function without outputs, such a function is pointless to apply verifiable computation to, as it can inherently be optimised out.

3.2.3 Arithmetic Circuit Encoding

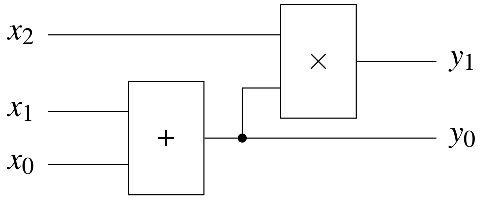
Arithmetic circuits are encoded as a number of input wires, a sequence of gates, and a list of output wires. See Listing 3.5 for the BNF description of the binary representation of arithmetic circuits used in this project. The BNF uses hexadecimal literals.

Listing 3.5: BNF of the binary encoding used for arithmetic circuits.

```
<circuit> ::= <inputs> <gate>* "80" <output>*
<inputs> ::= <16-bit-literal>
<gate> ::= <op1> <input1> <input1> | <op2> <input1> <input2> | <op3>
    <input2> <input2> | <const> <input2>
<op1> ::= <add1> | <mul1>
<op2> ::= <add2> | <mul2>
<op3> ::= <add3> | <mul3>
<output> ::= <16-bit-literal>
<input1> ::= <8-bit-literal>
<input2> ::= <16-bit-literal>
<add1> ::= "00"
<mul1> ::= "01"
<add2> ::= "10"
<mul2> ::= "11"
```

```
<add3> ::= "20"
<mul3> ::= "21"
<const> ::= "ff"
```

Specifically, this encoding starts by defining n wires as inputs to a circuit, labelled from 0 to $n-1$. Then, each gate defines a new wire as the output of the gate. This is either a constant 16-bit integer value, or an addition or multiplication of the values of two previous wires. Finally, a sequence of wires is specified as the 'outputs' of the circuit. An example encoding can be found below, in Figure 3.2. Note that while constants are here encoded as 16-bit integers, this does *not* mean that the circuit itself operates on 16-bit integers. This representation is agnostic toward the integer size used in evaluating the circuit.



Encoding	Meaning
`0003`	x_0, x_1, and x_2 are inputs.
`000001`	$x_3 = x_0 + x_1$
`010203`	$x_4 = x_2 \times x_3$
`80`	Break
`0003`	$y_0 = x_3$
`0004`	$y_1 = x_4$

Figure 3.2: A simple arithmetic circuit, and a breakdown of its encoding. The final encoding is `00030000010102038000030004`.

3.2.4 Input Parameters Encoding

Input parameters to an arithmetic circuit are encoded as a sequence of 32-bit integers, in the order of the wires they are to be assigned to within the circuit. They are packed into a single byte array, and specifying an array of input parameters which does not match the input wires of the corresponding circuit is an error.

3.2.5 Output Parameters Encoding

There are two separate positions in which output parameters are encoded: In the return value of the verify call, and in the proof/output tuple passed into the verify call. These are subtly different, not just as during proof generation they do not match, but because the output parameters passed in through the proof/output tuple need to precisely encode the output of the arithmetic circuit. In particular, as arithmetic circuits operate on 254-bit fields within Pinocchio [28], they must be able to capture this range of values. As a result, each output parameter in the proof/output tuple is encoded as a 256-bit word. Otherwise, the encoding is similar to that of input parameters, they are packed into a single byte array, in the order the output parameters appear in the corresponding circuit. For the output parameters returned from the verify function, the same applies, except that they are encoded as 32-bit integers instead of 256-bit. The additional bits are truncated.

3.2.6 Proof Array Encoding

The proof array itself is an array of proof/output parameter pairs. Each pair is a byte array prefixed with a 256-bit word denoting the length of the array in bytes. The proof is a byte array (as directly output by Pinocchio), also prefixed by a 256-bit word denoting its length. The input parameters are encoded as a byte array as specified in Section 3.2.4, again prefixed by 256-bits describing its length. Finally, the input/output pairs are simply packed together into the proof. Directly passing this into a rewritten Solidity function as a '`bytes`' value will have the entire proof array be prefixed by its length as a 256-bit word, while this is superfluous for this encoding, it can be ignored by using an initial index of 32.

3.3 Changes to the Solidity Language

In this report, the formal changes to the language and semantics of Solidity will be separated from the implementation details of said changes, with this section covering the former. As mentioned in Section 3.1, we need a way to annotate functions as using verifiable computation. Solidity already has a concept of function modifiers, one such built-in modifier being '`payable`', which is required in order for a function to accept a transaction giving funds. This design is to prevent users accidentally sending funds to a contract which has no means of managing or returning them. Similarly, we introduce a keyword '`verifiable`', which marks a function as using verifiable computation. For the changes to Solidity, we consider the generic design, in which an external contract handles verification. To define the contract used for this, the function modifier takes the contract address as an argument. For example, the toy addition contract shown in Listing 3.6 would be written as in Listing 3.7 to utilise verifiable computation. The argument given to the '`verifiable`' function modifier is not necessarily a constant, and can in fact be an arbitrarily complex Solidity expression. This is particularly convenient, as it allows the verifier contract to be set within a contract state variable. It is worth noting that this representation of the verifying address still functions with the built-in verification call used in this project, as Ethereum built-in functions are implemented as calls to a predefined address [14].

