CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

MultiCall: an interpreter implemented on the Ethereum Virtual Machine

Master's thesis in Computer science and engineering

William Hughes

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

MultiCall: an interpreter implemented on the Ethereum Virtual Machine

William Hughes

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

MultiCall: an interpreter implemented on the Ethereum Virtual Machine

William Hughes

Supervisor: Alejandro Russo, Department of Computer Science and Engineering
Examiner: Gerardo Schneider, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

MultiCall: an interpreter implemented on the Ethereum Virtual Machine

William Hughes
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Cryptocurrency protocols such as Bitcoin and Ethereum provide distributed ledgers, decentralized databases which can be used to make censorship-proof payments without a trusted intermediary. The computation performed to update the ledger is performed redundantly across thousands of servers and is therefore expensive. Ethereum additionally supports Turing complete programs stored in the ledger; they can be used to retrofit the protocol with new capabilities and optimizations. The following thesis presents the MultiCall interpreter, a smart contract implemented on the Ethereum blockchain which is capable of significantly reducing the cost of payments by emulating multiple transactions with a single invocation of the interpreter.

Keywords: blockchain, ethereum, smart contracts

Acknowledgements

I would like to thank my supervisor Alejandro Russo for his support and valuable guidance, and my opponent Lamiya for her extraordinary patience.

William Hughes, Gothenburg, March 2020

Contents

1
Introduction

In recent years, a new kind of distributed data structure has risen to prominence. Termed a distributed ledger, it allows multiple mutually suspicious parties to maintain and access a consistent database without a central provider capable of manipulating, shutting down or controlling access to the database. By replicating the database among many nodes, it also becomes difficult for an outside adversary to shut it down. This property is known as censorship resistance. Using an open distributed ledger, anyone with an internet connection and sufficient computational resources may participate in maintaining, securing and updating the database. As the ledger is maintained redundantly, it can remain live and consistent as individual maintainers enter and exit.

1.1 Blockchains

The main implementation technology of distributed ledgers is the *blockchain*, a distributed data structure to which users can append *blocks* of *transactions* in a linked list structure. The cryptographic properties of blockchain protocols make it expensive to create multiple contradictory chains, and for subsets of maintainers to prevent a valid transaction from being included in the chain; this allows blockchains to serve as a tamper- and censorship-proof distributed ledger. Blockchains are primarily used to record transactions in currency-like tokens. Such tokens are termed cryptocurrencies when recorded on a distributed ledger.

Transactions are a unit of interaction with the blockchain, typically signed by the sender and specifying a recipient and amount of cryptocurrency to send.

Blockchain protocols typically provide special support for a particular cryptocurrency called a *native cryptocurrency*. The native cryptocurrency is then the unit of account used to pay transaction fees to chain maintainers, and is issued to maintainers by the protocol in return for appending new blocks to the chain. Chain maintainers that add new blocks to the chain are known as *miners*, and the process of receiving new cryptocurrency in return for appending new blocks is termed *mining*. A server that keeps the full state of the ledger available for the purpose of verifying blocks is called a *full node*; a more lightweight program which requests smaller cryptographic proofs about subsets of the state when needed is known as a *light client*. Live ledger state is distinct from the full transaction history; new transactions are interpreted to update the virtual state of the ledger, whereas the blockchain is a record of all updates to that state. Servers which store the entire chain (which may be many gigabytes in size) are termed *archival nodes*. Not all cryp-

tocurrencies are native: the blockchain Ethereum hosts many currencies created by users.

1.1.1 History

The first blockchain to be published was the Bitcoin blockchain [25], released by the pseudonymous Satoshi Nakamoto in 2009. Its native cryptocurrency, Bitcoin (BTC), rapidly appreciated in value as the protocol made mining new BTC more expensive and the cryptocurrency gained adoption. Bitcoin has been used for anonymous and illicit payments, but also to avoid hyperinflation in countries with high inflation and capital controls such as Venezuela[1]. A speculative frenzy emerged, sending Bitcoin's price rising from cents in 2010 to hundreds of dollars by 2014[2]. Exchanges on which to trade and store cryptocurrency emerged, with a number of high-profile failures[3]. Inspired by the technological potential and success of Bitcoin, many new blockchains with their own native cryptocurrencies were developed, such as DogeCoin [24] and LiteCoin [2]. While many blockchains were derivative, some offered exciting new capabilities.

1.1.2 The Ethereum blockchain

In 2015, a new type of distributed ledger was released by Vitalik Buterin et al.: the Turing-complete blockchain Ethereum [31]. A Turing-complete distributed ledger allows users to upload and invoke Turing-complete programs within the state of the ledger. A program which runs within the state of a distributed ledger is termed a smart contract, regardless of whether it is Turing-complete. Bitcoin provides support for smart contracts, but its scripting language is non-Turing-complete and thus quite limited. By virtue of their Turing-completeness, Ethereum smart contracts are much more expressive. The state of the Ethereum ledger consists of a mapping between 160-bit addresses and balances of ether, the native cryptocurrency. Not coincidentally, the public keys of the public key cryptography scheme used to sign Ethereum transactions are also 160-bit values. When an address is equal to a public key, the holder of the corresponding private key may send transactions from that address by submitting transactions signed by their private key. Such addresses are known as externally owned accounts or EOAs. An Ethereum address may also map to the program text of a smart contract. Ethereum transactions may either send an amount of ether and a bytestring message called calldata to an address, or it may create a new smart contract at a pseudorandom address. When a transaction is sent to an EOA, its balance of ether is simply incremented. When a transaction is sent to a smart contract, the smart contract's program is run; the calldata is provided as an input to the program. The interpreter for Ethereum smart contracts is called the Ethereum Virtual Machine or EVM, which provides instructions specialized to the blockchain environment alongside typical arithmetic and control flow instructions. For example, the instructions `CALL` and `CREATE` call other smart contracts and create

[1] `https://qz.com/1300832/bitcoin-trading-in-venezuela-is-skyrocketing-amid-14000-inflation/`
[2] `https://www.buybitcoinworldwide.com/price/`
[3] `https://www.wired.com/2014/03/bitcoin-exchange/`

them respectively, as if sending a transaction. While a transaction may only perform a single call or create, smart contracts may execute the `CALL` and `CREATE` instructions repeatedly.

1.2 Blockchain cost models

Because untrusted third parties may submit transactions for execution on distributed ledgers, there must be a means to protect the ledger from denial of service attacks. This is implemented via cost models built into the protocol. Blockchain cost models limit the computational resources used per block; because the rate of block creation is limited, that limits the computational resources needed to verify blocks as they arrive. For non-Turing-complete blockchains, the cost model can be simple: Bitcoin simply limits the total size in bytes of transactions submitted. Since the amount of computation performed per byte of data uploaded is limited, that successfully prevents a DoS of the computational resources of the nodes which maintain and verify the ledger.
Because it is Turing-complete, Ethereum's cost model is more complex. Each transaction sent, byte of data uploaded and EVM instruction executed costs a certain amount of a unit of account called gas. The amount of gas used per block is limited. EVM instructions must be metered because smart contracts may perform looping computation, which could go on forever if its execution was not limited.
The transaction capacity of blockchains is limited. The economic cost of creating new blocks is also high by design, and there is contention among users who wish to include their transactions in the blockchain. Consequently, miners require fees from transaction senders in return for accepting their transaction. Transaction execution costs are billed in cryptocurrency; the cost in real terms is high, on the order of several USD cents per transaction for Ethereum [11]. This is due to massively redundant ledger update execution and ledger state storage in current blockchain protocols. While redundancy is perhaps not an inherent problem of distributed ledger technology given possible improvements to distributed verification, it is a problem today. As such it is of great interest to blockchain space participants and researchers to reduce execution costs. The technique of transaction batching is particularly relevant to this thesis; other optimization technologies are discussed in more detail in chapter 2.

1.3 Transaction batching

Because the gas cost of performing a CALL or CREATE is lower than sending an equivalent transaction, gas costs can be reduced by using a contract which executes multiple such instructions as specified by calldata. Contracts which do this are known as batchers. These are already implemented on the Ethereum blockchain and can reduce transaction costs in some instances. However, existing batchers are limited in their expressive power: they can make a sequence of payments of different amounts and to different recipients, but only in a single cryptocurrency on behalf of a single sender. They implement this simple functionality using a for loop in

Solidity [17], the most popular high-level language targeting the EVM.

1.4 MultiCall, a general batching interpreter

In this thesis I describe the development and implementation of MultiCall, an Ethereum smart contract capable of emulating arbitrary sequences of typical transactions. My contribution is in realizing that batching corresponds to interpretation, and that by devising a more general interpreter more transactions can be batched together. Indeed, it is possible to batch not only payments from a single sender but general calls and creates from multiple senders! By doing so, a full block of transactions can be emulated using a single transaction. Batching full blocks as a matter of course holds the potential to save Ethereum network participants a large amount of money, on the order of millions of US dollars per year. One may ask why such wondrous functionality has been left unimplemented so far. Comprehending the underlying concept is not sufficient to implement a performant and safe interpreter on the EVM; there are significant practical challenges.

1.5 Implementation challenges

A general batching interpreter is an unusually complex smart contract, both in its implementation and design. First, an instruction set capable of emulating the full functionality of a series of transactions must be designed. It must include a mechanism for providing each user with a unique identity when calling smart contracts which rely on the caller address for authentication; it must also be possible for scripts from multiple senders to be run together, with the permissions of each separated. Invariants must be maintained to ensure that users cannot steal or destroy each other's funds. Expressive power and potential optimizations must be traded off against the complexity of the contract, which reduces implementation feasibility and renders verifying the absence of bugs more difficult.

Once the high-level design is complete, low-level design is no less challenging. To process cryptocurrency transactions and reduce transaction costs for users, high requirements are placed on both security and performance. Not least, an efficient interpretation scheme must be devised in Ethereum's unusual instruction set and cost model. As uploading data is expensive, the size of instructions must be minimized. In practice, instructions must be able to parse immediate arguments of variable size. The programmer requires low-level control of generated EVM code, data layout and calldata parsing. To manage the complexity of large and performant EVM contracts, it must be possible to compose them from smaller units and to implement functions which automatically generate and combine performant code. It is the opinion of the author that the design of Solidity makes these tasks needlessly difficult, and its adoption may have impeded development of efficient and secure smart contracts on Ethereum. As such I felt the need to develop an EVM bytecode generation DSL to circumvent it, a non-trivial task in itself.

1.6 Hypothesis

My hypotheses are as follows:

- **H1**: It is possible to implement an Ethereum interpreter smart contract which can support functionality practically equivalent to a block of typical transactions in a single invocation.
- **H2**: It is possible to implement an interpreter smart contract which fulfills H1 while reducing gas costs compared to sending those transactions individually, for transaction sequences of useful length.

I define practically equivalent functionality to a block of transactions to mean that that the interpreter may emulate any typical sequence of transactions. To emulate a typical sequence of transactions the interpreter must support 1) an arbitrary interleaving of calls or creates to typical EVM code, each of which may be on behalf of a different signatory; 2) a unique Ethereum smart contract `proxy(A)` for each signatory with address `A`, which only `A` may control and which may make calls and creates; 3) a separate ether balance for each signatory, such that payments made by `A` are deducted from `A`'s account.

Typical EVM code is agnostic to whether its caller is a smart contract or an EOA, and does not implement another interpreter. Only top-level transactions have a public key caller address; smart contracts can only make calls from smart contract addresses. A smart contract developer who wished to prevent an interpreter or batcher from emulating call transactions to their contract could check for and reject calls from smart contract addresses. However, that is not common and does not grant any practical benefit; as the possibility does not impact the usefulness of the interpreter I choose to ignore it.

A call to another interpreter may be more efficient to send directly, if it is more efficient for the functionality required by the user. For some use cases, such as making many payments from a single user, a less general batcher may therefore be more efficient. Most transactions are not to batchers; as I have defined such transactions to be atypical I do not consider them in evaluating H2. The threat to the usefulness of MultiCall posed by competing batching solutions is discussed in more detail in chapter 2.

Because interpretation imposes an overhead and the fixed transaction cost is paid for sending a call transaction to an interpreter or batcher, there is some minimum sequence length of transactions greater than 1 below which it is more efficient to send the transactions directly than to batch them. The higher the minimum length, the less practical it would be to adopt the interpreter as more transaction senders would need to coordinate. Ethereum blocks frequently contain over 100 transactions; I will consider a minimum efficient sequence length below 50 to be satisfactory for H2.

1.7 Methodology

1.7.1 Design Science Research

I chose to use the framework of *design science research* [23] to guide development and scientific evaluation of the interpreter. Design science research is a methodology

developed in order to formalize research in computer science, which by the nature of the field commonly deals with software artifacts created by the author. This poses a challenge to fitting computer science research into the traditional scientific framework, which primarily deals with measurement of preexisting objects in the natural world; design science research is an established solution.
Where the natural scientific framework focuses on proving or disproving a hypothesis about a natural phenomenon, design science research emphasizes design and evaluation of a useful and novel artifact. DSR describes research using two main concepts, *processes* and *artifacts*.
Artifacts encompass the concepts (termed constructs) and tools used to guide and aid design, methods of implementation (whether programmatic or applied by humans) and the finished product (termed instantiation). Example artifacts used in the course of this thesis include the monad and functional DSL concepts, the EVM-generating DSL tool and combinators derived from it, optimization techniques such as caching and batching, and the interpreter implementation itself.
The work of research is categorized into two processes, *build* and *evaluate*. Prior to building, the relevance and rigor cycles initialize the research environment with knowledge of the existing context and stakeholder needs (to ensure the design's relevance) and prior work (to ensure novelty of the design and thus scientific rigor) respectively. Building consists of a number of steps. First the researcher investigates the needs of stakeholders and the tools available and assembles constructs to meet those needs. They then combine those constructs into a design, assess the design, and implement it. In the evaluate process, the researcher assesses the utility of the instantiation and the degree to which it fulfills the design goals set in the build process. The build and evaluate processes are typically iterated a number of times, with the researcher using knowledge gained from evaluation to refine the problem definition and design. This is called the design cycle.

1.7.2 Thesis Procedure

Proceeding according to the principles of DSR, I designed and evaluated the MultiCall interpreter in order to validate the hypotheses H1 and H2. The work was divided into a number of steps corresponding to the DSR approach, described in more detail below.

1. **Relevance cycle**: First I surveyed the field of Ethereum, choosing to focus on gas cost optimization of smart contracts. I studied the behavior and cost model of the EVM, observing that there were multiple fixed costs that could be amortized through efficient program design. In particular, I concluded that the fixed cost of transactions was significant and that an interpreter could amortize that. Initially I favored a Turing-complete design, as that would allow more expressive Ethereum transactions and be able to share more fixed costs (for example, the cost of uploading a script to run could be amortized by looping over it).
2. **Rigor cycle**: I then investigated methods of transaction cost optimization; this is described in more detail in chapter 2.
3. **Design and implementation**: Having settled on the idea of a batching in-

terpreter smart contract and verified its novelty and usefulness, during this phase I took the steps necessary to implement a concrete instantiation. I assembled the conceptual and software tools required for the interpreter's implementation, iteratively developed prototype contracts, and finally implemented a refined design. First I sketched the initial design of the interpreter, encompassing an efficient dispatch method, a set of features, and a collection of optimization techniques to use. Then I considered the programming languages available targeting the EVM, and settled on personally developing a language for the task. Detail on my rationale for doing so and the language's design are given in chapter 5. Once the language was developed, I wrote test contracts in it to verify that it generated correct code (finding and fixing bugs in the process). Standard Ethereum development tools were used to run the contracts on a private chain. I then proceeded to develop and test a series of prototype interpreters, verifying first that it was at all possible to create one. I used feedback from developing and evaluating prototypes to extend the EVM code-generating language with new features as necessary and for convenience. Prototype development culminated in the creation of a Turing-complete interpreter which emulated the EVM instruction set. This verified that a Turing-complete interpreter could be built with the tools I had available. While the prototype imperfectly supported calls (making a dummy call to a fixed address instead of one chosen by the user), it showed that calls from interpreter instructions were feasible and efficient compared to equivalent calls from transactions. Finally I used accumulated experience from developing interpreters to design and implement a final interpreter, which I chose to call MultiCall. I focused on transaction batching, adding new features for the purpose and cutting extraneous ones (including Turing-completeness). I continued to develop and evaluate iteratively, adding and testing new features and patching or extending the language as necessary, until MultiCall was complete. During this process I verified that the interpreter behaves as intended and supports equivalent functionality to a sequence of transactions.

4. **Final design evaluation**: Once the interpreter was complete, it was time to evaluate it. The functional correctness of MultiCall was verified using informal unit tests during debugging; correct functionality is illustrated in Evaluation. H1 was verified through observing that the instructions provided by the interpreter together implement the functionality provided by a sequence of transactions: calls and creates from multiple signatories, each with their own separate identity and balance. H2 was evaluated through unit tests (shown in Evaluation) in which sequences of 0-99 instances of the same instruction were executed and the marginal gas cost of each instruction measured.

1.7.3 Limitations

As noted above, H1 and H2 were evaluated using unit testing. MultiCall would benefit from more advanced verification and testing techniques, specifically formal verification and property-based testing. MultiCall lacks a formal specification of its intended behaviour and a machine-verifiable proof of its correctness; there may be a

subtle bug in the interpreter which would be discovered in the course of attempting to formally prove the desired properties. Gas usage profiling in order to verify H2 was implemented by running many instances of the same instruction in sequence. A property-based testing tool could generate and profile randomly generated sequences of different instructions, potentially finding complex interactions between instructions which break H2. I assume 1) there is no bug in MultiCall violating functional correctness and H1 which would be discovered by formal verification; 2) there is no complex interaction between instruction costs which would be discovered by property-based testing.
In December 2019, the Istanbul hard fork of Ethereum was accepted by miners. In a hard fork the rules of a blockchain are altered, rendering its subsequent evolution incompatible with the old protocol. The Istanbul hard fork primarily altered relative gas costs; in particular, the cost of uploading data was reduced to 16 gas per byte from 68. MultiCall was evaluated on a pre-Istanbul private chain, where I found the data upload cost to be a significant contributor to the execution cost of instructions. As such I believe the Istanbul changes improve MultiCall's performance, but I have not verified that. The Ethereum protocol is in continual flux; redesign and reevaluation of MultiCall for new versions of Ethereum is a subject for future work.

2
Related Work

In this chapter, I describe some technologies closely related to the MultiCall interpreter, both in purpose and due to similarity to the interpreter itself.

2.1 Off-chain scaling solutions

An off-chain scaling solution optimizes transactions secured by the blockchain by moving them off it, thus avoiding the high cost of massively redundant state storage and computation. Off-chain scaling still benefits from the consistency and censorship resistance of the blockchain by committing summaries of a number of transactions rather than each individual transaction. It functions by allocating a new distributed ledger with less redundancy (and typically fewer involved parties), which is linked to the parent distributed ledger such that state from the child can be committed to the parent. Off-chain scaling may either be implemented at the user level above the host distributed ledger protocol, or implemented in the protocol itself.

2.1.1 State channels

State channels [26] are an example of the former, where linking is implemented via a smart contract. State channels implement a new distributed ledger controlled by two or more signatories. To create it, a smart contract is first created and funded by the signatories. That contract serves as the link to the new ledger. The ledger contains a nonce, and typically a cryptocurrency balance per signatory. The initial balance value for each signatory typically corresponds to the amount deposited by them. To update the state of the ledger, all signatories must sign a new state with an incremented nonce. That ensures that each signatory can trust the state of the ledger to remain consistent. Funds may be withdrawn from the smart contract by unanimous agreement among the signatories, or by one party submitting a tentative ledger and no other ledgers with a higher nonce being submitted during a timeout period. State channels require only the signatories to keep a record of the ledger state and compute whether updates to it are valid; that significantly reduces the computational cost of transacting between them. However, note that *only* the signatories can trust the state channel ledger to remain consistent! Therefore only they can securely receive payments from the state channel, unlike a blockchain where funds may be transferred to any key.
State channels can render the cost of peer-to-peer cryptocurrency payments negligible and reduce their latency to the network round-trip time. That may at first

glance seem to threaten the value of on-chain optimization solutions such as MultiCall. However, off-chain payments still rely on the blockchain for security, and users must interact with it to deposit to or withdraw funds from state channels. Off-chain scaling increases the transaction value secured by on-chain transactions, thus increasing the value of on-chain capacity and optimizations. In turn, on-chain optimizations increase the blockchain transaction capacity, allowing more scaling solutions to be used. Rather than being competing solutions, on-chain optimization and off-chain scaling are complementary approaches.