Listing 3.6: A toy addition contract written in standard Solidity.

```
contract Simple {
    function add(int32 a, int32 b) returns (int32) {
        return a + b;
    }
}
```

Listing 3.7: A toy addition contract written in modified Solidity, utilising verifiable computation.

```
contract Simple {
    address private verifier;

    function Simple(address _verifier) {
        verifier = _verifier;
    }

    function add(int32 a, int32 b) verifiable(verifier) returns (
       int32) {
        return a + b;
    }
}
```

As functions calling these ‘verifiable’ function need to have their signature rewritten, this could potentially break existing ABIs. To prevent this, only specifically marked functions will be rewritten to be proof-aware. The keyword ‘`verifying`’ is used in this case. Functions with this keyword will be rewritten to receive the proof array and proof index as inputs, as well as the modified proof index as an output. Functions without it will be treated as in vanilla Solidity. This comes with restrictions, however: A non-proof aware function *cannot* call a proof-aware function, as this would require the proof array to be created from nothing. Again, this keyword is used as a function modifier, although this time without any parameters. The ‘`verifiable`’ and ‘`verifying`’ keywords are mutually exclusive. An example showcasing which functions can call each other, and which cannot can be found in Listing 3.8.

Listing 3.8: A contract demonstrating all permitted kinds of function calls.

```
contract Calls {
    address private verifier;

    function Calls(address _verifier) {
        verifier = _verifier;
    }

    function fa(int32 a) verifiable(verifier) returns (int32) {
        // No function calls are allowed within verifiable
        // functions.
        return a + b;
    }

    function fb(int32 a) verifying returns (int32) {
        // All function calls are allowed within verifying
        // functions.
        if(a == 0) {
            // Okay, verifying -> verifiable function calls are
            // allowed.
            return fa(a, 5);
        } else if(a > 0) {
            // Okay, verifying -> verifying function calls are
            // allowed.
            return fb(a - 1);
        } else {
            // Okay, verifying -> normal function calls are allowed.
            return fc(a);
        }
    }

    function fc(int32 a) returns (int32) {
        // 'Normal' non-proof aware function can only call other
        // non-proof aware functions.
        if(a < 2) {
            return 1;
        } else {
            // Okay, normal -> normal function calls are allowed.
            return fc(a - 1, a - 2);
        }
    }
}
```

Finally, the code which is permitted to appear in verifiable functions is severely limited. This limitation is partially to prevent accessing external state from within verifiable functions, and partially to make implementing a functional compiler for them feasible within the scope of this project. The latter restrictions could be manually lifted one at a time, by adding support for more language features into the arithmetic circuit code generation.

To limit accessing of external state, calls to other functions are disallowed (theoretically, calls to other verifiable functions would be possible, this was not implemented and falls under the 'feasibility' restrictions). Furthermore, variable accesses are permitted only for local function variables, and attempts to use Ethereum-specific built-ins, such as retrieving the amount of remaining gas, or initiating a transfer, are not permitted.

On the feasibility side of things, types of variables are either '`int32`', '`bool`' or a fixed-size array of either. This decision was made as Pinocchio also operates solely on 32-bit integers and booleans [28]. The actually permitted operations are also severely limited: Variable declarations and assignments are permitted, as well as expressions using additions and multiplications. If-then-else blocks are supported, and fixed-iteration `for` loops of the form shown in Listing 3.9. Array accesses are permitted, however only with a constant index. It is worth noting, that as loops are unrolled, using a loop variable (or a constant expression involving it) as an index is effectively constant. Note that for technical reasons, the type of the loop index of for loops is `uint32`, instead of the typically only permitted `int32`. This is because only unsigned values can be used in Solidity as array indices.

Listing 3.9: The accepted format of `for` loops used in verifiable functions, where `i` may be any variable name, and `0` and `10` may be replaced with any integer literal.

```
for(uint32 i = 0; i < 10; i++) {
    // ...
}
```

While these restrictions on the Solidity language may seem impractically restrictive, they are expressive enough for complex arithmetic calculations, and are suitable for providing a proof-of-concept, as this project aims to do. The restriction to fixed-size inputs and outputs, as well as loops with a fixed number of iterations would make practical adoption particularly challenging, but is due to inherent restrictions of arithmetic circuits. For more detail on this, see Section 4.6.

3.4 Modifications of solc

Modifications to the solc compiler fall into four main categories: modifying the lexer/-parser, adding a pass enforcing restrictions on verifiable and verifying function, modifying code generation for verifying function, and adding code generation for verifiable functions.

3.4.1 Lexer and Parser Modifications

The lexer modifications were limited to adding two new keywords: verifiable and verifying, each corresponding to a new AST token of the same name. The parser was modified at the parsing of a function header to recognize the new tokens, similarly to how the payable keyword was already recognized. As the verifiable keyword requires an argument, a new parsing function was written for this.

In order to accommodate the new function modifiers, the AST for function definitions was adjusted, and given a boolean field for whether or not the function is verifying. Furthermore, they are given an AST expression denoting the verifier specified by the verifiable keyword, or null if the keyword was not present. While explicit checks enforce that verifying function cannot be called from ‘normal’ functions, a naive implementation of this might ignore indirect calls through local function variables. In order to prevent this, the types of functions themselves were also annotated with whether or not they are verifying/verifiable.