2.1.2 Sharding

Sharding [16] is an optimization technique originating in database management where a single logical database is distributed across several machines, or a service is split into multiple separate databases to handle customer demand. In the crypto space, it refers to scaling a distributed data structure by implementing it using multiple distributed ledgers which may be updated in parallel, termed shards. By dividing the verifier nodes across the shards, redundant computation and storage can be reduced and the transaction capacity of the composite distributed ledger can be improved by orders of magnitude. The blockchain Zilliqa [28] supports sharding in its protocol, and has achieved over 2000 transactions per second[1]; Ethereum is currently limited to around 10[2]. There have been proposals to add sharding to Ethereum, but they have not yet been implemented.
Sharding renders decentralized computation more efficient and abundant, thus reducing the economic value of transaction batching on any one shard. However, a batching interpreter could be run on *every* shard. The benefits of single-shard sequential speedups and parallelization via sharding should scale multiplicatively, but whether the benefits are captured by end users or optimization authors remains to be seen.

2.2 Transaction batching

Transaction batching is a method of reducing on-chain transaction execution costs by emulating a number of transactions with a smaller number that have an equivalent effect. The concept is well-known to Ethereum developers. Several payment batching contracts exist [3] [4] which allow the caller to make payments in a single currency to a number of recipients. The company Autherium also provides wallet contracts controlled by a single user which may make a series of calls on behalf of that user [12]. Preexisting Ethereum batchers differ from MultiCall in functionality and design. Where MultiCall supports multiple instruction types in a single call using a jump table and assembly-level dispatch code, existing batchers support only one instruction type via a loop in the high-level language Solidity. They are not explicitly referred to as interpreters by their creators; I believe the connection between batching and interpretation may be a novel contribution of this thesis. It is certainly

[1] `https://zilliqa.com/platform`
[2] `https://etherscan.io`

not impossible to implement a multi-instruction interpreter in Solidity, as I show in the next section. Prior batchers additionally do not provide support for batching contract creation or acting on behalf of multiple signatories in a single call. They can therefore only batch a small proportion of transactions together. MultiCall is designed to emulate any sequence of transactions, and is to my knowledge the first batcher to achieve that functionality.
Transaction batching is not exclusive to Ethereum or Turing-complete blockchains, and does not always need to be accomplished using a smart contract: Bitcoin transactions natively support sending to multiple outputs, which can be used to reduce transaction costs by up to 80% [22].

2.3 On-chain interpreters

While not the most common type of smart contract, other multi-instruction interpreter contracts do exist. A number of Solidity interpreters for the toy esoteric programming language Brainfuck were written for a programming contest in 2018[3]. On the Cardano blockchain [7], an interpreter smart contract is used to implement a financial contract representation language called Marlowe [27]. On Ethereum, an interpreter for a simple virtual machine called Lanai is used to verify off-chain computations [1]. The application of a multi-instruction interpreter to transaction batching appears to be novel.

2.4 Related techniques

To perform payments and other instructions on behalf of multiple users within a single transaction, MultiCall verifies the signatures of signed scripts created by the users before executing them with the permissions of the signatory. Data which is not a valid Ethereum transaction but is signed by an Ethereum private key and used for authentication in a smart contract is known as a *meta transaction* [19]. This technique is used to assist in onboarding Ethereum smart contract users, so that they do not need to hold ether for transaction fees before being able to transact on-chain.
Proxy contracts [18] are a type of contract controlled by another user or contract, which forwards calls on their behalf. They are used to provide Ethereum users with on-chain accounts enhanced with smart contract functionality such as transaction batching; Autherium's account contracts are an example of a proxy. Proxies are useful because Ethereum smart contracts use the address of the caller for authentication, for example to determine whose balance in internal storage to debit when a payment method call is made. MultiCall allows users to spawn and control their own proxy contracts in order to give each user a separate identity.

[3] `https://g.solidity.cc/challenges/brainfuck`

3
Approach

The following chapter is a brief illustrated overview of the software architecture of the MultiCall interpreter smart contract.

3.1 Architecture

The MultiCall interpreter is a program designed to run on the Ethereum blockchain. To do so, it must consist of EVM bytecode, which is a bytestring containing EVM instructions and potentially some data. The concrete implementation of MultiCall is simply a text file containing a hex string representing EVM code; when submitted to the blockchain via a create transaction, a smart contract is created with a pseudorandom address which implements the functionality of MultiCall. The difference between the EVM code file and the smart contract on the chain is analogous to the difference between an executable file and an operating system process running it. Some EVM programs such as option or escrow contracts are intended to have multiple instances; MultiCall only needs one to provide its functionality. Once created at address `MC`, users can send call transactions to `MC` with MultiCall scripts as their calldata. The MultiCall instruction set emulates the functionality of arbitrary sequences of typical transactions, allowing users to achieve the same effect on the blockchain with fewer transactions and thus a lower monetary cost. The basic gist is illustrated in figures 3.1 and 3.2.

3.1.1 EVM code generation

Writing anything but the most trivial programs directly in hex code is impractical; instead I wrote a library in the functional programming language Haskell to generate EVM bytecode, and then used that to write MultiCall. The Haskell library implements a small single-purpose programming language I named C, for code generation. C language scripts are Haskell expressions which are interpreted to generate EVM bytecode, which can then be written to a file for later use. Writing in that language allowed me low-level control over the EVM instructions generated, the ability to build large EVM programs compositionally from small and easy to understand parts, and to automatically generate tedious boilerplate such as jump tables.
A strength of my C language is that it's easy to combine simpler components into more complex ones using functional combinators. It is itself multi-layered, with higher-level features defined in terms of lower-level ones. That makes each level simpler and easier to debug. The language can broadly be divided into three layers

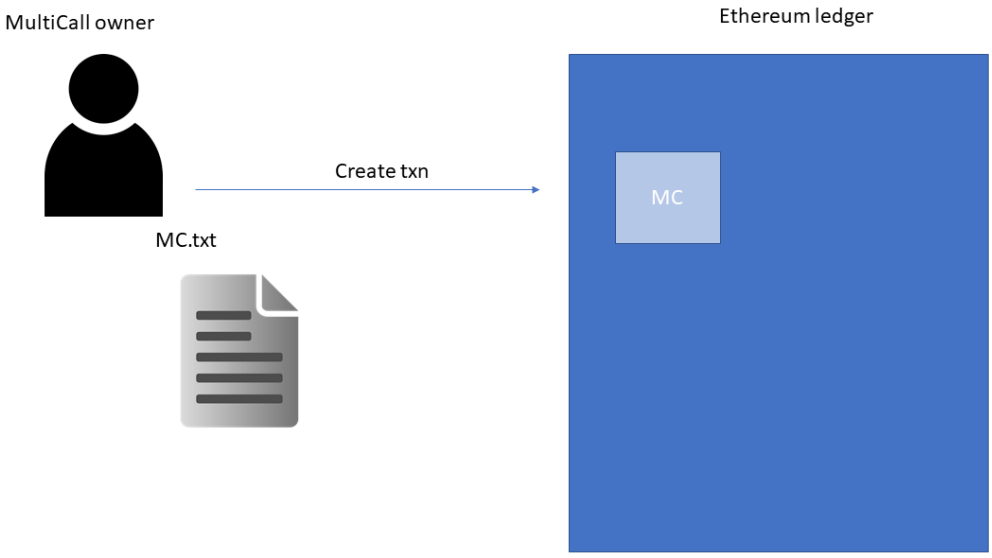

Figure 3.1: To publish MultiCall, the owner uploads the EVM bytecode stored in MC.txt to the Ethereum blockchain using a create transaction. It is then accessible at a pseudorandom address MC.

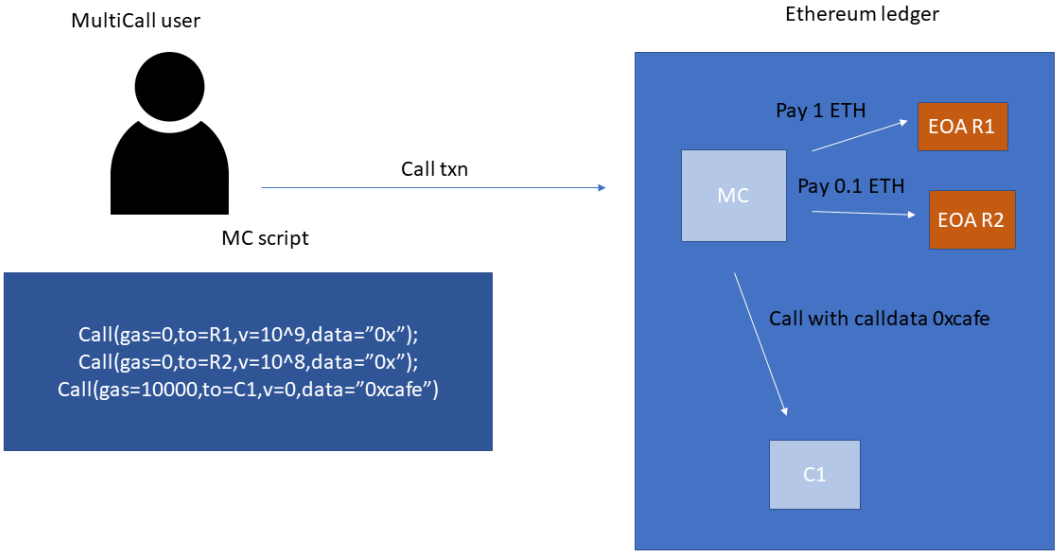

Figure 3.2: Users can emulate multiple transactions by sending a sequence of MultiCall bytecode instructions to MC in a call transaction. In this example the user makes payments to two externally owned accounts and calls the smart contract C1 with 2 bytes of calldata.

of complexity. At the lowest level, there are primitives which perform basic tasks of code generation such as outputting bytes and tracking and placing jump labels. On top of them, there is an intermediate layer of functions which generate EVM instructions. On the third and final layer, those instructions are composed to implement structured programming patterns, simple data types such as strings, and stack and global variables.

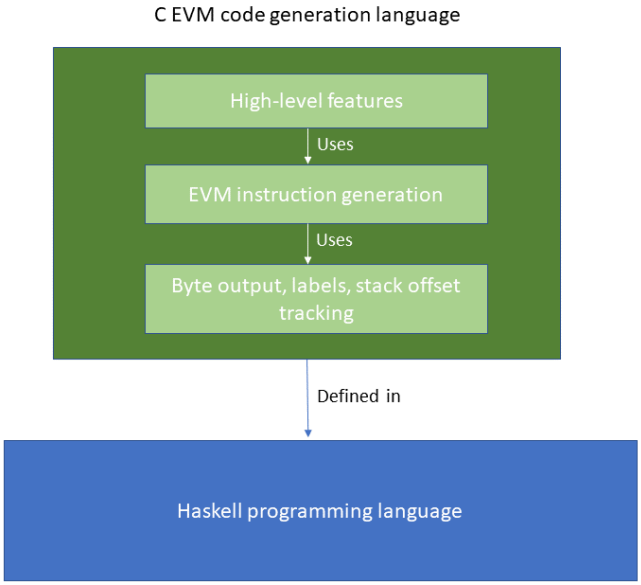

Figure 3.3: The design structure of the "C" EVM code generation library.

3.1.2 MultiCall structure

MultiCall is written in my C language, and is also composed from simpler components. It is defined in the C script `mc_alt_deploy`. When compiled to EVM code and submitted to the Ethereum blockchain using a create transaction, it creates an interpreter whose immutable code is specified by the C script `mc_alt` and performs some creation side effects. The creation side effects initialize some persistent storage variables of the interpreter, among other things setting the administrator of the interpreter to the transaction sender. The administrator can set the fees charged for use of the interpreter and collect those fees; this is described in more detail in later chapters.

The most important components of `mc_alt` are its dispatch mechanism, instruction set and initialization code. The dispatch mechanism is specified by two values, `dispatch_mc` and `jumptable_mc`. The macro `dispatch_mc` generates bytecode to fetch and interpret the next MultiCall instruction; it is used by all instructions except those which exit the interpreter. The function `jumptable_mc` takes a list of instructions defined in the C language, generates a jump table to them, and lays them out in code. In the definition of MultiCall, it is applied to the instruction set

`iset_alt`. The instruction set `iset_alt` is conceptually divided into a number of features, each of which is designed to emulate some aspect of transaction functionality. For example, the Call feature allows the user to make calls; the Signed feature uses signature verification to allow scripts from multiple signatories to run in a single transaction; the Proxy feature gives each signatory a unique identity from the perspective of the smart contracts they call. Stop instructions exit the interpreter and maintain its invariants by throwing an exception and reverting execution when they would be violated. The initialization code `init_alt` is the entry code for MultiCall, which is always executed when MultiCall is called; it initializes the volatile state of the interpreter and then enters it. The structure of MultiCall is illustrated in figure 3.5.

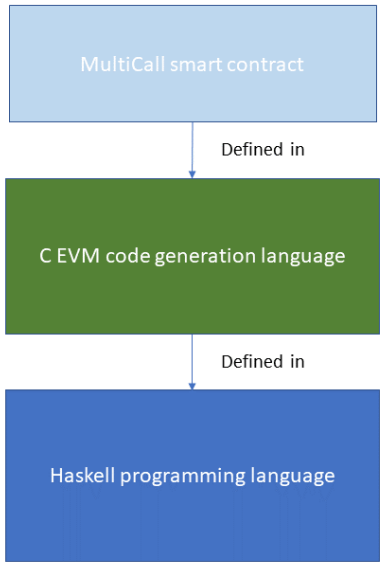

Figure 3.4: The dependencies of the MultiCall smart contract. It is defined as a C script which may be compiled to a text file and uploaded to the blockchain.

3.2 Usage Demo

The MultiCall smart contract itself is agnostic to how it's deployed and accessed; currently I use the standard Ethereum RPC library web3 to interface with a private blockchain via truffle, a javascript shell specialized for Ethereum. I use my own utility library util.js to deploy contracts from text files, and my API library MC_api.js to generate MultiCall instructions from their mnemonic names and read the contract's state.

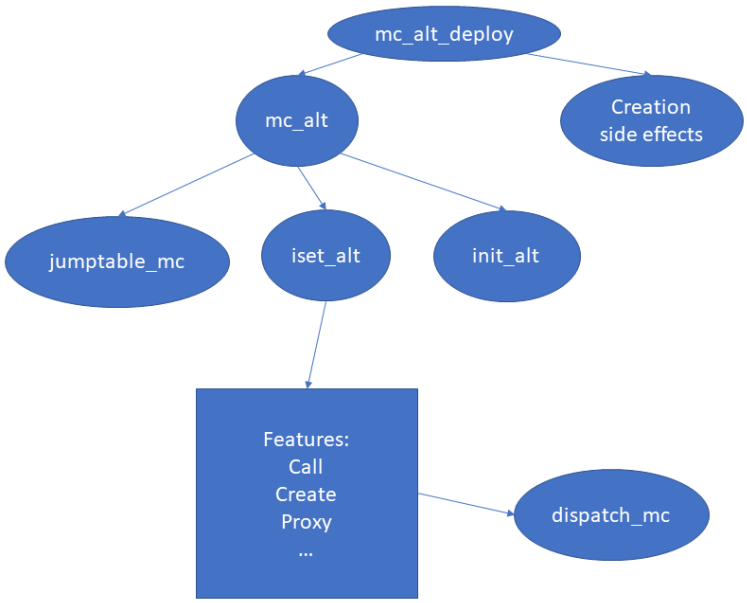

Figure 3.5: MultiCall creation code internal structure. Arrows indicate usage.

3.2.1 Building and deploying MultiCall

First MultiCall must be compiled to EVM code. This is done in the Haskell environment using the function `hackyDeploy`, which compiles a given C EVM program and writes it to a file. In the folder containing my Haskell code, I load the module SimpleMC.hs which defines the MultiCall interpreter; I then compile MultiCall to the file MC.txt in my demo folder. This is shown in figure 3.6.
I then start a private chain instance using ganache-cli and open a javascript console to it using truffle. This is pictured in figures 3.7 and 3.8
Then I load my utility and MultiCall API modules, deploy MultiCall and bind its Ethereum address to the variable `MC`. This is shown in figure 3.9. Figure 3.10 shows the transaction which deploys the MultiCall interpreter registered on the private blockchain.

3.2.2 Transaction batching

The following commands (see figure 3.11) show the balances of four externally owned accounts on the private chain, `acc[1]` to `acc[4]`, which start with 100 ether each by default. I rename the array of accounts to `acc` from `web3.eth.accounts` for brevity. One ether is defined to be 10 to the eighteenth times the minimum currency unit representable in the Ethereum protocol, the wei. The first command shows them as big integer objects; the second coerces them to more readable primitive numbers.
With the following command (see figure 3.12), I send one billion wei to each of the EOAs from the EOA `acc[0]` using four separate call transactions. Figure 3.13 shows the gas cost is 21000 per call, 84000 in total. Figure 3.14 shows their balances have been incremented by one billion, also called a gwei.

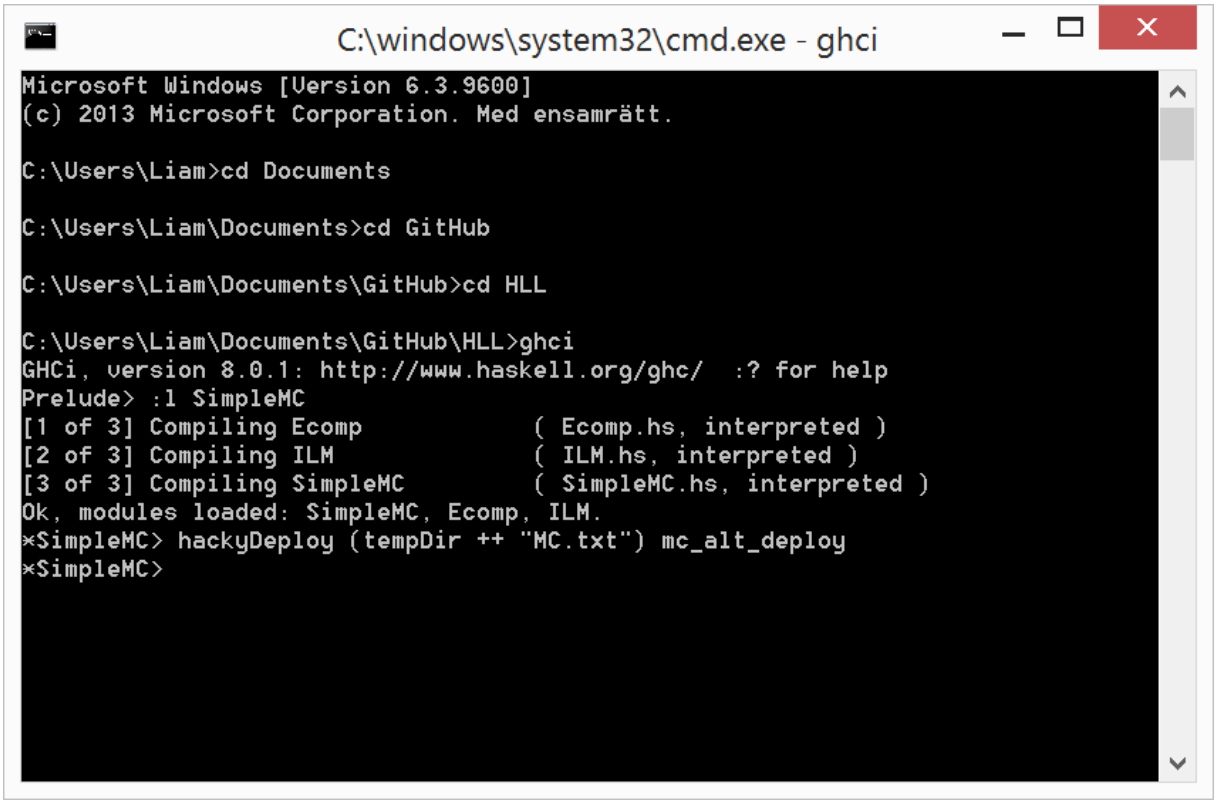

Figure 3.6: Compiling MultiCall.