3.4.2 Restriction Checks

An additional pass was added to the compiler after identifier resolution and type checking to enforce the constraints laid out in Section 3.3. This pass implements an AST visitor which changes its behaviour depending on if it is visiting a ‘normal’ function, a ‘verifiable’ function, or a ‘verifying’ function. Verifying functions are without any restrictions, and the AST visitor ignores them entirely. ‘Normal’ functions cannot call verifiable or verifying functions, which the visitor enforces by checking the type of all identifiers used in the function. If any of these is a function type with either the verifying or verifiable flags set, the visitor produces an error message indicating that this function type is not allowed in the context.

Within a verifiable function, the checks enforce the language limitations laid out above for verifiable functions. In particular, the type of any declared variables is checked to be either int32, bool, or a fixed-size array of these. for loops are permitted only when adhering to the specific format described above, and the ‘verifier’ parameter, which specifies the external verifier contract is checked to be a type which can be cast to an address. Variable identifiers are ensured to refer to a variable declaration previously declared within a verifiable function. As these are local to the function, this ensures they refer only to function-local variables. Binary operands are restricted to addition and multiplication. Furthermore, the following constructs are forbidden outright:

- While loops
- Break and continue statements
- Tuple expressions
- Unary operations
- Throw statements
- Creation of new contracts

3.4.3 Verifying Function Code Generation

Code generation for verifying function was modified in three main ways: The input parameters were adjusted to add `__proof_data` and `__proof_idx`, function calls to verifying or verifiable functions were rewritten, and the return values of the function were adjusted to return the modified proof index. To do this, three variable declarations AST nodes were created, one for `__proof_data`, one for `__proof_idx`, and one for `__proof_idx_return`. These are given the types `bytes`, `uint32`, and `uint32` respectively. They are created whenever code generation first visits a verifying or verifiable function. `__proof_data` and `__proof_idx` are appended to the input parameters, and `__proof_idx_return` to the output parameters before the rest of code generation begins. As a result, the code generation effectively treats the function as having the modified signature.

A call to a verifiable or verifying function from within a verifying function is performed by pushing `__proof_data`, and `__proof_idx` onto the stack before calling. After the call completes, the top of the stack is the newly returned proof index, which is moved into `__proof_idx` once again. Before returning, the value of `__proof_idx` is copied into `__proof_idx_return`. As return variables are declared ahead of time, this by itself is sufficient to guarantee a correct return value.

3.4.4 Verifiable Function Code Generation

Verifiable functions are greatly different in code generation to any other Solidity function, as the main function body is compiled to an arithmetic circuit instead of EVM bytecode. The same code adding in the additional parameters is reused from verifiable functions. Typically, Solidity preallocates space for all local variables on the stack. This is removed for verifiable functions, as local variables will be compiled into the arithmetic circuit as well. The code generation then begins by generating the arithmetic circuit corresponding to the function body.

The arithmetic circuit is generated with an AST visitor on the function body. The current state of the generation consists of the already generated part of the circuit, and a mapping from AST nodes to the values they represent in the circuit. These values can either be an integer constant, a single wire in the circuit, or an array of wires. For example, the AST node for a variable of type `int32[4]` would have a value of an array of four wires. Variable states are kept by reassigning the values mapped to the variable's declaration. An assignment to a variable therefore sets the variable's value to the value of the right-hand side expression, while an identifier has the value corresponding to the variable it refers to.

This allows variable assignments and accesses to function purely as aliases for wires. Even array accesses work this way: The *n*th entry in an array has the value of the *n*th entry in the array's value, which must be an array of wires. Writing to an arrays *n*th index corresponds to changing the wire the *n*th index refers to. Of course, if all the program could be defined as aliases, there would not be a program to begin with. In particular addition and multiplication expression map directly into addition and multiplication gates in an arithmetic circuit. One of these expressions adds the corresponding gate to the circuit, with the two input wires that correspond to the expressions on either side of the operand. The entire expression is then mapped to the output wire of the new gate in the circuit.

This covers simple variable assignments and expressions, however does not deal with control flow structures. While these are limited to begin with, limited support for conditionals and `for` loops does exist. Conditional if statements are implemented by first evaluating the condition. This boolean condition should evaluate either to a 1, indicating true, or 0, indicating false. Then, both branches of the conditional are evaluated separately, and the variable states between the two branches are compared. Where the variables differ, taking c as the conditional, and s_c and $s_{\neg c}$ being the states after the 'then' and 'else' branches respectively, the new, unified state s' is calculated as follows:

$$s' = (s_c c) + (s_{\neg c}(1 - c))$$

Computing $1 - c$ is done by multiplying c with the twos complement representation for -1. `for` loops are easier to implement with the restrictions that have already been imposed on them. They are implemented by simply visiting the body of the loop however many times are specified, each time assigning the loop index the corresponding constant value.

Once the circuit has been generated, the compiler invokes Pinocchio in order to generate a corresponding public key. This is done by invoking a small tool `ethvc-genkey`, originally created within the scope of the parity modifications, and described in more detail in Section 3.5.3. Once all of these are present, all values needed to call the external verify function are present: The input parameters, the arithmetic circuit, the public key, and the proof/output pair in the proof/output array. These need to be encoded into memory in sequence, in the format described in Section 3.2.2.

To do this, a mixture of hand-written and automatically generated EVM assembly is used, first packing input parameters into a memory byte array, then writing the first the generated circuit, and next the generated public key, both as byte arrays into memory. Finally, the next tuple of the proof/output array is copied into memory. The external call is performed, and the return value is checked. If the first word of the return value is zero, an error condition is raised. If not, the return values are unpacked, and assigned to the functions return parameters. Finally, the proof index is advanced, and the function returns.