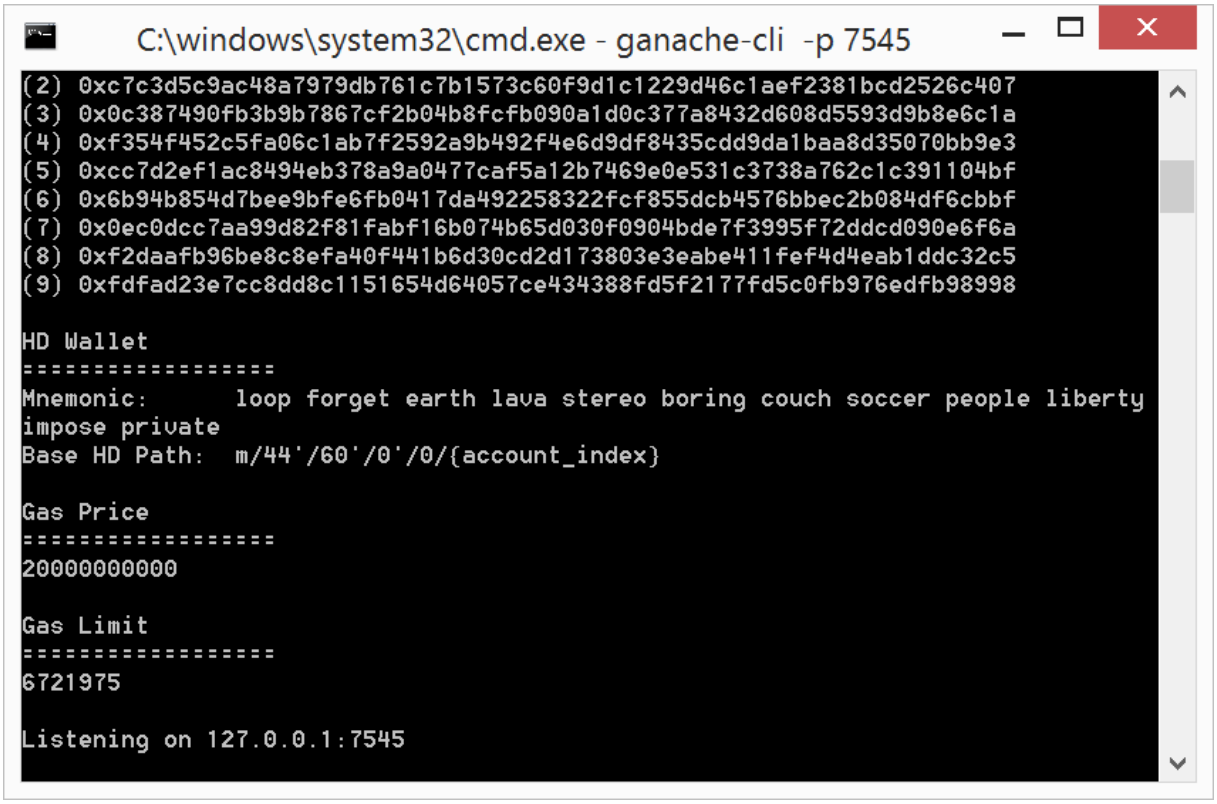

Figure 3.7: A private instance of Ethereum.

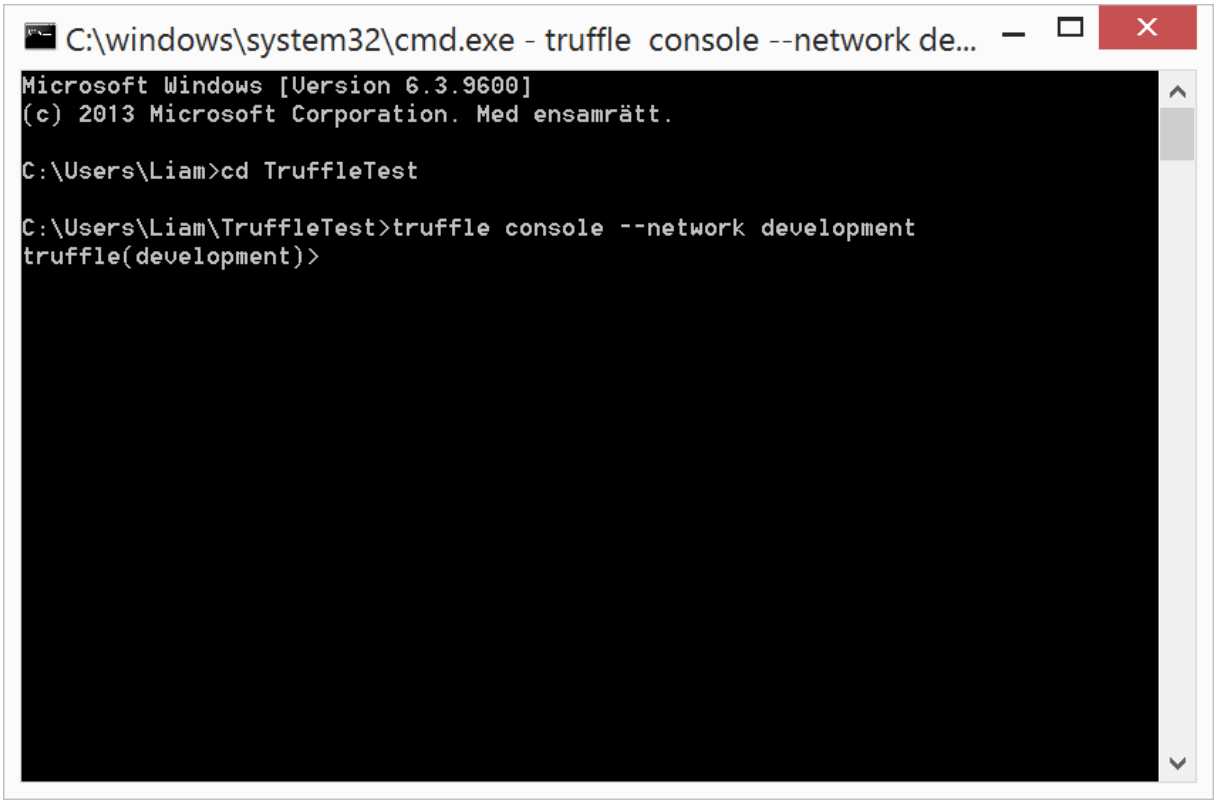

Figure 3.8: Starting a console to the private chain.

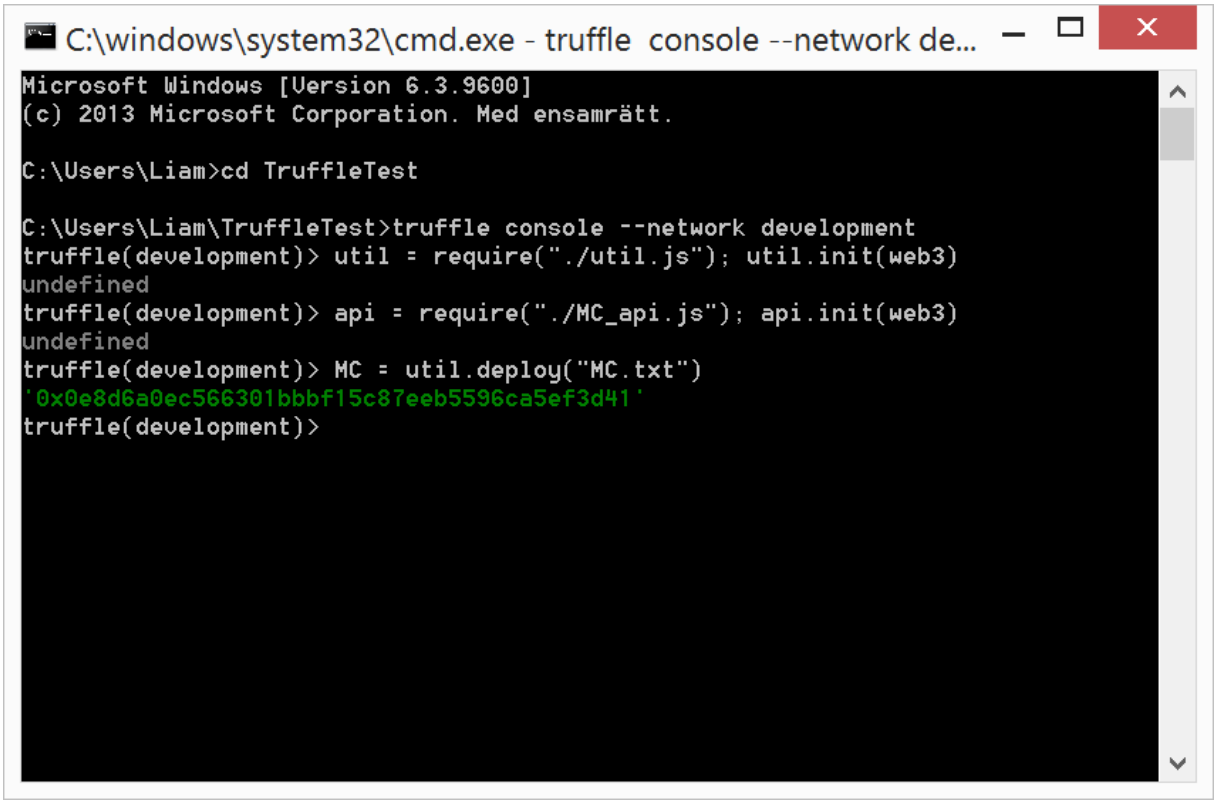

Figure 3.9: Loading javascript tools, deploying MultiCall.

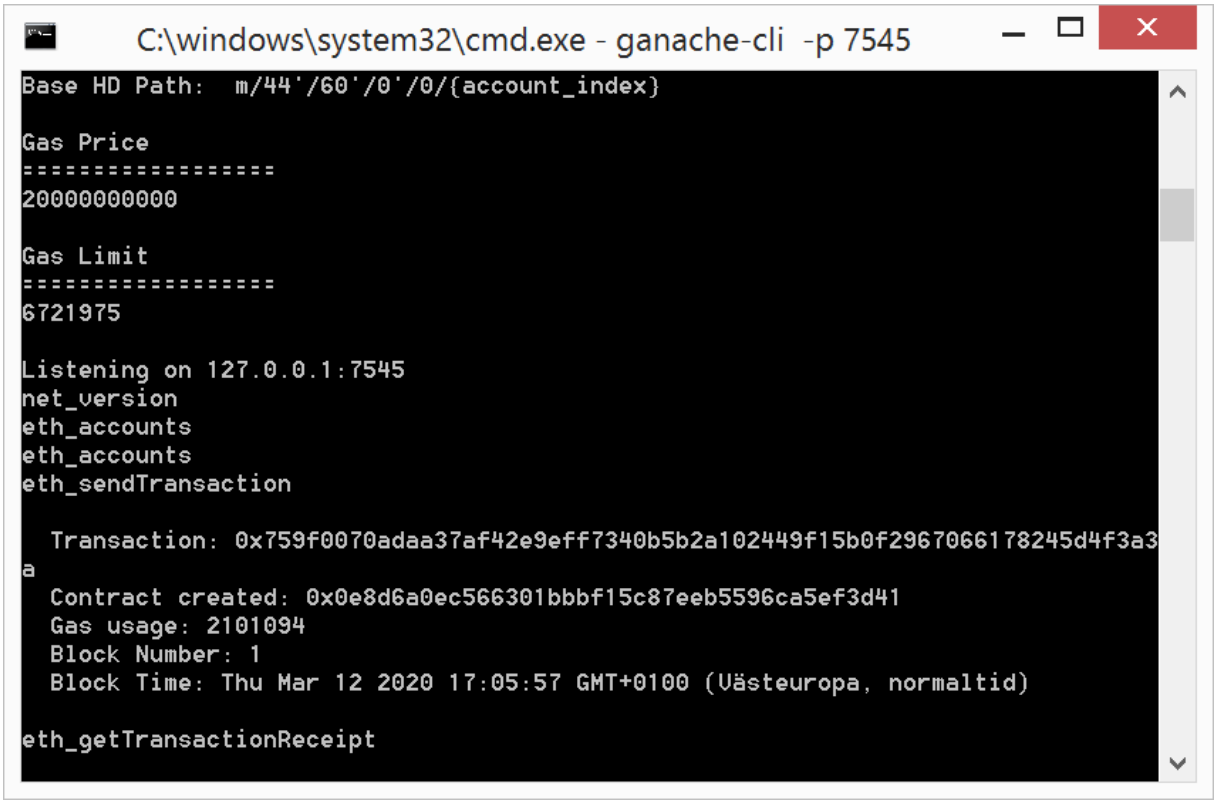

Figure 3.10: MultiCall create transaction registered on the private blockchain.

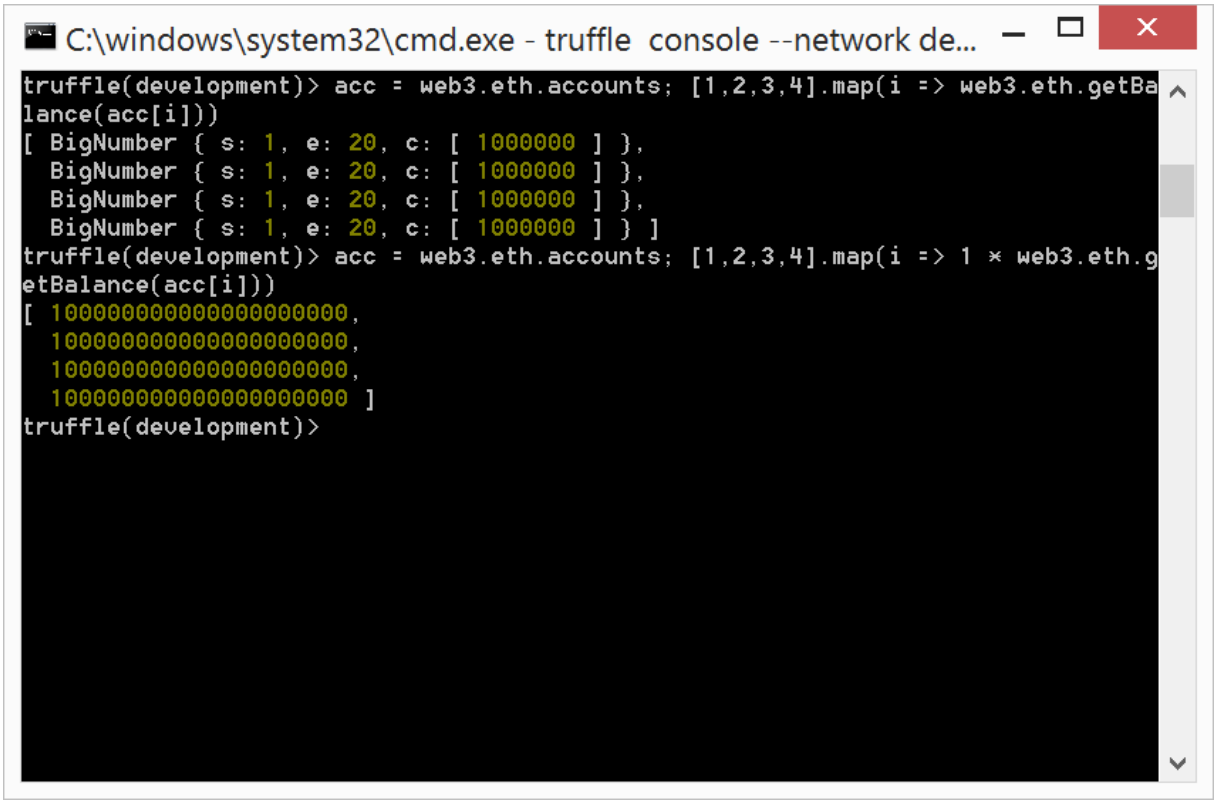

Figure 3.11: The private chain starts with a number of accounts. These all have initial balances of 100 ether.

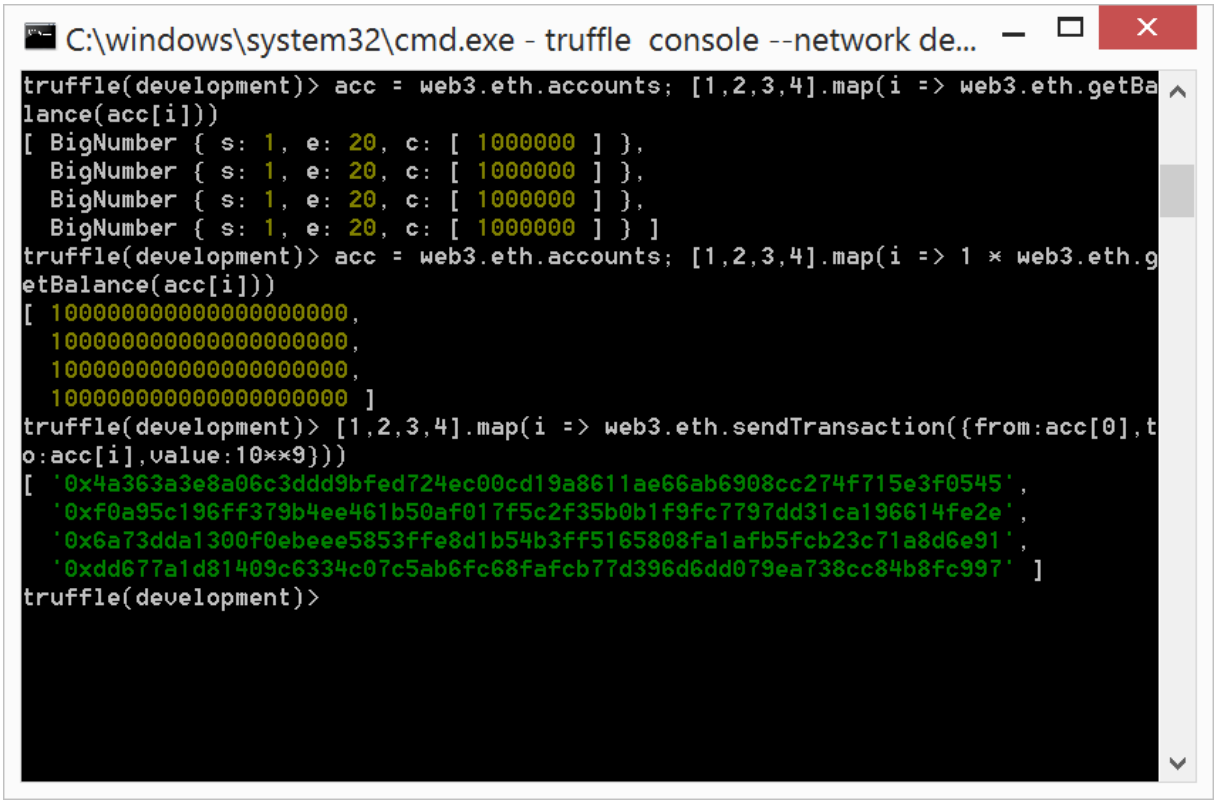

Figure 3.12: Four unbatched payments.

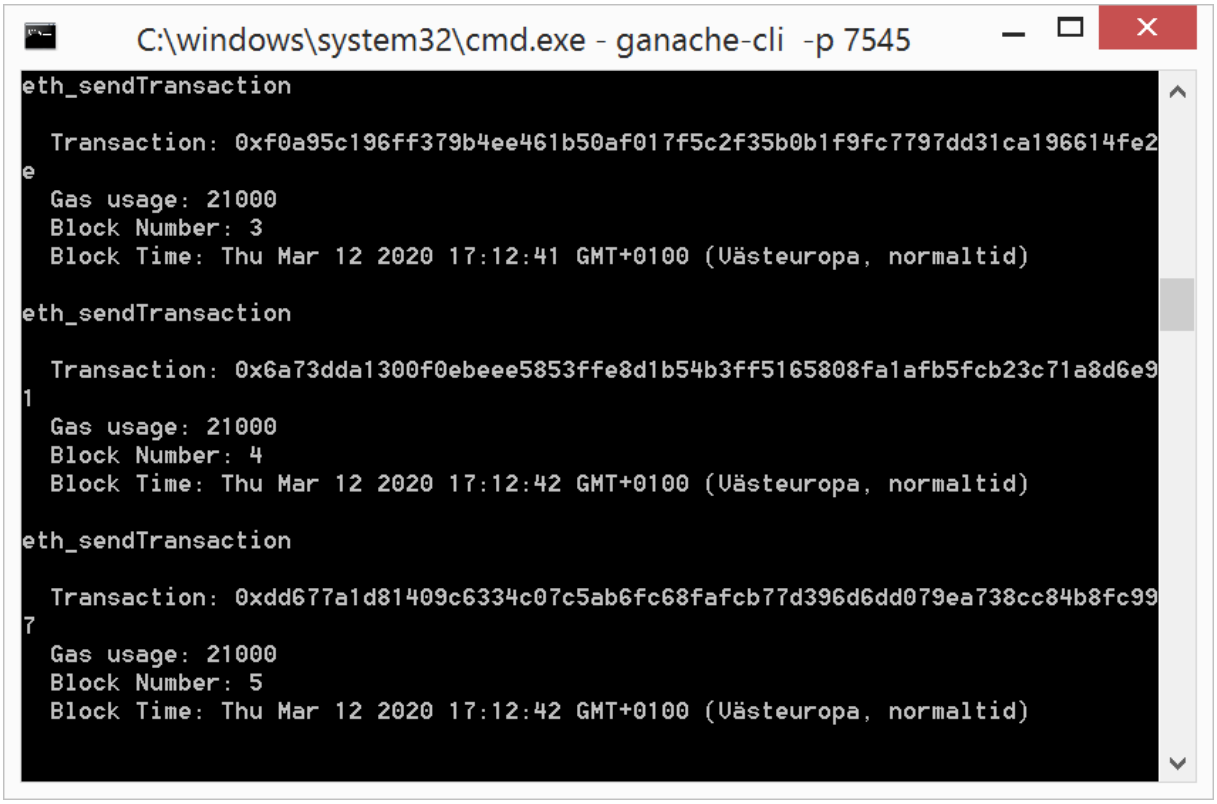

Figure 3.13: Each transaction costs 21000 gas.

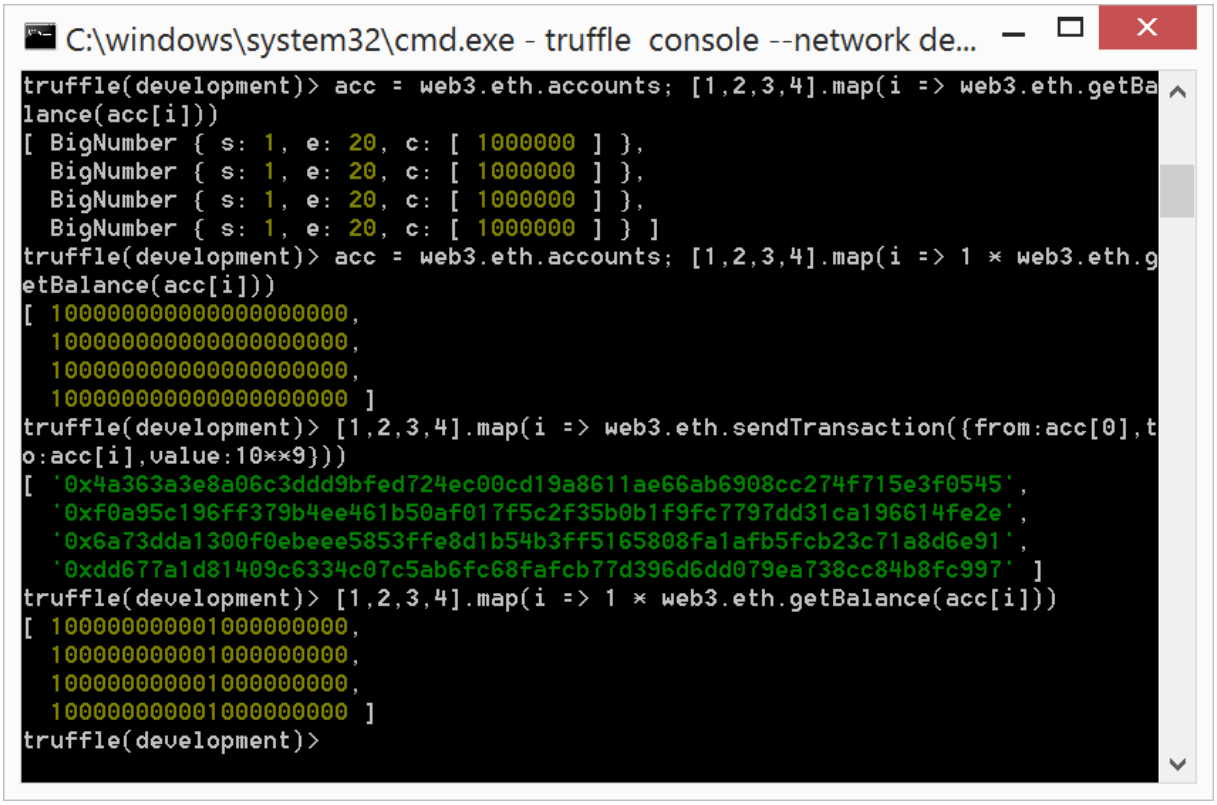

Figure 3.14: After the payments have been made, each balance has one gwei more.

Now we compare the performance of unbatched transactions with batching via MultiCall. In figure 3.15 I initialize the balance of `acc[0]` held in `MC` by paying one ether to `MC` from `acc[0]`. That is done separately from the batching to avoid the cost of allocating `acc[0]`'s balance (which only needs to be done once) from confounding the cost comparison. I also define four MultiCall instruction instances `pay1` to `pay4` which implement 1 gwei payments to `acc[1]` to `acc[4]` respectively.

I then send a batched transaction (see 3.16) consisting of the string concatenation of a number of payments followed by the stop instruction `api.iset.stop()` which exits the interpreter. The transaction's gas cost is 59795 (3.17), significantly cheaper than 84000. However, looking at the balances again (3.18) we can see that I only paid to `acc[1]` to `acc[3]`. Oops! However, batching three payments is still marginally cheaper than sending them separately (which costs 63000).

Let's try that again. This time I pay to all four recipients, as we can see from the change to their balances (3.19). The gas cost is 68986 (3.20), which is still significantly cheaper than sending the four payments separately! Because the fixed cost has been overcome, emulating further transactions would provide even better proportional savings. One can do more than just make ether payments with MultiCall, but for the sake of brevity that is not shown in this demo.

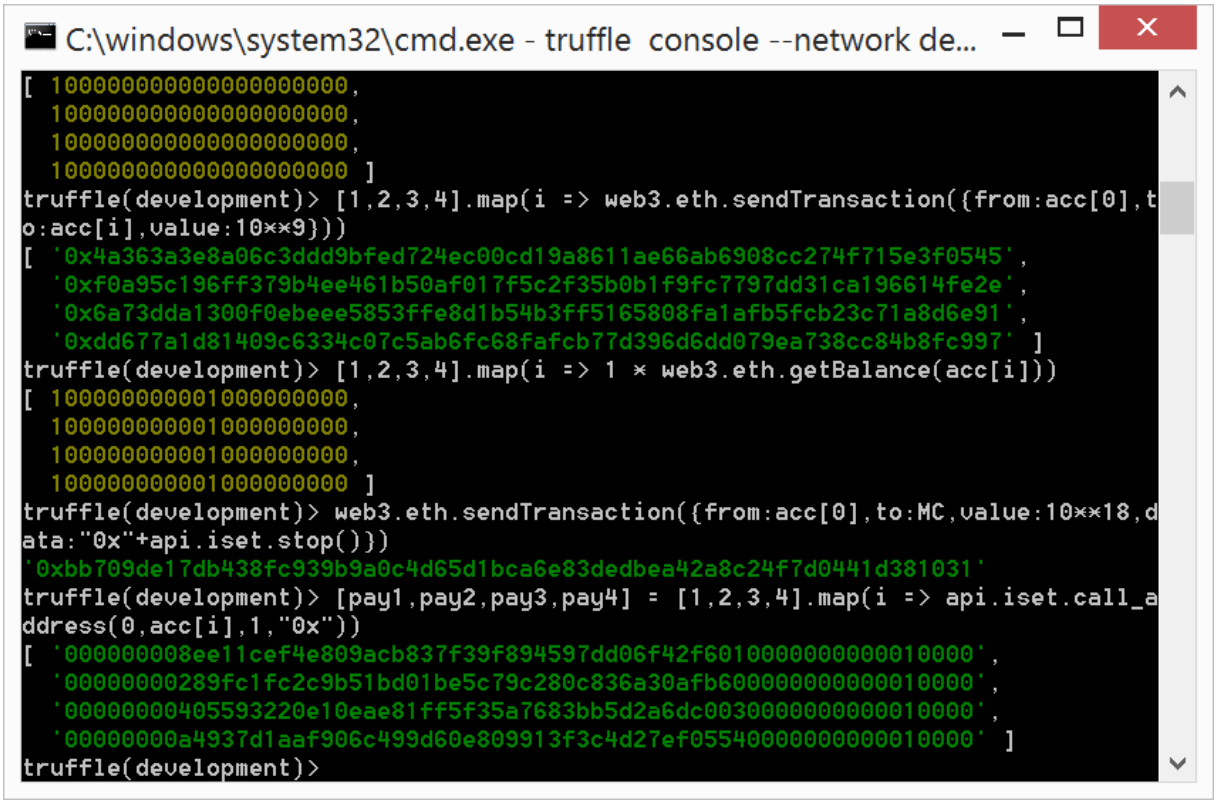

Figure 3.15: Setting the stage...

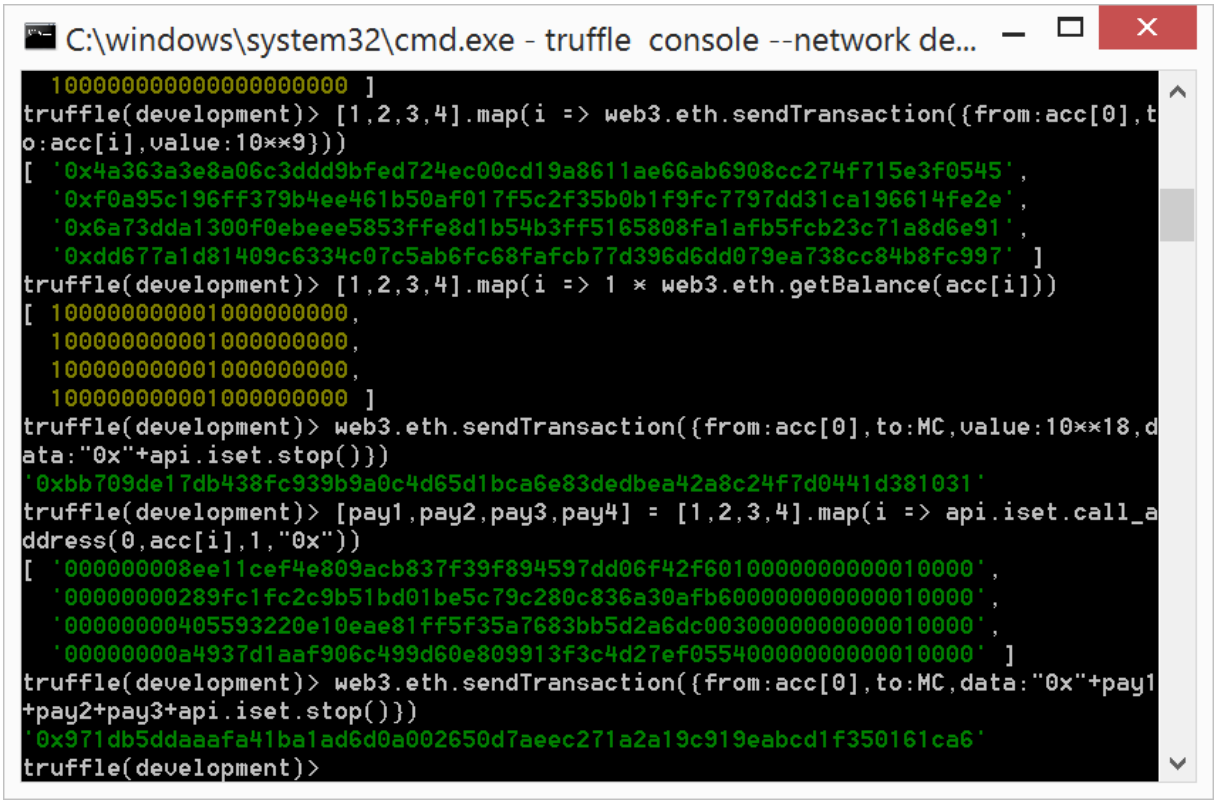

Figure 3.16: A batched transaction.

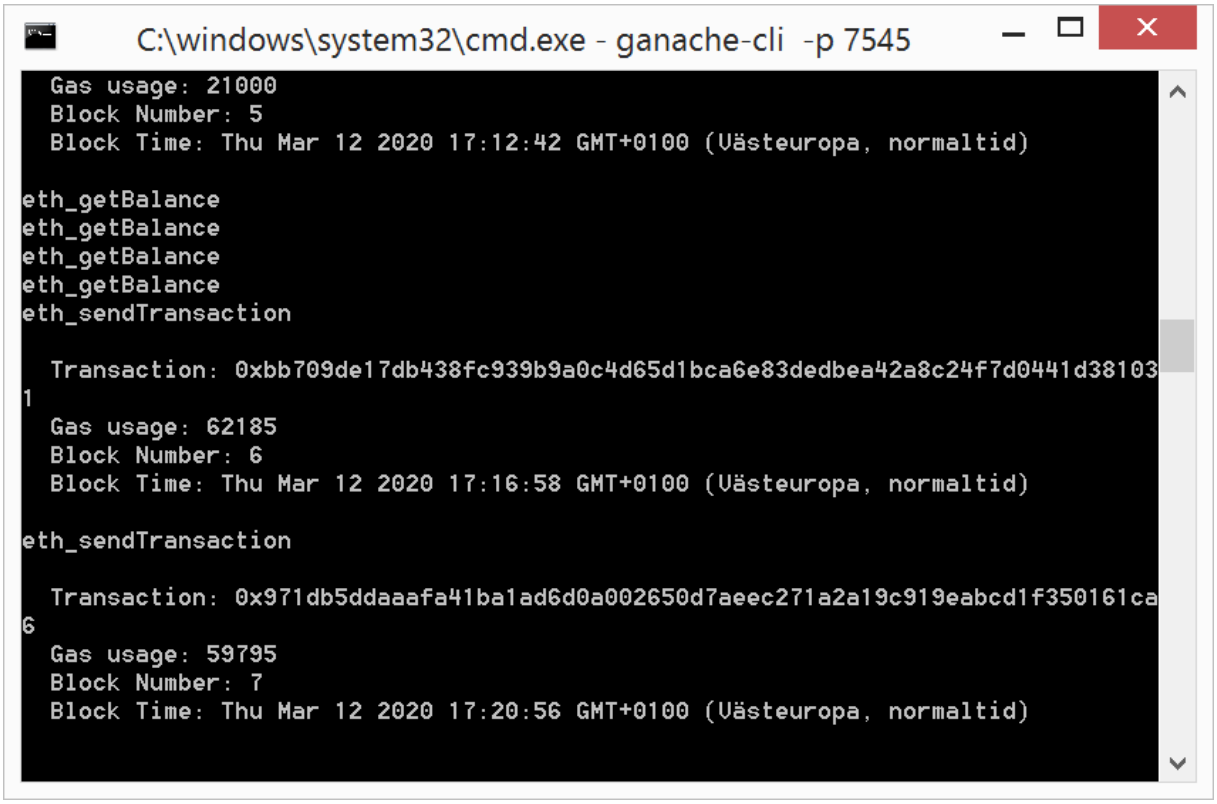

Figure 3.17: It's suspiciously cheap given fixed overheads should dominate when you batch so few transactions...

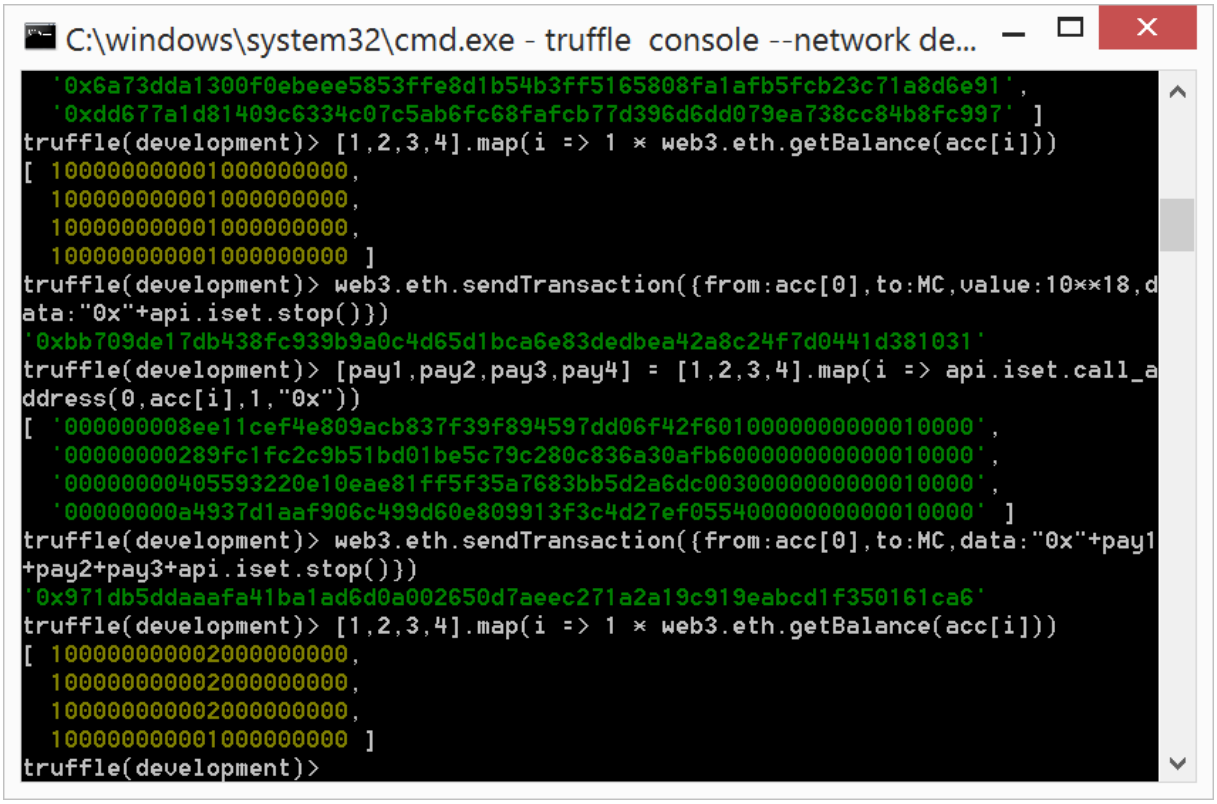

Figure 3.18: Looks like I only paid the first three intended recipients!