3.5 Modifications of Parity

The modifications to `solc` provide a means for automatically rewriting contracts into a framework which can be used to supply and verify proofs against arithmetic circuits. It does not, however, implement the verification of these proofs themselves, or allow for the easy generation of said proofs. While the former could be implemented on-chain with a contract replicating Pinocchio's verification stage, as discussed before, this goes beyond the scope of this project, and instead a built-in verify function was added.

In order to add this, as well as provide a means for proof generation, the Parity client for Ethereum was modified. Parity was chosen for this, as from prior experience it appeared to be the most elegant Ethereum client implemented, as well as being cleanly coded in the Rust programming language [22], which I am strongly familiar with.

3.5.1 Adding a Verify Built-in

Adding a built-in function to parity is actually two separate matters, and not a single issue. It consists of adding the function itself, a natively implemented function mapping byte arrays to byte arrays, and specifying a blockchain which utilises this function. For instance, Ethereum's `sha256` built-in's two parts are implemented as a) a call to a low-level cryptography library, and b) an entry into Ethereum's blockchain specification file, linking the address `0x0000000000000000000000000000000000000002` to the `sha256` built-in, and specifying the costs of calling it. The former is easily implemented, simply linking directly to the Pinocchio abstraction library, described in Section 3.5.3.

For the latter, a new blockchain specification was created, linking the address `0x0000000000000000000000000000000000000005` to the verification built-in. To simplify testing, the blockchain specification used an authority-based engine, meaning that the creation of new blocks was based on an authoritative signature, instead of proof of work or other methods. This effectively allows immediate and free mining of any transactions, as no effort must be expended to create a block, simplifying testing.

3.5.2 Adding a Prove API Call

A method for proof generation is needed for any practical use of the system. While it is possible to generate proofs manually, this defeats to purpose of automatic rewriting of contracts. First, a means for proof generation is added, and then it is linked to a public API for the Parity client. The parity client follows Ethereum standards for a network and/or UNIX socket based JSON-RPC API [16, 36]. Proof generation is implemented as by switching the functionality of the verify built-in to instead generate a proof and compute the correct output, ignoring the input proof/output tuple. The Pinocchio abstraction library holds its own buffer for generated proofs, which is consumed once the transaction has been fully executed. Then, the generated proof is returned via the API. Specifically, the `eth_call` JSON-RPC method [16] is mimicked, which evaluates a transaction without sending it to the network. The new JSON-RPC method is called `eth_prove`, and has the same signature as `eth_call`, except that it returns the generated proof instead of the return value of the transaction.

3.5.3 Adding a Pinocchio Abstraction Layer

The Pinocchio abstraction layer deals with two things primarily: Converting between this project's binary representations and Pinocchio's plain text representations, and executing Pinocchio itself. Specifically, Pinocchio uses different encodings for the arithmetic circuits, as well as circuit input and output values. These are designed to be plain text input files in Pinocchio. However, this representation is unsuitable for Ethereum, as memory is highly limited. As a result the abstraction layer is largely boilerplate conversion code.

Furthermore, Pinocchio is only available as a Windows binary, and cannot be easily recompiled for other platforms as it links against proprietary Microsoft cryptography libraries [27]. As a result, it is not possible to directly link against Pinocchio, and it must instead be run as a command line program. As Pinocchio is only available as a Windows program, and this project was developed and tested under Linux, Pinocchio is invoked through Wine [2]. In practice, the flow of the abstraction layer is as follows:

1. Create a temporary directory
2. Transcode the inputs into files in the directory.
3. Execute the Pinocchio executable
4. Transcode the outputs from the directory into a binary representation.

The abstraction layer implements three main operations: Verification, proof generation, and key generation. Verification and proof generation both adhere to the verify ABI, described in Section 3.2.2, for both inputs and outputs. The proof generated by proof generation is appended to an array, which can separately be consumed. Key generation takes only the arithmetic circuit as an input, and produces the public key as its output. All of these operations are also implemented as small, external programs.

3.6 Extending the `geth` JavaScript Console

In order to utilise the modified API, a front end for Ethereum had to be modified. While Parity comes with its own web front end, this is a complex system in itself. The front end is not the focus of this project, so instead an additional program, `geth` [1] was modified. `geth` is an Ethereum client just as parity is, however this is not relevant for the project. `geth` also implements a JavaScript console for interacting with the

JSON-RPC API, and can interact with Parity in this way [37]. The modifications to `geth` lie solely in adding API calls for proving to the JavaScript API.

Chapter 4

Achievements and Limitations

This chapter will cover the basics of using the modified Ethereum ecosystem, from setup to usage. It will then further delve into both theoretical and practical limitations of the system as it currently stands. It will provide the reasons for these limitations, evaluate their seriousness, and suggest how they could potentially be mitigated.

4.1 Demonstration of Functionality

In this section the usage of the system will be demonstrated through an example contract, although the general principal applies to any desired contract. We will specifically consider a contract a simple matrix multiplication contract, as written in Listing 4.1.