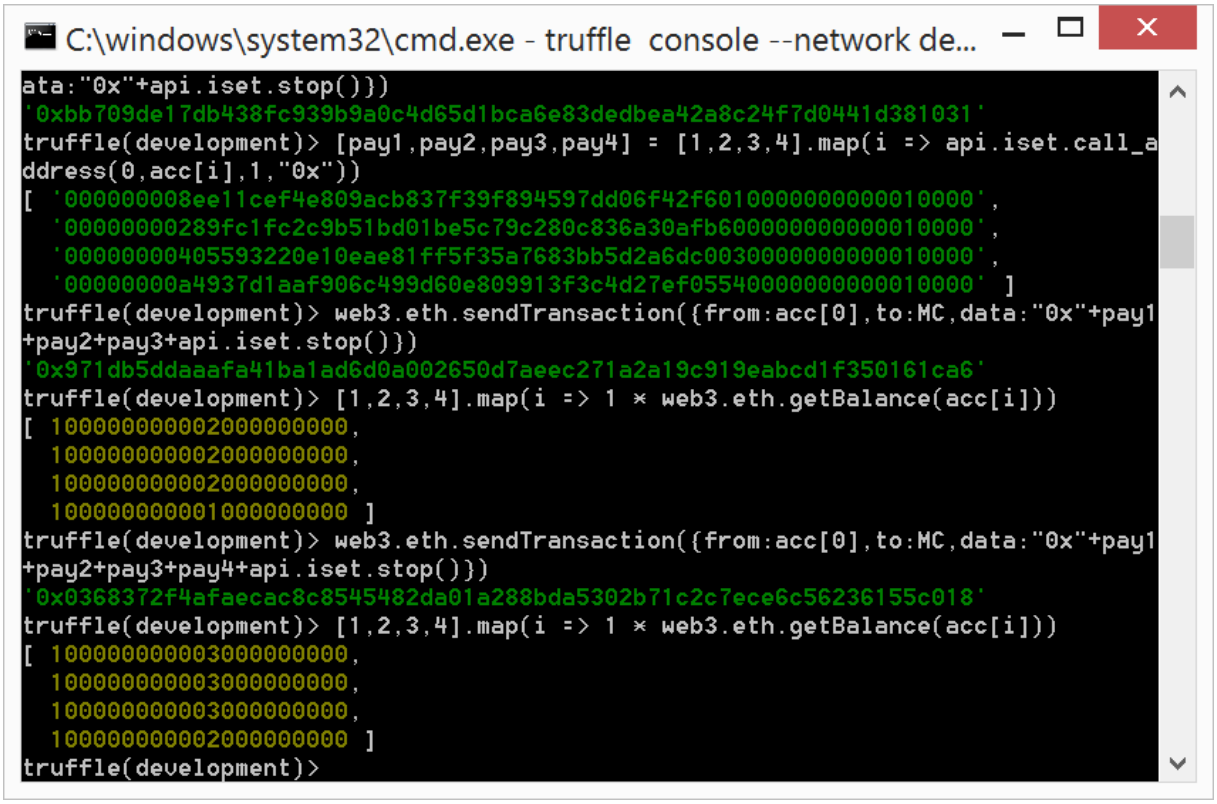

Figure 3.19: Correctly paying all four recipients.

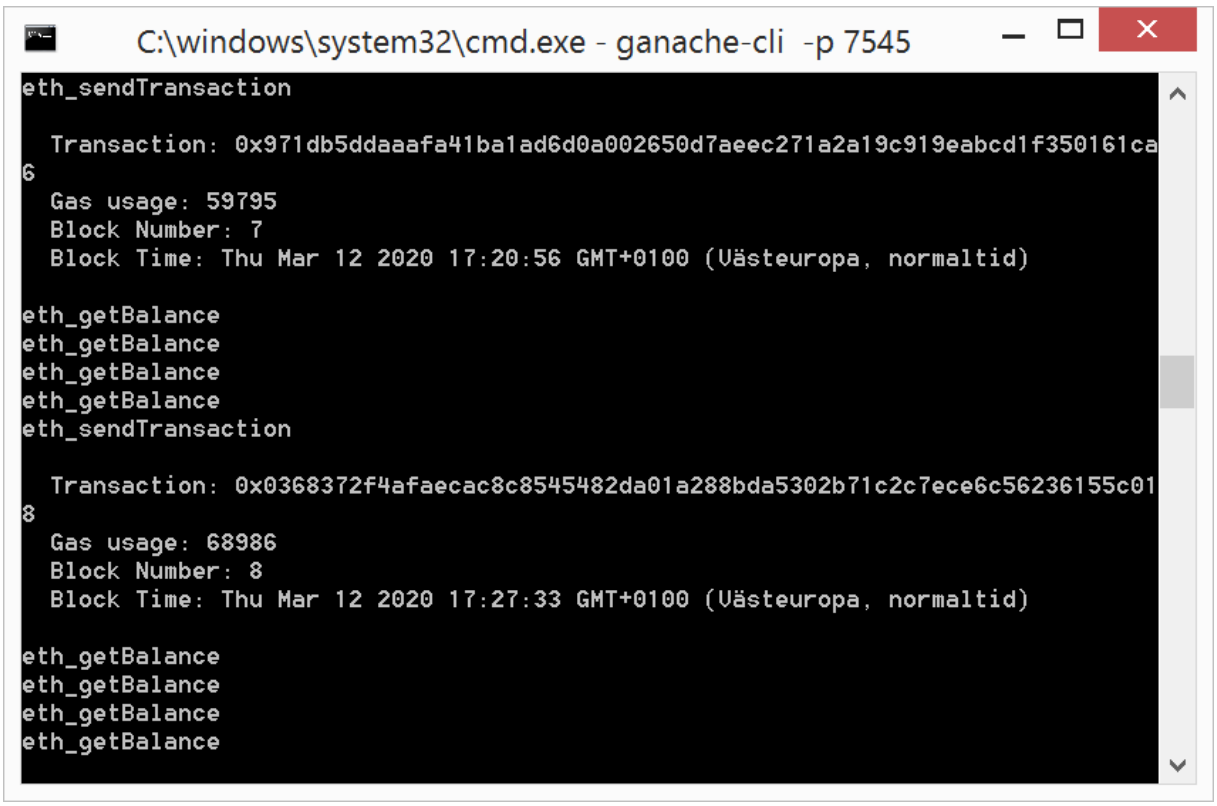

Figure 3.20: It's still cheaper than sending the payments separately!

3.3 Interfacing software

For practical use of MultiCall it must be convenient for Ethereum users to generate and send scripts to the smart contract. That requires either changes to existing Ethereum blockchain graphical user interfaces, or the creation of a new one. Because it's efficient for multiple signatories to share a single call to MultiCall using its signed script execution feature, I propose that there should be an intermediate server which accumulates signed scripts and then sends them in one transaction. The structure of Ethereum interface software might then look like the illustration below. Adding an intermediate transaction collection stage would be an architectural change to the distributed ledger ecosystem, and could be an interesting subject of future study. The proposed new structure is shown in figure 3.21.

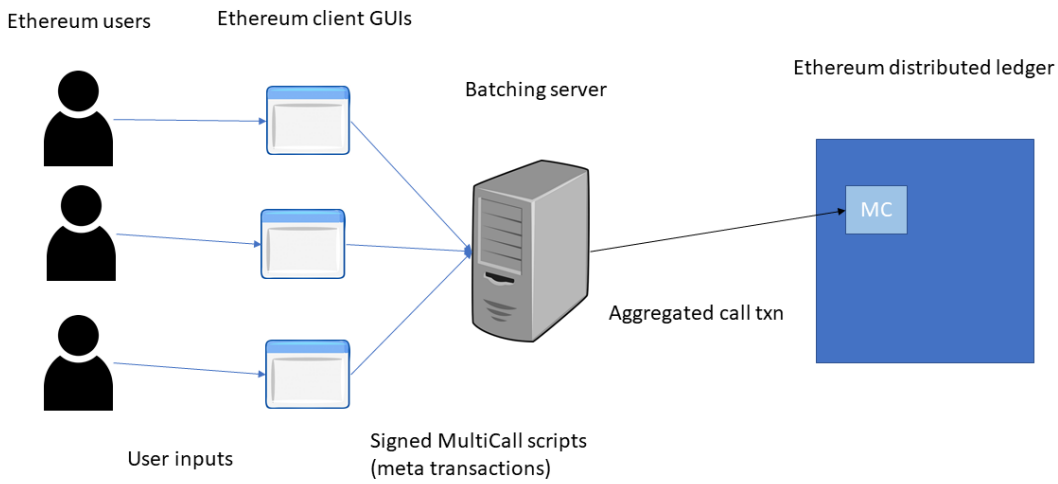

Figure 3.21: Proposed blockchain interface architecture taking advantage of MultiCall's novel features. User inputs are converted to meta transactions in each user's GUI, then sent to a shared batching server, aggregated and uploaded to the blockchain in a single transaction.

4
Background

The details of the consensus algorithm for the Ethereum blockchain are not relevant to this thesis; the MultiCall interpreter runs within Ethereum's state transition function and may assume the distributed database is consistent. Here follows a description of the Ethereum ledger's semantics.

The Ethereum protocol maintains a distributed database mapping 160-bit addresses to ether balances and a nonce, which is set to 1 on account creation and incremented when transactions are sent from that address in order to protect against replay attacks. In addition, an address may map to a smart contract. Smart contracts consist of immutable contract text and a private persistent storage space which maps 256-bit indices to 256-bit values.

The database may be updated by sending transactions, i.e. signed messages which are meaningful to the protocol. There are two types of transaction in Ethereum: call and contract creation transactions. Both types take as their arguments a gas price, gas limit, ether value, calldata bytestring and nonce. Call transactions differ in that they also take a callee address. All transactions have a public key sender, and must be signed by the corresponding private key to be valid. The nonce of the transaction must equal the nonce of its sender in the distributed ledger; this protects against replay attacks. The gas limit restricts how much gas can be spent by a transaction and thus how much ether can be charged in fees. Attempting to use more gas than is available triggers an exception. All exceptions raised in a smart contract call revert the call's execution; if such a reversion occurs in the context of the top-level call, then the transaction only has the effect of paying fees to the miner and incrementing the nonce.

Contract creation transactions interpret the given calldata as EVM bytecode; this bytecode performs the initialization of the smart contract to be created and returns its text, transferring the given ether value to the contract's address. Call transactions invoke the callee address. If the callee is not a smart contract, no execution occurs and the ether value specified is transferred to the callee's balance. If the callee address is mapped to a contract then the contract text is interpreted as EVM instructions and executed in the context of the contract.

The EVM is a stack machine with 256-bit words. While executing, the contract has its own volatile memory and stack; there are EVM instructions to read its calldata, read the text of other contracts, and read and write to the contract's persistent storage. All instructions pop and push an integer number of 256-bit stack elements. Execution always starts at byte 0 of the contract text.

Opcodes are one byte; the constant push instructions PUSH1-32 also have 1-32 bytes of immediate argument data containing the constant which they push onto

the stack. In addition to the usual stack manipulation, arithmetic and jump instructions, the EVM also has smart contract-specific instructions. The instruction `CALL` invokes a given smart contract with given calldata and transfers an amount of ether to it. This is the primary means of transferring ether. The instruction `CREATE` creates a contract with a given initialization code bytestring and transfers a given amount of ether to it. These two instructions may be used to emulate call and contract creation transactions, respectively. Whether created by a transaction or instruction, contracts are given a pseudorandom address according to their creator address and the creator's nonce. The creation address for a given creator and nonce is deterministic; this can be used to transfer funds to a contract's balance before the contract is created. In theory, calls from a contract could also be spoofed by finding a private key whose corresponding public key is the contract's address; I have hardened the interpreter against this attack. The `CALLER` and `CALLVALUE` instructions push the caller's address and the ether value deposited, respectively. They are used for authenticating identity and payment. The instructions `LOG0`-`LOG4` record a log message in the block containing the transaction that executed them, with 0 to 4 256-bit log topics and a variable-length message bytestring. Ethereum blocks are structured so that a client can cheaply search for log messages from a particular contract and with a particular set of topics. That allows contracts to record events such as a payment made from a sender to a given recipient, such that the recipient can cheaply search for payments made to them from the given sender. The details of how this is implemented are described in the Ethereum Yellow Paper [31].

Below is a summary of notable gas costs of the Ethereum protocol. Knowing them is useful for reasoning about and optimizing smart contracts. Transactions each have a fixed cost of 21000 gas; an additional 68 gas is charged per non-zero byte of calldata, and 4 per zero byte. Zero bytes are cheaper because they supposedly allow more efficient compression. Whether that is actually true in practice is not my concern; my objective is to minimize gas usage.

- **CREATE**: Creates a contract. Cost: 32000 gas + 200 gas per byte of contract text created. The gas cost of the initialization code is counted separately.
- **SSTORE**: Writes a word to persistent storage. Cost: 5000 gas + 15000 if the word is allocated (set to a nonzero value from 0), with a 15000 refund if it's deallocated.
- **SLOAD**: Loads a word from persistent storage. Cost: 200 gas
- **CALL**: calls an address, transferring a given amount of ether to it. It also takes a gas limit; if the called contract attempts to use more gas than the limit, it throws an exception instead. Cost: 700 gas + 6700 gas if a nonzero amount of ether is transferred. The gas cost of the contract invocation is counted separately.
- **EXTCODECOPY**: copies a range of bytecode of the contract at the given address into volatile memory. This can be used for cheap read-only storage. Cost: 700 gas + 3 gas per 256-bit word copied.
- Stack duplication, pop and constant push instructions each cost 3 gas.
- **ADD**: Pops two values and pushes their sum. Cost: 3 gas. This is representative of the cost of arithmetic and logical instructions in general.
- **JUMP** and **JUMPI** implement unconditional and conditional jumps. They

cost 5 and 8 gas respectively. They consume a stack value as their jump destination; there is no penalty for dynamic jumps, as they are the only type available.

Note that the smart contract-specific instructions are significantly more expensive than arithmetic and control flow. This suggests that the interpretation overhead of an interpreter contract which performs a series of calls, creates and storage writes could be relatively low. Compare the 7400 cost of a `CALL` instruction which makes a transfer to the 21000 fixed transaction cost. Even if the interpretation overhead and data cost of submitting the call instruction to an interpreter were 3000 gas, the fixed cost of emulating a call transaction would still be less than 50% of the per-transaction fixed cost.

5

Implementation

Because the fixed gas cost of a smart contract-initiated call is significantly lower than that of a call transaction, I conclude that a transaction-emulating interpreter running on the EVM may provide significant savings. The question arises how to write such a program. Writing hexcode directly is in general tedious and error-prone; it can be dismissed out of hand. In practice, smart contract authors write in high-level languages targeting EVM bytecode. I primarily considered two preexisting languages in which to implement the interpreter: Solidity [17] and LLL [15]. Instead, I chose to develop a code-generating EDSL in Haskell and used it to implement the interpreter. While I have not written alternate interpreters in Solidity and LLL and thus cannot conclusively state that it was the best choice, here follows my rationale for choosing the EDSL.

5.1 Solidity

Most contracts today are written in *Solidity*, a C++ and Javascript-inspired language intended to make it easy for developers to begin writing smart contracts. It uses an object-oriented programming model, where each contract is treated as an object which exposes a number of methods. Advocates of C++ tout its zero-cost abstractions; Solidity's abstractions are far from zero-cost. The format for method calls is called the Solidity ABI, or application binary interface, and it is the primary means by which contracts communicate today. It specifies that each method call corresponds to an Ethereum call, with a 32-bit method identifier placed in the first four bytes of calldata. Multiple interactions with a contract require multiple calls. That is inefficient: even if the calls are sent from a contract, they cost at least 700 gas per call and any storage data used must be refetched from storage into the callee's volatile memory each time. Method arguments are also packed inefficiently, with all fixed-size types taking up 256 bits regardless of size. The 8-bit signed int `-1` is represented as 32 `0xff` bytes in calldata, at a cost of 2176 gas if sent from a transaction. Furthermore, the Solidity compiler has had a number of serious bugs: an example is *ConstantOptimizerSubtraction*, described as "In some situations, the optimizer replaces certain numbers in the code with routines that compute different numbers" [14]. Solidity contracts have also been the subject of several multi-decamillion US dollar thefts and losses because of programming errors[1]. It is the opinion of the author that a language targeting a highly performance

[1] `https://www.bloomberg.com/features/2017-the-ether-thief/`, `https://hackernoon.com/parity-wallet-hack-2-electric-boogaloo-e493f2365303`

and security-intensive environment such as the Ethereum blockchain should provide either low-level control of instructions and a simple compilation procedure that the author can trust and easily audit, superior performance to hand-rolled assembly, or guarantees of security properties. Solidity provides none of the three. I chose not to use it.

5.2 LLL

LLL [15] is a low-level Lisp-like language (hence the name). It is lower-level than Solidity, and does not provide its own memory allocation or natively support the ABI. Like Lisp, it allows the author to define macros to generate code. That was enticing, as generation of interpreter jump tables benefits from automation. Ultimately, I chose to write an EVM code-generation EDSL in Haskell, as that provides access to the combinators and type system of a mature general-purpose programming language.

5.3 The C(odeGen) monad

5.3.1 Foundations

Basic familiarity with Haskell is assumed. However, it is worthwhile to note the language-specific features central to the implementation of the code-generating EDSL. An EDSL [9] is a domain-specific language embedded within another programming language; in this case it is a monad which generates EVM bytecode embedded within Haskell, a typed lazy purely functional language. Monads are any type which implement the `Monad` type class (and satisfy the monad laws [30]):

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

The `(>>=)` operator is pronounced bind. A monad [30] is a type of value representing a computation which may return a value or be sequenced, where the output of the first monad value is passed to a continuation function which returns the next. The result of sequencing monadic computations is to apply the effect of the first value, and then the second. Values are returned with the `return` function, and monadic computations are sequenced using the monadic bind operator `(>>=)`.
Monads are a useful means of expressing effectful computations, and are thus widely used in Haskell. There are many monad types; a notable example is the `IO` monad, which is used to represent impure computations as expressions in Haskell's pure semantics. Not all monads are impure; for example state monads modify an internal state deterministically and can be interpreted in terms of a pure function. Here is an instance for a state monad:

```
--This defines the state datatype
data State s a = State (s -> (a,s))
```

```
--This defines the state monad's interpreter
interpret :: s -> State s a -> (a,s)
interpret s (State f) = f s
--This defines monadic operations on the datatype
instance Monad (State s) where
    return a = State (\s -> (a,s))
    sm >>= g = State (\s -> let (a,s') = interpret s sm
                            in interpret s' (g a))
```

State monads are implemented purely; in the definition of bind, the state which is mutated by the monad `sm` is passed as a function argument to the the pure `interpret` function; the continuation `g` also accepts `sm`'s return value `a`.
The functions `put` and `get`, defined below, encapsulate the basic functionality of the state monad.

```
put :: s -> State s ()
put s = State (\_ -> ((),s))
get :: State s s
get = State (\s -> (s,s))
```

They allow the user to read and write to a state value in an imperative style while maintaining functional purity. For example, the following code provides an incrementing counter. By using a more complex record type as the state, multiple variables can be tracked in a state monad.

```
counter :: State Int Int
counter = do n <- get
             put (n + 1)
             return n
```

Haskell achieves overloading of operations such as monad bind and return via its type class system. A type class specifies an interface of class methods whose implementation is determined at compile time depending on their inferred type at the use site. For example, the `Num` class is used for numeric types:

```
class Num a where
    (+) :: a -> a -> a
    (-) :: a -> a -> a
    (*) :: a -> a -> a
    fromInteger :: Integer -> a
    --Additional methods omitted
```

Note that in `(+)`, `(*)` and `(-)` applications, the implementation to use can be inferred from the type of their arguments. For a `fromInteger` application, the function used to convert from the `Integer` type to `a` must instead be inferred from the type of the application's return value. That would be impossible in a dynamically typed language with OO-style overloading such as Javascript. Integer literals are syntactic sugar for applications of `fromInteger`, and may be of any type which

supports `Num`.
In the same way that numeric operator implementations are selected based on their type, the monad operator implementations depend on the type of monad value used and expected. Monad values can be created using do notation, which desugars expressions such as

```
do a
   x <- b
   c x x
```

to a sequence of binds and returns:

```
a >>= (\_ -> b >>= (\x -> c x x))
```

The `(>>)` operator (used in the sections ahead) sequences two monads, ignoring the return value of the first:

```
m1 >> m2 = m1 >>= (\_ -> m2)
```

5.3.2 Application to the EDSL

The `C` monad EDSL is implemented as a state monad which modifies a record of type `Env` containing a number of fields relevant to code generation. Here is its definition:

```
data Env = Env {label_ctr   :: Int,
                text_offset :: Int,
                syms        :: Syms,
                future_syms :: Syms,
                prog_text   :: Program,
                stackOffset :: Integer
                }
         deriving (Eq,Ord,Read,Show)
```

The `Program` type synonym defines program bytecode to be a linked list of integers. Linked lists are cheap to prepend one element at a time, so they are a natural choice for a bytecode accumulator. The bytecode is accumulated in reverse order and then reversed at the end, in order to make appending bytes cheap.

```
type Program = [Integer]
```

The `Syms` type synonym defines the mapping from labels to integers in `syms` and `future_syms` to be a linked list of pairs of label and integer. Label lookups are $O(n)$ in the number of mappings, so a more efficient data structure might be used in a code generator with stricter performance requirements such as a JIT compiler.

```
type Syms = [(Label,Integer)]
```

The `C a` type is the type of code-generator monads which return a value of type `a`. Each constructor corresponds to an action which can be performed by the monad's interpreter function `interpret`; the type defines the monad's primitive instruction set from which other functionality is derived.

```
data C a where
```

```
  NewLabel       :: C Label
  (:=)           :: Label -> Integer -> C ()
  Byte           :: Integer -> C ()
  GetTextOffset  :: C Int
  SetTextOffset  :: Int -> C ()
  LookupLabel    :: Label -> C Integer
  FullEnv        :: C Env --Used for debugging.
  Return         :: a -> C a
  (:>>=)         :: C a -> (a -> C b) -> C b
  GetStackOffset :: C Integer
  SetStackOffset :: Integer -> C ()
```

The behaviour of the `C` state monad is defined by its interpretation function:

```
interpret :: C a -> (a,Env)
interpret c = let (a,env) = interp envInit c
                  envInit = Env {label_ctr = 0,
                                 text_offset = 0,
                                 syms = [],
                                 future_syms = syms env, --TIME TRAVEL!
                                 prog_text = [],
                                 stackOffset = 0,
                                 memOffset = 0
                                 }
              in (a,env{syms = reverse $ syms env,
                        prog_text = reverse $ prog_text env})
interp env c = case c of
  ... --one case for each C datatype constructor
```

Monad class instance:

```
instance Monad C where
  (>>=) = (:>>=)
  return = Return
```

And Num class instance:

```
instance Num (C a) where
  (+) a b = b >> a >> byte op_ADD >> incSO (-1) >> return undefined
  (-) a b = b >> a >> byte op_SUB >> incSO (-1) >> return undefined
  (*) a b = b >> a >> byte op_MUL >> incSO (-1) >> return undefined
  fromInteger n = push n >> return undefined
```

The `(:>>=)` and `Return` constructors are interpreted by `interp` as ordinary state monad bind and return. The function `push :: Integer -> C ()` generates an appropriately-sized EVM constant push instruction. For integers larger than 32 bytes, it errors. For convenience and simplicity, an ordinary Haskell exception is thrown; future versions of the monad may represent and handle errors itself.

Do notation and `Num` syntactic sugar allows C monad programs to be expressed concisely. For example, the following program generates EVM bytecode to increment

a word at address 0 in memory by 5, and then copy it to address 32:

```
do mstore 0 (mload 0 + 5)
   mstore 32 (mload 0)
```

When a C monad program runs, it modifies a program text `prog_text`; a stack offset indicating the net stack effect of code `stackOffset`; a text offset indicating the byte offset of the next byte `text_offset`, and a map of Label values to Integer values `syms`. The `prog_text` field of the state record returned from running `interpret` on a C monad is the bytecode it generates. It is modified via the byte function, which pushes a single byte (assumed to be a non-negative integer) to the program text:

```
--A case in the interpretation function interp:
Byte n -> ((), env{text_offset = text_offset env + 1,
                   prog_text =
                        (if n > 255
                         then error $ "Byte too big: " ++ show(n,env)
                         else n) : prog_text env})
```

The `env` variable is the `Env` state argument passed to the `interp` interpretation function. The meaning of the pair returned is that `Byte n` returns the empty tuple `()`, increments the `text_offset` field by 1, and pushes `n` to the `prog_text` accumulator if `n` is less than or equal to 255. If it is greater than 255, it errors. The result of sequencing two C monad values `a` and `b` is another C monad which generates the concatenation of the bytecode generated by `a` and `b`.
The `Label` type is used to refer to jump destinations and other static constants such as string lengths:

```
data Label = LNamed String | LAnon Int
```

They may either be named or anonymous. Unique anonymous labels are generated by incrementing a counter in the C state:

```
--This is the interp case for NewLabel, the monad constructor for
--generating new anonymous labels
NewLabel -> let n = label_ctr env in
            (LAnon n,env{label_ctr = n + 1})
```

Labels can be 'placed', mapped to the current text offset using the place function. That is used to mark the location of jump destinations and strings in bytecode.

```
place :: Label -> C ()
place l = do n <- GetTextOffset
             l := fromIntegral n
```

Note that the `(:=)` operator is not a syntactic primitive or pseudocode, it is an ordinary constructor; to distinguish them from non-constructor operators such as `(+)`, infix constructor names all begin with colon in Haskell. The `(l := n)` C monad constructor adds a mapping from the label `l` to the integer `n` in the field `syms`:

```
--This is the interp case for (:=); it adds the mapping (label,n)
--to the list of mappings syms env using the cons operator (:)
label := n -> ((),env{syms = (label,n) : syms env})
```

The integer value to which a label maps can be accessed as an EVM constant instruction via the `labelExpr16` function:

```
labelExpr16 l = do n <- LookupLabel l
                   pushN 2
                   byte $ fromInteger $ n `div` 256
                   byte $ fromInteger $ n `rem` 256
```

The `div` and `rem` operators implement integer division and remainder respectively. `labelExpr16` uses the `C` monad constructor `LookupLabel` to look up the integer value to which the label maps; pushes the EVM opcode for `PUSH2`, which expects 2 bytes of immediate argument data, and then outputs the label value as two bytes.

An interesting feature of the monad is it needs to output values dependent on the label-integer mapping before it is created, for example when jumping forward:

```
do l <- NewLabel
   jump_ (labelExpr16 l) --Dynamic jump to the offset of l
   ... --Some intervening code
   place l --The offset to which to jump, set after its use!
```

This could be implemented by collecting the integer values of labels in one pass, tracking the locations of uses of labels and then inserting the integer values at the use points in a second pass. However, I found a more elegant way to express it using Haskell's laziness. As Haskell is a lazy language, expressions may be defined and referred to before they are evaluated; they may even be defined in terms of themselves. In the interpret function, I set the `future_syms` field of the initial state to the `syms` field of the final state; the `future_syms` field is used for label lookups.

This may seem like magic but it is not. Haskell is not a formula solver; if there were a true cyclical dependency where an expression's evaluation depended on its own value it would deadlock. Because the number of bytes pushed by `labelExpr16` is independent of the label value, the `bytes_offset` field used to calculate labels is unaffected by the value of the labels and thus does not form a cyclical dependency. If you were instead to push a variable number of bytes depending on the label value, the computation would deadlock.

5.4 More C monad functionality

5.4.1 Jumpdest placement

The function `place_jumpdest` outputs a `JUMPDEST` opcode and places the given label at its offset. Jumpdests mark valid jump destinations in order to ease static analysis and optimization of EVM bytecode. Jumping to an instruction other than a jumpdest raises an exception.

```
place_jumpdest l = place l >> byte op_JUMPDEST
```

5.4.2 EVM instruction combinators

Instructions which consume N stack values are represented as functions which take N C monad arguments. The fully applied function sequences each argument and then pushes the relevant opcode using `byte`. The effect is that abstract syntax trees of instructions are pushed to bytecode in depth-first order, with the C monad acting as a simple non-optimizing compiler. This straightforward transformation allows both a convenient C language-like syntax for EVM code with nested expressions, with reduced risk of insidious compiler bugs and thus reduced need for bytecode auditing. Note that arguments are sequenced in reverse order; that is because EVM instructions expect their arguments to be pushed to the stack in reverse order. For example, the bytecode for `(3 - 5)` first outputs the `push 5` instruction, then `push 3`, then the `SUB` opcode. That leads to side-effects in nested expressions occurring in the reverse of the order the programmer familiar with imperative languages might expect. The `incSO` statements at the end of definitions modifies the `stackOffset` field in the monad environment to track the number of elements pushed or popped by the instruction. Here is an example instruction combinator definition, for the EVM instruction `MSTORE`, which takes an index and a value and stores the value to the index in memory:

```
mstore :: C() -> C() -> C()
mstore = i2_void op_MSTORE
```

The expression `op_MSTORE` is a byte constant defined as 0x52; `i2_void` is a combinator for generating instructions which take two arguments and return none given their opcode:

```
i2_void o a b = b >> a >> byte o >> incSO (-2)
```

As we can see, arguments are evaluated in reverse order. The stack offset is decremented by 2 because the function pops two stack values without pushing any.

5.4.3 Void

The void value is the empty C monad, which pushes no bytes and has no other effect.

```
void :: C ()
void = return ()
```

It is convenient to use `(void + e)` to increment the top of the stack by some expression `e`. Note that `(void - 1)` does **not** decrement the top of the stack as might be expected, as `1` is placed at the top of the stack and thus becomes the value subtracted from. Decrementing or dividing the top of the stack requires an additional swap instruction:

```
decrement = do
 1           --Pushes 1 to the stack
 swap 2      --Swaps the top stack element with the second
 void - void --Just outputs the SUB opcode and reduces stackOffset
```

5.4.4 Control flow

The functions `jump` and `jumpi` respectively implement jumps and conditional jumps to the offset of a given label. They use the labelExpr16 function to obtain the label's value:

```
jump l = jump_ (labelExpr16 l)
jumpi cond dest = jumpi_ (labelExpr16 dest) cond
```

The raw `jump_` and `jumpi_` functions use arbitrary expressions as jump destinations and should be used with care. The function `jump_` is useful for implementing an interpreter's dispatch code.

The `ifnot`, `iff` and `doWhile` combinators provide structured programming patterns:

```
ifnot cond code = do
  so <- getSO
  skip <- label
  jumpi cond skip
  code
  place_jumpdest skip
  so' <- getSO
  incSO (so - so')
iff e = ifnot (iszero e)
dowhile cond body = do
  so <- getSO
  loop <- jumpdest
  body
  jumpi cond loop
  so' <- getSO
  incSO (so - so')
```

The `getSO` and `setSO` wrapper guarantees that the code within the conditional clause does not affect the stack offset of the outside code. The stack offset is calculated in a naive fashion where the stack effects of all sequential code is added, so this is necessary to avoid confusing the stack offset system if you for example run a function which pushes stack elements and never returns (such as the dispatch function of an interpreter) within the if clause. The programmer must ensure that the conditionally executed clause leaves the stack offset unchanged if it returns.

The ease with which higher-level patterns can be implemented in terms of functional combinators is an advantage of using an EDSL; you essentially get a Turing-complete macro system equipped with the features of the host language. The combinator `ifnot` is less intuitive to use, but it's slightly more efficient than `iff`, which is implemented in terms of `ifnot` with an additional instruction `iszero` which negates a Boolean condition. If clauses such as `"if (a > 5) f()"` could be compiled to `ifnot` by negating the condition: `"ifnot (6 > a) f()"`, but that would require the monad to preserve information about the structure of expressions. That may be of interest in a future compiler, but the initial focus of the EDSL is simplicity.

5.4.5 Stack variables

Stack elements are duplicated and swapped with the `dup` and `swap` functions, which generate the appropriate opcode given the desired stack offset to duplicate or swap with the top element. The stack can be used to implement local variables. As the stack offset constantly changes as stack values are pushed and popped, manually managing stack offsets used as variables is difficult. Like manually calculating opcodes, it is tedious and error-prone, precisely the sort of task best left to computers. The `SVar` datatype automates the process of tracking the stack offset:

```
data SVar = SVar{originalOffset::Integer,varOffset::Integer}
```

New `SVar`'s can be declared with the `newVar :: C () -> C SVar` function, which pushes the given expression to the stack and records its offset. Variables may be accessed with the `dupVar` and `swapVar` functions, which calculate the current stack offset of the variable from the difference between the `stackOffset` field at creation and the current one. The top `n` elements of the stack may be aliased to a list of stack variables with the `declareVars n` command:

```
--Declare the top 3 stack elements to be x, y, and z
--Could be used in a subroutine to accept arguments.
[x,y,z] <- declareVars 3
```

5.4.6 Global variables

The `GVar` (Global Variable) datatype provides mnemonics for offsets into memory and storage:

```
data GVar = Memory Integer | Storage Integer
```

The `load` and `store` operations are overloaded to manage both memory and storage globals. The `(+=)` and `(-=)` functions are syntactic sugar for global variable increments and decrements, implemented in terms of load and store. SVars and GVars are an example of extending the monad with a new type which communicates static information. Other such types could be added, such as a subtype of SVars which may not be mutated, or labels which point to subroutines which push and pop certain types.

5.4.7 Contract initialization code

Contracts are not uploaded to the blockchain as bytecode directly. Instead, contract creation bytecode is uploaded, which may perform some computation or side effects and then return the text of the final code. To easily make contract creation code, I have defined the `distrCode` combinator which takes the initial side-effecting code and the final code and generates the initialization code:

```
distrCode :: C() -> C() -> C()
distrCode initCode finalCode = do
  --Statically obtain the program text of the final code
  let text = prog_text $ snd $ interpret finalCode
```

```
  --Run the init code
  initCode
  --Return the string defined by the named label "FINALCODE"
  returnString "FINALCODE"
  --Places the program text in bytecode; defines FINALCODE_ptr as the
  --offset and FINALCODE_len as the length, used by returnString
  defByteString "FINALCODE" text
```

It is now feasible to generate a smart contract using the C monad. Below is the definition of the creation code for a contract which simply writes the first word of its calldata to storage and starts with the value 1 in its storage:

```
testStorageWrite = distrCode (sstore 0 1) (sstore 0 (calldataload 0))
```

The expression `calldataload n` loads a 256-bit word from the calldata at byte offset `n`; `sstore ix e` writes the value `e` to the persistent storage index `ix`. The contract `testStorageWrite` was used to test the interpreter's call functionality.

5.4.7.1 Generated bytecode example

When compiled to a file, the initialization code `testStorageWrite` generates the following EVM bytecode:

```
0x60016000556100066100146000396100066000f3600035600055
```

Divided into individual instructions for readability, it looks like this (`--` denotes a comment):

```
--sstore(0,1)
60 01 --push1 1
60 00 --push1 0
55    --sstore
--codecopy(0,FINALCODE_ptr,FINALCODE_len)
61 0006 --push2 6, FINALCODE_len
61 0014 --push2 20, FINALCODE_ptr
60 00
39      --codecopy
--return(0,FINALCODE_len)
61 0006
60 00
f3
--FINALCODE_ptr:
--it = calldataload(0)
60 00
35
--sstore(0,it)
60 00
55
```

Each instruction opcode is one byte; the 16 opcodes `0x60-0x7f` are the push instructions `PUSH1-PUSH32`. Push instructions take a 1-32 byte sequence of immediate

argument data to push onto the stack as a single number, and are used to provide constant arguments to other instructions. In the above code, only `PUSH1` (`0x60`) and `PUSH2` (`0x61`) are used. Each line is one instruction; the bytes following push opcodes on the same line are the instruction's immediate arguments.
The initialization code first writes `1` to persistent storage, then returns the final code of the contract to be created. Recall that the bytestring returned by the initialization code becomes the text of the created contract. Storage writes made during creation persist in the created contract's storage. The first three instructions implement the storage write and are the result of compiling the expression `sstore 0 1`. This corresponds to the argument `initCode` in the definition of `distrCode`. Note that arguments are pushed in reverse order.
Return takes a memory pointer and length as argument, and as such any data must be copied into memory before it can be returned. The next four instructions copy the final code from the offset given by the label `"FINALCODE_ptr"` to offset `0` in memory. The number of bytes copied is `labelExpr16 "FINALCODE_len"`. Then the following three return the given data. These seven instructions correspond to the expression `returnString "FINALCODE"` in the definition of `distrCode`. The `returnString` and `defByteString` functions use the convention that a static bytestring identified by the label `n` has its offset in code and length in bytes defined by the labels `(n # "_ptr")` abd `(n # "_len")`, where `(#)` denotes string concatenation for named labels. The expression `"FINALCODE"` in the definition of `distrCode` is a label; Haskell's syntax has been overloaded to allow named labels to be created from string literals.
After the return opcode `0xf3` execution ceases; the last four instructions are final code which is returned. The expression `defByteString "FINALCODE" text` places the bytestring `text` in the bytecode and maps the labels `"FINALCODE_ptr"` and `"FINALCODE_len"` to refer to its offset in the program text and byte length. When creation is complete those four instructions become the program text. They correspond to the expression `sstore 0 (calldataload 0)`. The end result is a smart contract which writes the first word of calldata it receives to storage, which made it possible to debug whether MultiCall was sending correct calldata by checking the contract's storage.

5.5 Running generated code

While I have shown how to write a simple smart contract with the C monad, to test that the contract works correctly it must be run on a version of the Ethereum blockchain. Debugging a contract requires repeatedly recompiling it, deploying it, and sending transactions to it; doing so on the Ethereum main chain would be prohibitively expensive. Therefore smart contract developers use private chains to debug their contracts. The tools I have selected for doing so are *ganache-cli* [21] and *truffle* [20]. Ganache is a program which provides a private Ethereum node; truffle is a Javascript console specialized for Ethereum which allows the programmer to link to the node and make rpc calls to it. I use the *web3* module for doing so, which is the standard approach. I have also written a Javascript utility library *util.js* for easier deployment of code and a library *MC_api.js* for interfacing with the MultiCall

contract.
Here is an example of how to compile, deploy and test a simple smart contract using my tools. I will use the testStorageWrite contract shown earlier. From the Haskell shell, I run the IO program:

```
hackyDeploy (tempDir ++ "TSW") testStorageWrite
```

The function `hackyDeploy :: FilePath -> C () -> IO ()` generates bytecode from the given C monad and writes it to the given file path. I write it to a file named TSW in `tempDir`, defined to be the directory used by truffle on my machine. In the truffle shell,

```
TSW = util.deploy("TSW")
```

uploads the contract to the private chain and binds its address to the variable TSW. Its current value at storage index 0 is 1, the value set at creation. In the truffle shell, the command:

```
web3.eth.getStorageAt(TSW,0)
```

returns `"0x01"`. The web3.eth.sendTransaction command sends a call to the contract:

```
web3.eth.sendTransaction({from: web3.eth.accounts[0],
                          to: TSW,
                          data: "0xdeadbeef"})
```

Its new storage value has `deadbeef` in its top 4 bytes, indicating it works as intended.

5.6 Low-level design of the interpreter

While it is now possible to deploy and run contracts generated using the C monad, problems specific to interpreter design remain. It may not be immediately obvious to the reader how to implement an interpreter in the EVM. Pertinent questions include how instructions are to be represented, where they are stored during execution, how instruction dispatch is implemented and how the interpreter is initialized. Multiple possible approaches exist; for example, instructions could be represented as abstract syntax trees. That might be efficient, as the interpreter could exploit information about the syntax tree structure to use specialized interpretation subroutines. I chose a simpler approach, described below.

5.6.1 Instruction format

Invocations of the MultiCall interpreter consist of the concatenation of a number of instructions, executed sequentially. MultiCall instructions consist of a one-byte opcode, followed by a number of immediate argument bytes. To interpret instructions, MultiCall reads them from the calldata at the offset of the PC's (program counter's) current value at the top of the stack, masks out the highest byte containing the opcode, adds the byte to the offset of a jump table stored in code, and jumps to it:

```
dispatch_mc = do
  calldataload $ dup 1
  jump_ $ labelExpr16 "JUMPTABLE_MC" + (dup 1 ! 0)
```

The expression `(dup 1)` duplicates the top stack element, which is initially the program counter; the bang operator is the EVM's byte indexing instruction `BYTE`, renamed to be reminiscent of Haskell's list indexing operator. `dispatch_mc` pushes the word containing the opcode onto the stack. This is to allow instructions to parse their immediate arguments without performing a duplicate `calldataload`. Each instruction parses zero or more bytes off the calldata and performs some side effect. Most instructions increment the PC and jump to the next instruction; stop instructions do not, instead they exit the interpreter and perform any finalization necessary to preserve invariants. MultiCall uses a number of variables in memory and storage to keep track of its state. There is only one global stack variable shared between instructions: the program counter on the top of the stack. As stack access is slightly cheaper, other globals could be transferred to the stack in future versions.