Listing 4.1: A matrix multiplication contract.

```
contract MatrixMultiplication {
    address private verifier;
    int32[9] state;

    function getState(uint idx) public constant returns (int32) {
        return state[idx];
    }

    function MatrixMultiplication(address _verifier) {
        verifier = _verifier;
        for(uint8 i = 0; i < 9; i++) {
            state[i] = i;
        }
    }

    function multiply(int32[9] a, int32[9] b) verifiable(verifier)
       returns (int32[9] c) {
        // Row index
        for(uint32 i = 0; i < 3; i++) {
            // Column index
            for(uint32 j = 0; j < 3; j++) {
                c[3*i + j] = 0;
                for(uint32 k = 0; k < 3; k++) {
                    c[3*i + j] = c[3*i + j] +
                                 a[3*i + k] * b[3*k + j];
                }
            }
        }
        return c;
    }

    function square() verifying {
        state = multiply(state, state);
    }
}
```

In order to use this contract, it must be compiled using '`solc --bin --abi <file >`'. This will produce the contract ABI, as well as the compiled binary. Next, the parity client needs to be started, using '`parity --config node0.toml --geth --force-ui`', and connected through both via the web interface and the `geth` console. The former can be accessed at `http://localhost:8080`, and is needed to authorise transactions. The latter can be accessed by executing '`geth attach`' in a new terminal. From the `geth` console, our new contract can be deployed and used with the following steps:

1. Enter '`eth.defaultAccount = eth.accounts[0]`', to specify which account to use to pay for transactions.
2. Enter '`var contract = eth.contract(<abi>)`', where `<abi>` is as returned by `solc`.
3. Enter '`var instance = contract.new("0x0000000000000000000000000000000000000005", {data: "0x<binary>", gas: 5000000})`', where `<verifier>` is the verifier address, and `<binary>` is the contract binary, as returned by `solc`.
4. Confirm the transaction from the web interface by entering the account password '`user`'.

The contract will now be (shortly) deployed, and can be accessed. The contract we are considering, as introduced in Listing 4.1 has rather limited interaction, you can get elements of the state matrix, or square it. The former can be simply done by executing '`instance.getState(<n>)`' from the `geth` console, where `<n>` is the element of the matrix we wish to access. Squaring it actually utilises verifiable computation, so we must generate a proof before we can create the actual transaction. We can obtain such a proof with the following `geth` console query:

```
instance.square.prove("0x000000000000000000000000000000000000000000
0000000000000000000000", 32)
```

In this query, a number of things may appear strange. We are passing in a proof argument, but this is not empty, instead it consists of 64 zeros. Further, we start at proof offset 32. The reasons for these are implementation details. The former is due to each time the external verify function is called, the proof/output tuple passed to it is copied in memory. As the astute reader may recall, the proof/output tuple is prefixed by its length, which is how the program knows how much data to copy. This will be done during proving as well, as it occurs outside of the verification function. As a result, if no proof were supplied, the program would attempt to copy as many bytes as specified by an uninitialised memory segment. Unless this segment is still zero at the time of execution, this will almost certainly fail. The offset of 32 bytes is due to byte 'bytes' type, which the proof is supplied as, being prefixed with a 256-bit word describing its length as well. As a result, all indexing needs to be offset by 256 bits, or 32 bytes. Once generated, the proof can be used to create a final transaction as follows:

```
instance.square(instance.square.prove("0x00000000000000000000000000
00000000000000000000000000000000000000",32), 32)
```

This will first compute the proof, and then create a transaction containing this proof as an argument. In our case, the function 'square' takes no arguments, a function which does so would have these supplied before the proof in both the proving, and final transaction creation call.

4.2 Limitation by the Verification Public Key Size

Initial designs assumed that a single verification public key could be shared across all arithmetic circuits. This would simplify the model, by allowing the public verification key to be essentially constant for all considerations except the generation of this key. However, this assumption was mistaken, as Pinocchio requires each arithmetic circuit to have its own verification public key. This increases the impact of the trust issue discussed later in Section 4.3. However, it brings another issue: The verification public key must be included as part of the contract, and loaded into memory every time a verifiable function is called. This seems reasonable in principal, however the size of these keys, as generated by Pinocchio, grows with the size of the circuits, and is within the kilobytes for even trivial circuits. The handling of the verification keys associated with large functions carries a massive overhead, and deploying a contract with too complex circuits may result in it being too large to be accepted by the network. For practical adoption, a method of verifiable computation which does not see such a massive increase in storage and memory usage is undoubtedly necessary. Without further work it is not clear whether Pinocchio can be modified to provide this, or whether a new approach to verifiable computation is required.

4.3 Trust Requirement Problem

The public verification key is fundamentally a cryptographic public key. As such, the party generating the key can also create a corresponding private key. This could, in turn, be used to 'prove' anything as correct, as it would verify correctly with the public key. This problem is similar to that in zero-knowledge blockchains, such as ZeroCoin [24] and ZeroCash [33], which rely on a trusted setup. This requirement undermines one of the core principals behind blockchains, however, which the elimination of a central trusted party [25].

This is further compounded by the key generation in Pinocchio being per circuit, meaning that even a means for generating a single key trustlessly would be insufficient in this case (while it would be sufficient for ZeroCoin and ZeroCash). It is possible that this limitation is not intrinsic, but only an implementation detail of Pinocchio, in which case the problem could be reduced to generating a single key trustlessly. This could potentially be done by ensuring transparency physically (i.e. have the key be generated publicly, on a machine verified by multiple parties). Further, the trustless generation of only a public key through cryptographic means is not necessarily impossible, but is an open question of research that goes far beyond the scope of this project.