5.6.2 Initialization

MultiCall's initialization is conceptually simple: it sets global variables, pushes the PC with value 0 onto the stack, and dispatches. To avoid confusion, note that the interpreter init code run on each invocation is distinct from the contract initialization code run when MultiCall is created.

5.6.3 Jump table

The following code generates the jump table into which `dispatch_mc jumps`, given an instruction set `is`:

```
jumptable_mc is = do
  let len = length is
      padlen = 52 - len
  if len > 52 then error "Too many instructions in JT" else void
  ls <- sequence [do l <- jumpdest
                      instr
                      return l
                  | instr <- is]
  place "JUMPTABLE_MC"
  sequence_ [do jumpdest
                jump_ $ labelExpr16 l
             | l <- ls]
  sequence_ [do jumpdest
                invalid
                stop
                stop
                stop
             | n <- [1..padlen]]
```

An advantage of using an EDSL is that one gains access to the host language's preexisting library of combinators. The `sequence` and `sequence_` functions are examples of this; they are monad combinators which take a list of monadic actions and run them in sequence, returning the list of results or an empty tuple respectively.

The jump table consists of sections, each a jumpdest instruction followed by a jump to the label where the code implementing an instruction is placed:

```
--This is a subexpression of jumptable_mc is
do jumpdest
   jump_ $ labelExpr16 l
```

The expression `(jumpdest :: C Label)` outputs the `JUMPDEST` opcode, allocates a label pointing to that byte, and returns the label. Valid opcodes jump to the offset of a jumpdest, and from there to the offset where the instruction code is placed. As each section is 5 bytes in size and an opcode byte may address offsets 0-255, there can be at most 52 valid instructions. That is why `jumptable_mc` checks whether the length of the list of instruction bodies `is` passed to it is greater than `52` and errors if so. MultiCall provides only 30 instructions, so it needs only 30 of the 52 available slots; the spare space is filled with sections which simply throw an exception:

```
--This is a subexpression of jumptable_mc is
do jumpdest
   invalid
   stop
   stop
   stop
```

The `invalid` instruction is an invalid opcode, which throws an exception. The `stop` instructions are to pad the section to 5 bytes total. The astute reader may note that jumping to a byte offset selected by a potential attacker seems like an unsafe practice: after all, they may jump to the an offset between jumpdests. However, the EVM only allows jumps to the `JUMPDEST` instruction; all others throw an exception, reverting contract execution. That limits the user to the instructions provided by the interpreter; if they preserve the interpreter's invariants, then the interpreter itself is secure.

5.6.4 Location of instructions

MultiCall's instructions are interpreted from calldata during execution. The calldata space is immutable: there are EVM instructions to read from it, but not to write to it. To implement code which may mutate itself or load code from storage (or the text of other contracts), storing bytecode in EVM's volatile memory space would be necessary. Doing so would allow program code to be stored and loaded more cheaply, rather than being resubmitted as transaction data each time. That would be a useful feature for a Turing-complete interpreter. MultiCall is intentionally not Turing-complete, and therefore calldata-stored instructions are sufficient. The reason for restricting its functionality is given in Section 5.7.1.

5.6.5 Putting it all together

The result of putting all the components together is the following:

```
mc_alt = do
  init_alt
  jumptable_mc iset_alt
```

which defines the final code of the interpreter. The creation code also sets storage variables such as `owner`. One could refactor the above code to parameterize it by the init code, dispatch code and instruction list, thus creating an interpreter-in-a-box combinator. As I have not needed to write a large number of different interpreters, I have not yet done so.

5.7 High-level design of the interpreter

5.7.1 High-level objectives

With the low-level interpreter implementation details and format of instructions decided, the question becomes what instructions to provide. These instructions are selected and designed according to the design objectives of the interpreter. To assess whether they achieve their purpose, those objectives must be made explicit. The purpose of the MultiCall smart contract is to reduce the gas cost of accesses to the Ethereum blockchain, thus creating commercial value. It does so by emulating multiple transactions using a single call transaction to the interpreter, eliminating the fixed transaction cost of the emulated transactions. The total fixed cost of transactions on the Ethereum blockchain is significant, on the order of 60,000 US dollars per day [11, 10]. Furthermore, MultiCall is intended to capture some of this value in the form of fees paid to the contract's owner. In the course of ordinary use of the interpreter, users may allocate data within MultiCall's persistent storage. As the Ethereum protocol will soon be modified to charge rent for storage on the blockchain, there must be a fee for data allocation. The fee scheme for gas saved and data use is described in Section 5.7.7. The compiled EVM bytecode of smart contracts is inherently public as it is uploaded to the blockchain. The source code is usually also made public to provide trust that the author has not added backdoors which steal user funds. As such, contracts are eminently piratable. If the contract generates significant revenue, there will be a strong incentive to pirate it. To mitigate the risk of piracy, the contract should benefit from network effects such that its value to each user grows with the number of users. How this is attempted is described in Section 5.7.8. To reason about the security of a contract, its intended properties should be clearly stated. Some intended invariants of MultiCall are described briefly and informally in Section 5.7.6. A full and formal specification of MultiCall's desired properties is beyond the scope of this thesis.
The instructions provided by the interpreter can be grouped into features, high-level design elements intended to implement a unit of functionality. My design includes the following features: *Call*, *Deposit*, *Proxy*, *Create* and *Signed*. These features are described in more detail below.

As the objective of the interpreter is to emulate a list of transactions, it provides a number of call transaction-emulating instructions and contract creation instructions. Each caller address is given a separate account, which they may deposit ether to or spend from to make calls, deposit to other accounts, or create contracts. As the interpreter stores money on behalf of users, program correctness is paramount. It should be impossible to unintentionally lose money, forge deposits or steal from other users. When a widely used Ethereum contract is incorrect, tens of millions of dollars worth of ether may be lost or stolen. It is the opinion of the author that a commercial version of the interpreter should therefore be proven correct. To ease formal verification, I have intentionally restricted the functionality of MultiCall to be Turing incomplete; the interpreter executes a sequence of instructions without jumping or looping in them. While a Turing-complete instruction set with control flow and arithmetic operations might reduce transaction costs by loading and reusing code already submitted to the blockchain and by decompressing instructions to reduce their data cost, such a contract would be more difficult to analyze and verify. There is also a strict lower bound on the overhead of emulating a transaction: the cost of the CALL and CREATE EVM instructions. This bounds the savings on the fixed transaction cost of emulating a paying call transaction in EVM code to (21000 - 7400 / 21000 = 64.76%). For emulating existing transactions, I believe a non-Turing-complete interpreter is sufficient. MultiCall achieves savings close to the theoretical maximum; this is described in detail in Chapter 4.

5.7.2 Monetary unit

All balances are denominated in the monetary unit of MultiCall, 1 *gwei*, equal to 10^9 of the smallest and indivisible ether unit, the *wei*. While the `ownerBalance` is a full 256 bit word, user balances are 48 bits wide. All instruction arguments specifying ether amounts are 48 bits and denominated in the monetary unit. This is done to reduce the size of accounts and argument data. Ether currently trades at around 250 USD, and therefore 1 gwei is worth approximately 0.25 micro-USD. The maximum amount of ether representable with a 48-bit gwei-denominated balance is approximately 280,000 ether, worth over 60 million USD. I believe that accuracy and maximum balance to be sufficient in practice, even considering the fact that cryptocurrencies such as ether have been known to appreciate and depreciate by an order of magnitude.

5.7.3 Persistent state overview

MultiCall uses 4 global variables and 2 arrays in persistent storage: `owner`, `fallback`, `ownerBalance` and `monVars`, and the arrays `accounts` and `rolodex`.

- `owner` contains the address of the owner, and is used to authenticate admin instructions such as `ownerWithdraw` (which withdraws funds from `ownerBalance`), `ownerSetOwner`, (which sets the owner variable to a new address), and `setMon`, which sets monVars.
- `fallback` is a a secondary address, which can be used to set `owner` and `fallback`. Since the owner's private key is required to withdraw fees, it must

be used frequently. That exposes it to the risk of theft; the fallback key provides additional security, and can be kept in cold storage cut off from the network until a compromise of the owner key occurs.

- `ownerBalance` is a balance containing the amount of ether paid in fees. When funds are withdrawn from it via `ownerWithdraw`, it is decremented. When a MultiCall invocation pays fees, it is incremented.
- `monVars` contains a struct of three variables used by the fee scheme. This is described in more detail in section 5.7.7.
- The `accounts` array is indexed by addresses; each element contains an `Account` struct. The `Account` struct is used to record the information relevant to each user address.

  ```
  //Pseudocode:
  struct Account {uint160 proxy, uint16 nonce, uint48 balance}
  ```

  The `proxy` field is used to record the address of the user's proxy, a contract which forwards calls on behalf of the user. The `balance` field records the amount of ether credited to the user in the monetary unit. The `nonce` field is used by the Signed feature.
- The `rolodex` array is a map from 24-bit indices to `Rolodex` structs.

  ```
  //Pseudocode:
  struct RolodexEntry {uint24 owner, uint160 address}
  ```

  The `address` field contains a 160-bit address. Three-byte rolodex indices can be used by variants of the MultiCall call and deposit instructions in place of a 20-byte address. Instead of parsing a 20-byte address directly off the calldata, rolodex-using instructions parse a 3-byte index off and use it to `SLOAD` from the rolodex array and mask out the address field, at a cost of approximately 200 gas. That saves around 1000 gas, saving a further 5% of the fixed transaction cost for calls and payments to frequently-used addresses. The `owner` field is a 24-bit rolodex index indicating which address, if any, may overwrite the rolodex entry. If it is 0, the entry is immutable. While writing an immutable entry permanently uses up one space in the array, it provides certainty to users that the entry will not be maliciously rewritten to redirect calls and deposits.

5.7.4 Volatile state overview

MultiCall uses 8 one-word in-memory variables, placed at address 0. Instructions such as `call` which use additional memory write to the memory beyond.

- `signatory` is the address on whose behalf MultiCall is currently executing. It is set to the caller address by the init code. The `signed` instruction temporarily overwrites it during the execution of signed scripts, to the signatory of the given script. Signed scripts consist of a sequence of instructions and some additional arguments passed to `signed`. Using the Signed feature, a single call to MultiCall can execute on behalf of multiple users.
- `balance` tracks the credit or debit of the signatory. When an instruction such as call or create sends ether to another contract, `balance` is debited. It is also debited when the user deposits ether to another account using the deposit

instruction. At the end of the script, MultiCall settles the balance against the signatory's account in persistent storage. If `balance` is positive, the account is credited. If `balance` is negative, MultiCall attempts to debit the account; if the account's balance is insufficient, it throws an exception and execution reverts. `balance` is set to the amount of gwei sent to MultiCall in the init code, allowing users to deposit ether to their account by sending ether to the contract.

- `cachedBalance` is used to cache the original signatory's balance during the execution of a signed script.
- `gasSaved` tracks the total gas saved by using the interpreter instead of sending transactions directly. For instructions which can be used to emulate transactions, 21000 less the overhead is added to `gasSaved`. For `createProxy` the proxy creation cost is deducted as it is an expense. The amounts added and deducted do not currently exactly correspond to the real gas savings and costs; that is required for a commercial version of MultiCall, but not for the proof of concept. In the init code of the interpreter 5200 gas is deducted to offset the cost of depositing fees to `ownerBalance`; an additional 21000 gas to cover the transaction cost is deducted if the call is from an externally owned account rather than another contract.
- `deltaWords` is used by the fee logic to track the number of persistent storage words allocated by the user. A storage word is allocated when it is set to a non-zero value from zero, and deallocated in the reverse case. MultiCall increments the `deltaWords` variable whenever an account or rolodex entry is allocated, and decrements it when they are deallocated.
- `gasPrice` is used by the fee logic to charge the correct amount of ether per gas saved. It's calculated from `monVars` in the initialization code.
- `wordCost` is charged in fees at the end of the script for each word allocated. It's retrieved from `monVars` in the initialization code.
- `nonce` is used by the Signed feature to protect from replay attacks. When MultiCall is not executing a signed script, it is `0`.

5.7.5 Instruction overview

The instructions provided by MultiCall are defined in `iset_alt`:

```
iset_alt = (let ?logging = False in iset_alt') ++
           (let ?logging = True in iset_alt') ++
           [stop_alt] ++
           --Admin
           [ownerWithdraw,ownerSetOwner,ownerSetMon,
            fallbackSetOwner,fallbackSetFallback]
--Those instructions which depend on whether ?logging = True
iset_alt' :: (?logging :: Bool) => [C()]
iset_alt' = (let ?addr = Address in iset_alt'') ++
            (let ?addr = Rolodex in iset_alt'') ++
            [create_alt,createProxy,proxy_dot_create,signed,vacate,writeRolo]
--Those instructions which depend on ?addr
```

```
iset_alt'' :: (?logging :: Bool, ?addr :: ArgType) => [C()]
iset_alt'' = [call_alt,proxy_dot_call,deposit]
```

The variable names prepended with '?' may be unfamiliar even to Haskell programmers; they are implicit parameters, enabled with the `ImplicitParams` language extension. Implicit parameters allow values to be passed to expressions without explicitly including them as function arguments. The implicit parameters used are the Boolean value `?logging` and argument type value `?addr`, which determine whether to include logging code and how to parse address arguments in the parameterized instructions respectively.
The instructions can be divided into features: sets of instructions which together provide certain functionality. The features are *Call*, *Proxy*, *Rolodex*, *Deposit*, *Admin*, *Create* and *Signed*. They appear to work correctly under manual testing; they are described below and evaluated in Chapter 4. The proof of correctness required for commercial use is beyond the scope of this thesis. Admin is intended to be used only rarely by the contract's owner; it's described briefly.

5.7.5.1 Call

The Call feature consists of 8 instructions which make calls, intended to emulate call transactions. They are divided into two call types with 4 variants each: `call` and `proxy_dot_call`. The instruction `call` performs an EVM CALL directly from the MultiCall contract, and can be used for payment and for contract calls which do not require caller authentication. The instruction `proxy_dot_call` commands the user's proxy to make the call instead and is also part of feature Proxy; it is described in more detail below. The 4 variants for each call type are the product of two binary choices: whether the call is to be logged or not, and whether the callee address is to be passed directly or via a rolodex index. Logging a call makes it easier for clients to verify it occurred, but costs more gas. Non-logging calls are suitable to transfers to recipients who run full nodes, are prepared to wait for confirmation, or are willing to trust a third party running a full node to verify that a payment occurred.
Calls to a rolodex index are suitable when there exists a rolodex entry with the desired callee address and the user trusts that the rolodex entry will not be mutated. If the user owns the entry or if it is immutable they may know with certainty it will not be maliciously mutated. The entry could also be controlled by a smart contract such as a rolodex entry market-maker whose code could be verified not to mutate the entry before a certain time.

5.7.5.2 Proxy

Ethereum smart contracts typically use the address of the caller for authentication; for example, a method call to transfer funds from user A to user B will require that the calling address is A. The MultiCall contract has only a single address, and thus only one identity from the perspective of other contracts. For practical use of the interpreter, MultiCall must be able to provide a separate identity for each user. This is achieved with the Proxy feature, which consists of the instructions `createProxy`, `proxy_dot_call` and `proxy_dot_create`. The instruction `createProxy(amount)`

creates a new proxy contract with a given amount of gwei as its initial balance and writes its address to `accounts[signatory].proxy`. Proxies forward calls from MultiCall if the calldata contains a special value as its last 4 bytes. That value is disallowed in the last 4 bytes of ordinary calls from MultiCall. The instruction `proxy_dot_call` calls the signatory's proxy (`accounts[signatory].proxy`) with data specifying a call to make, which the proxy then executes. The Proxy feature thus grants each user a unique identity controllable only by them, available from the interpreter.
The instruction `proxy_dot_create` works similarly, using a different 4-byte special value. It's useful to create contracts from the proxy rather than MultiCall because it's a common pattern that initialization code sets the created contract's controller to the address that created it.

5.7.5.3 Rolodex

The Rolodex feature consists of the instruction `writeRolo` and the instructions of Call and Deposit which use rolodex indices. The instruction `writeRolo(roloIx,roloStruct)` writes the rolodex struct `roloStruct` to the rolodex index `roloIx` if the signatory is entitled to do so. The signatory is only allowed to write to entries which are empty (`rolodex[roloIx] == 0`) or owned by them (`rolodex[rolodex[roloIx].owner].address == signatory`). Using rolodex shorthands provides significant gas savings. Rolodex entries can be traded between users by setting their owner.

5.7.5.4 Deposit

The Deposit feature consists of 4 variants of the `deposit(recipient,amount)` instruction. Like calls, they are either logging or non-logging and use either 20-byte addresses or 3-byte rolodex indices to identify the recipient. The functionality of deposit is simple: it increments the recipient's account balance by the given amount and deducts the same amount from the `balance` memory variable. It can be used to implement transfers more cheaply than paying calls, as reading and writing a storage index costs 5200 gas as opposed to the 7400 gas cost of a paying call.

5.7.5.5 Create

Create consists of two instructions: `create` and `proxy_dot_create`. They create a contract directly from MultiCall or from the signatory's proxy respectively. Both instructions accept two arguments, a 6-byte field specifying the number of gwei to transfer to the contract upon creation, and a bytestring consisting of a 16-bit length field followed by a variable number of bytes. The bytestring is the initialization code of the contract to be created.

5.7.5.6 Signed

Signed consists of a single instruction:
`signed(signature,scriptLen,deadline,tip,nonce)`, which is intended to be followed by a sequence of instructions. Those instructions, together with the `deadline`,

`tip` and `nonce`, may be termed a *signed script*. The length of that script in bytes is given by `scriptLen`. The parameter `signature` is a cryptographic signature of that script. Given the hash of the script and the signature, the signatory's 160-bit Ethereum public key can be recovered using `ECRECOVER`, a primitive contract built into the EVM. The instruction `signed` sets the signatory variable to that value and then executes the script in the context of that signatory. The deadline is checked prior to signature verification; if the current block's timestamp (available from the `TIMESTAMP` EVM instruction) is greater than the deadline, then the signed script is invalid and will not be run. That allows users to limit the timeframe in which a signed script can be run, as the timestamp corresponds approximately to the real time. The tip field is used to pay the user who includes the signed script in their transaction for doing so; it is analogous to Ethereum transaction fees. The nonce is used to protect against replay attacks; signed scripts fail when the nonce field of the signatory's account does not match the given nonce, and it is incremented on each signed script execution. The `vacate` instruction zeroes the signatory's account, thus also zeroing the nonce field. That is why the deadline field is necessary: to prevent replay attacks after account vacation and recreation. In combination with the Proxy feature, this feature allows calls from multiple addresses to be made in a single invocation of the interpreter.

5.7.5.7 Admin

The admin instructions are only usable by the owner and fallback keys; for other users, they have no effect. They allow the owner to configure monetization variables, withdraw fees paid, and transfer ownership. The fallback key may recover ownership of the contract on compromise of the owner key.

5.7.5.8 Pseudocode

Here follows pseudocode of a few instructions, to give a better idea of how they are defined in practice. The PC counter increment and dispatch is implicit. The pseudocode keyword `continue` goes to the next instruction, popping temporary variables off the stack. `Proxy(v)` is the constructor for proxies, which takes an initial value of ether in gwei to give the proxy. The names `CALL` and `STOP` refer to the EVM instructions so named. Assignments to non-global vars denote local variable allocation. `if ?logging then a else b` is statically resolved, analogously to a C `#ifdef`. The expression `specialString` is a 4-byte bitpattern placed last in calldata, denoting a proxy call if the lowest bit is 0 and a proxy create otherwise.