4.4 State of Research for Verifiable Computation

As mentioned previously, Pinocchio, while achieving verification times lower than execution times in some cases, does not provide benefits in all situations, and the speedups that do exist are not sufficiently great to justify the other limitations [28]. That said, verifiable computation is an active field of research in which much progress has been made in the past few years, and it is likely this concern will become less relevant as the field matures. Furthermore, it is possible that for this problem smart contracts have a unique benefit: Typically performance will be measured against native code execution for verifiable computation, however in smart contracts the baseline is the execution of a virtual machine instead of native code. By comparison, the native verification could appear much faster.

4.5 Black-Box Usage of Pinocchio

Throughout this project, Pinocchio was treated as a black box. While this has significant conceptual advantages, as an in depth understanding of it is not required, it limits the flexibility of its use. Several of the problems listed above could potentially be fully or partially solved by modifying the underlying functionality of Pinocchio, and tailoring it toward use in Ethereum. In particular, the size of public keys, and the need for individual public keys for each arithmetic circuit could potentially be limited. Furthermore, implementing a verification contract, rather than requiring verification being a built-in function, would allow verifiable computation to be used on the active Ethereum blockchain, without the requirement for forking. Even just running Pinocchio more efficiently, by not needing to rely on Wine and format conversions, would likely be of significant benefit in a low-resource environment such as the EVM.

4.6 Verifiable Function Restrictions

The restrictions placed on verifiable functions in the extended Solidity language are very harsh, and are likely to hinder a typical programming workflow. To a large part, these restrictions are simply a matter of implementing missing features, such as types other than `int32` and `bool`, more arithmetic operations, tuples, `break` and `continue`, or even calling other verifiable functions (through inlining). However, some restrictions are not so easily lifted, such as loops with an unbounded number of iterations, recursion, and variable-size inputs. These are more fundamental restrictions of verifiable computation, at least as it is approached in Pinocchio. A followup paper by Microsoft Research describes an approach for splitting a program into parts, each of which is compiled into an arithmetic circuit, instead of generating a single circuit for the entire program [6]. This kind of approach could be used to allow more rich expressions in verifiable functions, leaving the compiler with the task of splitting the program into circuits.

4.7 Invalidation of Proofs through State Change

As proof generation occurs off-chain, and before the miner, and other nodes in the network verify the transaction, it is possible for the blockchain state to have changed in the meantime. For instance, consider the contract in Listing 4.2, where a verifiable function depends on a contract's state. Initially, the state vector is $\vec{0}$. Supposing both Alice and Bob create a transaction, adding the vectors $\vec{a}$ and $\vec{b}$, respectively. Alice computes a proof that $\vec{0}+\vec{a}=\vec{a}$, while Bob computes a proof that $\vec{0}+\vec{b}=\vec{b}$. If Alice's transaction is mined first, the state vector will have changed to $\vec{a}$. Bob's transaction will then be rejected, as the verification will attempt to verify that $\vec{a}+\vec{b}=\vec{b}$, which will fail (both as it is empirically false, and because it does not match the proof provided).

As transaction fees will be consumed for the failed transaction regardless, the effective cost could increase under certain circumstances when using verifiable computation rather than decrease. In particular, verifiable computation is unlikely to be well suited for contracts which see very frequent changes in state. Furthermore, this could potentially be abused to perform denial of service attacks, by consistently making small modifications to a state in order to prevent legitimate transactions from going through.

This being said, at the time of writing, no contract sees sufficient traffic for this happening accidentally being a serious concern, with many blocks containing no transactions whatsoever. A targeted denial of service attack, on the other hand, can not be immediately dismissed, and it is not clear how to easily prevent it in the general case. Despite this, many applications can afford to restrict the ability to change state sufficiently that such an attack is not easily possible.

Listing 4.2: A toy contract for incrementing a state vector.

```
contract VectorAdd {
    address private verifier;
    int32[16] state;

    function VectorAdd(address _verifier) {
        verifier = _verifier;
    }

    function vector_add(int32[16] a, int32[16] b) verifiable(
       verifier) returns (int32[16] c) {
        for(uint32 i = 0; i < 16; i++) {
            c[i] = a[i] + b[i];
        }
        return c;
    }

    function increment(int32[16] diff) verifying {
        state = vector_add(state, diff);
    }
}
```

Chapter 5

Avenues Explored

Throughout this final year project, multiple avenues were pursued which did not directly make it into the final project. This chapter will briefly sketch them for the sake of completeness and posterity, and justify why these avenues were not followed up, are why they were changed.

5.1 Privacy in Ethereum

Initially during the project the we explored various paths before settling on verifiable computation and smart contracts as the most interesting avenue. Most notably, performing a privacy analysis on Ethereum similar to previous work on denonymising BitCoin was considered. There are broadly two separate approaches to denonymising BitCoin transactions: Extracting information from the underlying network protocol, and correlating addresses on the blockchain. The former is based on the principal that the node from which you will first hear about a transaction is likely the author of this transaction [4]. The latter traces the movement of funds across the network, and attempts to correlate addresses belonging to the same entity [31]. For example, since funds must be spent in their entirety in BitCoin, if Alice wishes to send Bob one BitCoin from her account, and she has a balance of five BitCoins, she does this by creating a new address and sending one BitCoin to Bob, and four to her new address. This pattern is very common for BitCoin transactions, and transferring the remaining amount can indicate that the recipient is the same user.

This direction was ultimately dropped for multiple reasons. Firstly, Ethereum is a very young technology, and it is unclear if an interesting information could be gained through denonymisation. Secondly, transaction linking as in BitCoin is of little interest for Ethereum, as Ethereum encourages the reuse of addresses, while BitCoin does not [3]. This may seem like a minor detail, but as the majority of users interact via clients which enforce these recommendations, most transactions in Ethereum are already linked.

5.2 Previous High-Level Design

A previous system design was made under the assumption that proof generation had to occur during mining. While all components would function roughly the same, the miner would be responsible for generating the proof instead of the transaction author. This has the advantage of avoiding the problem that state may have changed, as described in Section 4.7 entirely, however has numerous downsides. Firstly, as proof generation is computationally more expensive, it makes designing incentives for both miners and users to utilise verifiable computation difficult. Miners are unlikely to accept more difficult transactions unless paid more fees, and users are unlikely to pay more fees for the same functionality. Furthermore, the system would require miners to handle transactions differently than they currently do, requiring a soft fork in Ethereum. The new design, with the transaction author generating the proof, can be applied to the active blockchain, provided a verifier contract is implemented.

5.3 Verify ABI Modifications

The ABI for the external verification function, as described in Section 3.2.2, underwent multiple design iterations before arriving at its current state. Each of these modifications required significant changes to the EVM assembly generated to call this function, as the definition of the function it was invoking changed. This section will describe the modifications to the ABI in chronological order, and motivate the reasoning behind each design.

5.3.1 Function Signature Hash and Output Returning

Function ABIs generated by Solidity are prefixed by a short four byte hash of the function name and signature [15]. This is used to identify the function within the contract to call. The first draft of the verify ABI matches exactly the ABI for a Solidity function as in Listing 5.1. However, it became apparent that for proof generation it would be useful to be able to pass a dummy circuit output value in, and have the verify function supply to correct output value. It was clear at this time that verification would be implemented with a built-in instead of a Solidity contract. Furthermore, the return value of the function had to be variable length (as different circuits have different output lengths), which is disallowed by Solidity. As a result, the 4 byte hash was removed, and the signature changed to something similar to, but not equal to (due to the missing hash) the ABI of the Solidity function in Listing 5.2.

Listing 5.1: The signature of a verification function, in its first iteration.

```
function verify(bytes input, bytes circuit, bytes proof) {
    // ...
}
```

Listing 5.2: The signature of a verification function, in its second iteration (ignoring the preceeding hash).

```
function verify(bytes input, bytes circuit, bytes proof)
        returns (bytes output) {
    // ...
}
```

5.3.2 Verification Key

At the time of the second ABI iteration, I was mistakenly assuming that the same verification public key could be used for any arithmetic circuit, and a fixed verification public key would be used for the entire protocol. As is discussed in Section 4.2, this assumption was mistaken, as Pinocchio requires a separate verification public key for each circuit. As a result, this key had to be added as an input to the verification function. While this is fully specified in Section 3.2.2, Listing 5.3 gives a rough view of what signature a function implementing it would look like, similar to the examples above.

Listing 5.3: The signature of a verification function, in its final iteration (ignoring the preceding hash).

```
function verify(bytes input,
                bytes circuit,
                bytes pubkey,
                bytes proof)
        returns (bytes output) {
    // ...
}
```

Chapter 6

Conclusion

This project has presented a methodology for integrating verifiable computation with smart contracts, and provided a limited, but functional proof-of-concept implementation. When a transaction is created, instead of directly broadcasting it to the network, the evaluation of some functions is first performed and proven locally, and the proofs and results of these evaluations are attached to the transaction. By compiling these functions into calls to a verification function instead of their typical code, the network will never have to evaluate the original function, and only verify the supplied proof.

To achieve this primary goal, specifically the Ethereum blockchain [39] was integrated with the Pinocchio [28] verifiable computation prover/verifier. The Solidity language [17] was extended to allow the programmer to specify which functions should utilise verifiable computation and which should not. The `solc` compiler [18] was also extended, compiling these 'verifiable' functions to arithmetic circuits, and generating calls to an external verification function. Furthermore, all required functions were rewritten to support passing in proof and results data, ensuring a clear external interface is available for supplying proofs, while eliminating internal clutter.

The Parity Ethereum client [36], as well as the `geth` JavaScript console [1] were extended to allow the generation of proofs for a transaction with a JavaScript API call. Furthermore, a modified blockchain based on Ethereum was specified with a built-in verification function, and implemented in the Parity client. The result is a functional client and front-end allowing off-chain proof generation and on-chain verification [21].

This project was, to my knowledge, the first functional implementation utilising verifiable computation within smart contracts. The application of verifiable computation to smart contracts is very appealing due to it leading directly to more complex contracts becoming feasible. The limitations of this proof-of-concept system make it clear, however, that this technology requires further work for widespread usage. In particular, verifiable computation as developed for the standard use cases, such as cloud computing, is not directly suited for smart contracts due to storage and memory constraints. Future work would ideally tailor an implementation of verifiable computation specifically for smart contracts, and implement the verification stage as a contract itself. Furthermore, the problem of generating a trusted public key in a trustless environment is crucial for the security of contracts utilising verifiable computation. A potential solution to this may be running a secure multi-party computation between multiple nodes in the network to generate new public keys, ensuring no party has all the information required to create the corresponding private key. Although this may appear straightforward, a naive implementation would be vulnerable to Sybil attacks.

Appendix A

Installation Instructions

As this project makes modifications to multiple large pieces of software, there are a number of dependencies that need to be installed to build it. The building process has been tested only under GNU/Linux; Microsoft Windows is currently not supported. Specifically, the following software is required: git, make, cmake, pkg-config, rustc, cargo, wine, boost, npm, autoconf, automake, gcc, g++, libtool, and go.

Once these are installed, it is a simple matter of cloning the repository [21], running `make`, and adding the `bin` directory to your `PATH`. The commands required to setup under a fresh ArchLinux installation can be found in Listing A.1.

Listing A.1: Installation under ArchLinux.

```
$ sudo pacman -S git make base-devel pkg-config rustup wine cmake
    boost npm go
$ rustup default stable
$ git clone --recursive https://git.tkerber.org/uni/proj.git
$ cd proj
$ make
$ export PATH="$(pwd)/bin:$PATH"
```

Bibliography

[1] Go-Ethereum authors. Go Ethereum. https://geth.ethereum.org, 2013–2017.

[2] Wine authors. Winehq – run Windows applications on Linux, BSD, Solaris and macOS. https://www.winehq.org, 1993–2017.

[3] Jaume Barcelo. User privacy in the public Bitcoin blockchain. http://www.dtic.upf.edu/~jbarcelo/papers/20140704_User_Privacy_in_the_Public_Bitcoin_Blockchain/paper.pdf, 2014.

[4] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in bitcoin p2p network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 15–29. ACM, 2014.

[5] CoinMarketCap. Cryptocurrency market capitalizations. https://coinmarketcap.com, 2017.

[6] Craig Costello, Cdric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Institute of Electrical and Electronics Engineers, May 2015.

[7] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[8] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2:6–10, 2016.

[9] Christian Decker and Roger Wattenhofer. Bitcoin transaction malleability and MtGox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.

[10] John R Douceur. The Sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[11] The Economist. The DAO of accrue. http://www.economist.com/news/finance-and-economics/21699159-new-automated-investment-fund-has-attracted-stacks-digital-money-dao, May 2016.

[12] Ethereum Foundation. Create a cryptocurrency contract in Ethereum. `https://ethereum.org/token`, 2016–2017.

[13] Ethereum Foundation. Design rationale. `https://github.com/ethereum/wiki/wiki/Design-Rationale`, 2016–2017.

[14] Ethereum Foundation. Ethereum chain spec format. `https://github.com/ethereum/wiki/wiki/Ethereum-Chain-Spec-Format`, 2016–2017.

[15] Ethereum Foundation. Ethereum contract ABI. `https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI`, 2016–2017.

[16] Ethereum Foundation. JSON-RPC API. `https://github.com/ethereum/wiki/wiki/JSON-RPC`, 2016–2017.

[17] Ethereum Foundation. Solidity 0.4.11 documentation. `http://solidity.readthedocs.io/en/latest/`, 2016–2017.

[18] Ethereum Foundation. The Solidity contract-oriented programming language. `https://github.com/ethereum/solidity`, 2016–2017.

[19] Forecast Foundation. Decentralized prediction markets – Augur Project. `https://augur.net`, 2015–2017.

[20] Ghassan Karame, Elli Androulaki, and Srdjan Capkun. Two Bitcoins at the price of one? double-spending attacks on fast payments in Bitcoin. *IACR Cryptology ePrint Archive*, 2012(248), 2012.

[21] Thomas Kerber. Verifiable computation on Ethereum. `https://git.tkerber.org/uni/proj`, 2016–2017.

[22] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[23] Gennady Medvinsky and Clifford Neuman. Netcash: A design for practical electronic currency on the internet. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 102–106. ACM, 1993.

[24] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 397–411. IEEE, 2013.

[25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[26] Satoshi Nakamoto and contributors. Bitcoin script reference implementation. `https://github.com/bitcoin/bitcoin/blob/master/src/script/interpreter.cpp`, 2009–2017.

[27] Bryan Parno. Pinocchio readme. `http://vc.codeplex.com/SourceControl/latest#pinocchio/README`, 2013–2016.

[28] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 238–252. IEEE, 2013.

[29] Bryan Parno, Mariana Raykova, and Vinod Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *Theory of Cryptography Conference*, pages 422–439. Springer, 2012.

[30] Doug Petkanics and Eric Tang. Decentralised auctions for on-chain asssets. `http://auctionhouse.dappbench.com/`, 2016–2017.

[31] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[32] Blockchain Luxembourg S.A. Bitcoins in circulation. `https://blockchain.info/charts/total-bitcoins`, 2017.

[33] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.

[34] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 19–30. ACM, 2010.

[35] Matthew Tan, Wee Chuan, and Jann Yik. Etherium gas price chart. `https://etherscan.io/chart/gasprice`, 2017.

[36] Parity Technologies. Parity. `https://parity.io/parity.html`, 2016–2017.

[37] Parity Technologies. Parity – basic usage. `https://github.com/paritytech/parity/wiki/Basic-Usage`, 2016–2017.

[38] Jeffrey Wilcke. Homestead release. `https://blog.ethereum.org/2016/02/29/homestead-release/`, 2016.

[39] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[40] XE. Convert XBT/GBP. `http://www.xe.com/currencyconverter/convert/?Amount=1&From=XBT&To=GBP`, 2017.