`Call:`

```
call(g:Gas, a: ?addr, v:Amount, len:Len, data:Bytes len){
    if (data[len-4..len-1] & 0xfffffffe) == specialString: continue
    success = CALL(g,a,v*10^9,calldata = data[0..len-1])
    if(success){
        //fdPC_const and fdNPC_const are constants referring to the
        //gas saved for paying and non-paying calls respectively.
        if(v) gasSaved += fdPC_const; else gasSaved += fdNPC_const
```

```
        balance -= v
        if ?logging then log(topic_call,signatory,a,v) else void
    }
}

proxy.call(g:Gas, a: ?addr, v:Amount, len:Len, data:Bytes len){
    prox = accounts[signatory].proxy
    succ = CALL(g,prox,calldata = {a,v,len,data[0..len-1] ++
                                   specialString})
    if(succ){
     gasSaved += fdProxyPC_const + (!v) * 6768
     if ?logging then
     log(topic_proxy_dot_call,signatory,a,v)
     else void
    }
}

Rolodex:

writeRolo(ix:Bytes 3, roloStruct:Bytes 23){
    rolo = rolodex[ix]
    //It currently also logs on failed writes; this should be changed.
    if ?logging then log(topic_writeRolo,signatory,addr) else void
    if(!rolo){
     rolodex[ix] = roloStruct
     deltaWords += (roloStruct > 0)
     continue
    }
    ownerIx = rolo.owner
    if(ownerIx == 0) continue
    ownerAddr = rolodex[ownerIx].address
    if(signatory != ownerAddr) continue
    rolodex[ix] = roloStruct
    if(roloStruct == 0) deltaWords -= 1
}

Proxy:

createProxy(v:Amount){
    acc = accounts[signatory]
    if(acc.proxy) continue
    prox = Proxy(v)
    if(!acc)
        accounts[signatory] = {proxy=prox,nonce=1,balance=0}
    else {
        acc.proxy = prox
```

```
        accounts[signatory] = acc
    }
    gasSaved -= fdCreateProxy
    balance -= v
    if ?logging then log(topic_createProxy,signatory,v) else void
}

stop(){
    if(nonce) goto stop_signed //Omitted for brevity
    fees = max(0,deltaWords)*wordCost +
               max(0,gasSaved)*gasPrice/(0xffff*10^9)
    if(balance == fees){
     if(!balance) STOP()
     ownerBalance += balance
     STOP()
    }
    acc = accounts[signatory]
    if(!acc){
        balance -= fees
        if(balance < 0) revert("Insufficient balance")
        if(wordCost >= balance){
         ownerBalance += balance + fees
         STOP()
         }
         ownerBalance += fees + wordCost
         //Quirk: don't deposit >= 2^18 ETH into empty account
         accounts[signatory] = {nonce=1,balance=balance-wordCost}
         STOP()
    }
    accbal = acc.balance
    accbal' = accbal + balance - fees
    if(!(accbal' >= 0 & accbal' < 2^48))
        revert("Balance over/underflow")
    if(fees) ownerBalance += fees
    if(accbal != accbal')  account[signatory] = acc + balance - fees
    STOP()
}

//Init is not an instruction, it is the interpreter initialization code
//which runs at the start of the call to MultiCall.
init(){
    mv = monVars
    wordCost = mv.wordCost
    gasPrice = max(mv.minGasPrice*(10^9 / 256))*cutOfGas
    signatory = CALLER()
    balance = CALLVALUE() / 10^9
```

```
    if(CALLER() == ORIGIN()) gasSaved -= 26200; else gasSaved -= 5200
    PC = 0
    dispatch_mc
}
```

The following is an example of the real DSL code, for the MultiCall interpreter entry code corresponding to `init` in the pseudocode:

```
init_alt = do
  load monVars
  [wordCost',minGasPrice,cutOfGas] <- unpackRight [Amount,Len,Len]
  store wordCost void
  --pops wordCost', now minGasPrice is top of stack
  void * fromInteger (giga `div` 256) --MGP now in wei
  ifnot (dupVar minGasPrice >? gasprice)
    (pop >> gasprice) --MGP max= gasprice
  void * dupVar cutOfGas
  --mul with COG, will *= gasSaved, /= gwei later
  --Ah, need to divide by 2^16-1 later as well. Why not 2^16?
  --Want to charge precisely 20% of gas saved for debugging...
  store gasPrice void
  --now cutOfGas is top of stack, but it can be safely ignored
  --since dispatch never returns
  store signatory caller --Needed for signed feature
  --balance = amount deposited in gwei
  store balance $ callvalue / giga
  ifte (caller ==? origin)
    (gasSaved -= 26200)
    (gasSaved -= 5200)
  0 --PC = 0
  dispatch_mc
```

5.7.6 Invariants

This is a non-exhaustive list of properties which the contract is intended to fulfill; more may be discovered in the course of future formal verification.

- Barring changes to the protocol which automatically deduct rent from the contract and ignoring deposits of fractions of the monetary unit, the sum of user balances and the ownerBalance should be equal to the ether balance of the contract in gwei.
- No two accounts should have the same proxy; proxies should only forward calls on behalf of their creator.
- It should only be possible for the owner to avoid paying fees.

In future versions of the code generator it may be desirable to build code generation combinators which enforce invariants by construction. However, the `C` monad operates at a level of abstraction corresponding approximately to assembly and is not well-suited for that purpose. Preserving information about the abstract syntax

tree of expressions and subjecting them to type-checking and static analysis would be of interest.

5.7.7 Fees

Fees consist of two components: gas fees and storage fees. Gas fees are a cut of the ether value of the gas saved. Storage fees are charged for net word allocation in a call. They can be configured by setting `monVars`; `monVars` is a struct with three fields: `wordCost`, `minGasPrice` and `cutOfGas`. The `wordCost` field is a 48-bit number of monetary units, and is the amount charged per word allocated. The field `cutOfGas` is a 16-bit number, the cut of ether savings as a fraction of $2^{16} - 1$. It may take values between 0 and 100%, with a precision of 0.0015%. The purpose of `minGasPrice` is to mitigate collusion between transaction senders and miners to accept the sender's transaction with a gas price of 0 in return for payment off-chain. If such collusion becomes common, the `minGasPrice` can be set to a reasonable value. The field `minGasPrice` is a 16-bit number denominated in gwei/256, and may take values between 0 and approximately 256 gwei, with a precision of 0.004 gwei. The gas price is currently around 10 gwei. Fees total the following formula:

```
max(0,deltaWords)*monVars.wordCost +
    max(0,gasSaved) *
    (max(gasprice,monVars.minGasPrice*(10^9/256))/10^9) *
    (monVars.cutOfGas/0xffff)
```

which is implemented in the program as

```
fees = max(0,deltaWords)*wordCost +
    max(0,gasSaved)*gasPrice/(0xffff*10^9)
```

where `wordCost` and `gasSaved` are memory variables computed from `monVars` in the initialization code. The variable `gasPrice` is set to

```
max(monVars.minGasPrice*(10^9 / 256))*cutOfGas
```

and `wordCost` is set to

```
monVars.wordCost
```

in the initialization code. The first summand of `fees` is the fee charged for net allocation. Note there is no negative fee on a net deallocation. However, users still have an incentive to deallocate words since they can deduct deallocations against allocations. The second summand is the fee charged for gas saved. The term is divided by 10^9 because the `gasPrice` is in wei and needs to be converted to the monetary unit gwei, which is equal to a billion wei. The term is further divided by `0xffff` because `gasPrice` was multiplied in the init code by `cutOfGas`, which represents a fraction of `0xffff`. The division by `0xffff` is delayed until after the variable is multiplied by `gasSaved` in order to reduce rounding errors.
The fee for storage use is implemented as a purchase rather than rent to reduce code complexity. It also provides users certainty that the MultiCall owner is not able to seize deposited balances by hiking their rent to unreasonable levels, or evict immutable rolodex entries. Allocation fees create some edge cases in the behaviour

of instructions: if a MultiCall invocation ends with the in-memory balance greater than 0 but less than `wordCost` and the in-storage account is unallocated, the entire balance is charged in fees. If an invocation ends with the balance greater than `wordCost` but the account is not allocated, `wordCost` is deducted from the balance before it is deposited to the account. Deposits to empty accounts deposit `wordCost` less, to compensate for the higher allocation fee paid by the depositor. Deposits of less than `wordCost` to empty accounts have no effect. That mitigates an attack where the recipient of a deposit could cause the depositor to pay them `wordCost` extra by vacating their account after the depositor publishes the depositing transaction but before it is accepted by the blockchain. An attack where funds are diverted unexpectedly by the attacker running a transaction before the victim's transaction is accepted but after it is irreversibly published may be termed a preemption attack. There is another such attack I have not mitigated, which can be performed by the MultiCall owner against users. If calls to the interpreter are preempted by the owner increasing fees, users may pay more than they expected. In particular, `wordCost` may currently take on a value of up to approximately 60 million dollars, and calls which end with a positive balance less than `wordCost` are paid to `ownerBalance` if the signatory's account is unallocated. The owner might be able to seize deposits to new accounts and cause transactions which allocate rolodex indices to pay all of the signatory's balance in fees. One way of mitigating the attack is to apply writes to `monVars` 24 hours after the write is registered to the contract, giving users time to respond and rendering preemption of transactions impractical. That would be of interest in a commercial version of MultiCall.

5.7.8 Network effects

Though it is perhaps a vain hope, the Deposit and Rolodex features are intended to render piracy less attractive by providing MultiCall with a positive network effect. The idea is that the more people use the "legitimate" interpreter contract deployed and controlled by its legal owner, the more accounts and rolodex entries it will accumulate. The deposit instruction is cheaper than a paying call; if there are many users with open accounts in the contract that reduces the cost of using the legitimate interpreter to pay them. If many popular addresses are written to the rolodex by users, that increases the probability that any given transfer can be achieved cheaply in the interpreter; a pirated interpreter would have to resubmit the same list of addresses to compete. If the setup cost of a competitive pirated interpreter outweighs the fees paid for using the legitimate one, piracy might be deterred.

6
Evaluation

Below I evaluate the marginal gas savings for the different variants of `call`, `proxy_dot_call` and `deposit`. First I must compile MultiCall via the Haskell shell and make it available to truffle:

```
> hackyDeploy (truffleDir ++ "MultiCall") mc_alt_deploy
```

That creates a file named MultiCall containing EVM bytecode in truffleDir, which I have defined as the path to the folder in which the truffle console runs. Then, I deploy MultiCall to the private chain:

```
> MultiCall = util.deploy("MultiCall")
```

That sets the MultiCall variable to the address of the deployed MultiCall contract. Creation costs 2113140 gas; the interpreter is a large contract and therefore creation is expensive. Gas usage can be seen in the ganache-cli shell. I also deploy a dummy contract testStop to the file TestStop; its only functionality is to immediately return to the caller when it is called.

```
> TSI = util.deploy("TestStop")
```

Then I prepare the contract's state by opening an account and depositing 1 ether (10^{18} wei) to it.

```
> web3.eth.sendTransaction({from: acc0,
                            to: MultiCall,
                            data: "0x" + api.iset.stop(),
                            value: 10**18})
```

`acc0` is a variable I set to one of the addresses available on the private chain, `web3.eth.accounts[0]`. Its private key is stored on the private Ethereum node and available for signing transactions. The module `api.iset` allows one to generate MultiCall instruction hexstrings from the instruction's mnemonic name and appropriate arguments. Scripts are sent to the interpreter as the hexadecimal number prefix `"0x"` prepended to the concatenation of a number of instructions; the addition operator concatenates strings in Javascript. The `api` module is used to interface with the MultiCall interpreter and inspect its state.

```
> api.account(MultiCall,acc0)
```

now returns:

```
"0x0100003b94af80"
```

That is an account struct, with the proxy field set to zero, the nonce set to 1,

and balance equal to `0x3b94af80`. That is the number of gwei deposited, less the wordCost:

```
> api.vars(MultiCall).monVars.wordCost
```

returns `"0x061a80"`, which is the value to which it is set on MultiCall constract creation. The number is denominated in gwei and equals 1/2500 ether. The wordCost has been charged in fees and deposited to the `ownerBalance`:

```
>api.vars(MultiCall).ownerBalance
```

which is now equal to `0x061a80`. The sum of `acc0`'s account balance and the `ownerBalance` is 10^9, the number of gwei monetary units deposited. I now set a one-byte rolodex index to `TSI` and create a proxy for `acc0`:

```
>web3.eth.sendTransaction({...,
    data: "0x" + api.iset.writeRolo(32,0,TSI) + api.iset.stop()}
>web3.eth.sendTransaction({...,
    data: "0x" + api.iset.createProxy(10000) + api.iset.stop(),
    gas: 100000})
```

The gas limit must be specified explicitly in the second transaction as proxy creation costs approximately 70k gas, and the default gas limit for transactions sent by web3 is only 90k. The gas cost of writing a new rolodex entry is approximately 20k, corresponding to the cost of a `SSTORE` instruction to an unallocated storage index. The cost of creating proxies and rolodex entries is amortized over multiple proxy calls and rolodex-using calls and deposit; it is primarily the cost of calls and deposits which determine MultiCall's efficiency. I therefore do not profile `createProxy` or `writeRolo` in more detail. `acc0`'s account now contains the proxy address; the proxy's balance is 10,000 gwei, as specified in the `createProxy` instruction:

```
> it = api.account(MultiCall,acc0)
> web3.eth.getBalance(it.substring(0,42))
```

is the balance of the account's proxy and returns:

```
BigNumber {s: 1, e: 13, c: [10000000000000]}
```

The above value is a Javascript bigint representing 10,000 gwei, as expected. In addition, I deposit some ether to `TSI`'s account, to ensure that it is not empty (which might cause deposits to fail due to the `wordCost`) and that the 15000 allocation gas cost does not confound profiling:

```
> web3.eth.sendTransaction({...,
    data: "0x" + api.iset.deposit_rolo(32,1000000)})
```

Now I can test the cost of `call`, `proxy_dot_call` and `deposit` variants. While I offer no formal proof that the execution cost of a MultiCall instruction is independent of its location, I hope the results will convince the reader. The costs of logging versus not logging, paying versus not paying and using an address versus a rolodex index are additive, as the logic of each does not interact. I can therefore avoid profiling some variants. The instructions used are the following:

- `call` variants

```
> callRoloPaying = api.iset.call_rolo(0,32,1,"0x")
```

The above instruction calls the address of the rolodex entry at index 32 (set to `TSI`) with 0 gas and deposits 1 wei with no calldata.

```
> callRoloNonPaying = api.iset.call_rolo(20*1024,32,0,"0x")
```

Since calling a contract with no ether transfer and a zero gas limit has no effect, I set the gas limit to a one-byte value. The data cost is the same as for `callRoloPaying`.

```
> callAddrPaying = api.iset.call_address(0,TSI,1,"0x")
```

Calls `TSI` using a direct address; has the same effect as `callRoloPaying`.

```
> callRoloPayingLogging = api.iset.call_logging_rolo(0,32,1,"0x")
```

Identical to `callRoloPaying`, except it logs the call.

- `proxy_dot_call` variants

```
> pCallRoloPaying =
    api.iset.proxy_dot_call_rolo(20*1024,32,1,"0x")
> pCallRoloNonPaying =
    api.iset.proxy_dot_call_rolo(20*1024,32,0,"0x")
> pCallAddrPaying =
    api.iset.proxy_dot_call_address(20*1024,TS,1,"0x")
> pCallRoloPayingLogging =
    api.iset.proxy_dot_call_logging_rolo(20*1024,32,1,"0x")
```

Each `proxy_dot_call` variant is analogous to their call equivalents.

- `deposit` variants

```
depositRolo = api.iset.deposit_rolo(32,1)
depositAddr = api.iset.deposit_address(TS,1)
depositLogging = api.iset.deposit_logging_rolo(32,1)
```

The cost of 0-99 such instructions is calculated with:

```
> arr = Array(100);
  for(var i = 0; i < 100; i++)
     arr[i] = util.gasUsed({
         from: acc0,
         to: MultiCall,
         data: "0x" + util.times(i,instr) + api.iset.stop(),
         gas: 8000000})
```

For `instr` equal to each instruction. The utility function `util.gasUsed` returns the gas used from sending a transaction with the given parameters. The expression `util.times(n,string)` repeats the given `string` `n` times. The differences between `arr` indices, indicating marginal instruction cost, is calculated as:

```
> diffs = Array(99);
    for(var i = 0; i < 99; i++) diffs[i] = arr[i+1] - arr[i]
```

Multiple identical instructions in were called at a time in order to find patterns in

their variation in cost. In fact, they vary only under limited circumstances. This is explained in more detail below.

6.1 Instruction parameter selection

The gas limit, number of monetary units transferred, and rolodex index values used in the profiled instructions contain at most one non-zero byte. The number of non-zero bytes contained in an instruction is important to its performance, as each such byte costs 64 extra gas. While gas limit and rolodex index arguments are 24 bits and ether amounts are 48 bits wide, I believe that values with one non-zero byte are sufficient for most use cases. There are 765 rolodex indices with one non-zero byte. Examination of Ethereum transactions with [10] suggest a small number of popular recipients receive a significant proportion of all Ethereum calls, though I have been unable to find hard statistics to that effect. One-byte gas amounts can represent values between 256 and 65280 gas with 256-gas precision, and values between 65536 and 16711680 gas with 65536-gas precision. I believe that is sufficient in practice. I believe 48-bit gwei amounts with one non-zero byte are also sufficiently accurate; for example, they can represent values between 1/64 and 4 ether with an accuracy of 1/64 ether, or approximately 4 USD at current prices. Like floating-point numbers, smaller values can be represented with greater accuracy, and larger values can be represented by using a greater exponent (i.e. a higher non-zero byte).

6.2 Results

For `callRoloPaying`, the `diffs` array contains 8171 except at `diffs[0]` and `diffs[2]`, where it's 13514 and 13383 respectively. That is because when the first paying call is made the account balance needs to be updated. When the third paying call is made, the gas saved estimated by the fee scheme exceeds the fixed 26200 deduction and fees are charged; then the `ownerBalance` is updated as well. The fixed cost of a call transaction to MultiCall is 32222 gas when both the account and `ownerBalance` are updated. If the caller sent precisely the amount spent by the script, the account write could be avoided, saving approximately 5200 gas. I do not do so, as the intention is only to find the marginal cost per instruction.
The other instructions exhibit a similar pattern, with the maximum fixed cost equal to 32222. After a small number of instructions (4 or less for all), the maximum fixed cost is charged and the cost of the instruction becomes solely the execution cost. For comparison, blocks typically contain in the order of 100 transactions. MultiCall is intended to be able to replace a full block of transactions with a single one, and as such only marginal performance for a large number of instructions is of interest. The marginal costs and savings vs transaction fees per instruction are:

- `callRoloPaying`: 8171 gas, savings: 61%
- `callRoloNonPaying`: 1481 gas, savings: 92.9%
- `callAddrPaying`: 9154 gas, savings: 56.4%
- `callRoloPayingLogging`: 9754 gas, savings: 53.6%
- `pCallRoloPaying`: 9347 gas, savings: 55.5%

- `pCallRoloNonPaying`: 2583 gas, savings: 87.7%
- `pCallAddrPaying`: 10394 gas, savings: 50.5%
- `pCallRoloPayingLogging`: 10930 gas, savings: 48.0%
- `depositRolo`: 5878 gas, savings: 72.0%
- `depositAddr`: 6946 gas, savings: 67.0%
- `depositLogging`: 7461 gas, savings: 64.5%

At the current gas price, ether price and daily transaction rate [11, 10], approximately 60 thousand USD is spent per day on the fixed transaction fee. If MultiCall could save 61% of that cost (the savings of a paying call) and transaction rates remained stable, approximately one million USD could be saved per month.

6.2.1 Further profiling

I also evaluated the Create and Signed features (consisting of the instructions `create`, `proxy_dot_create` and `signed`) in a separate session. While these instructions are not expected to be used as frequently as calls and deposits and thus are not as important to optimize, it is still interesting to note their efficiency. The `create` and `proxy_dot_create` instructions were evaluated in the manner used for calls above. The specific instruction instances `api.iset.create(1,"0x00")` and `api.iset.proxy_dot_create(1,"0x00")` were used to evaluate the instructions. They create an empty contract with an initial balance of 1 gwei, and cost 32420 and 33616 gas each respectively. As opposed to calls, there is no additional overhead for creating a contract via EVM instruction rather than directly with a transaction; the 32000 creation cost is paid by creating transactions as well. The savings versus the fixed transaction cost are consequently high, 98% and 92.3% respectively.
The `signed` instruction proved more difficult to profile. The old profiling approach of running many identical instructions in sequence could not be used; instead I needed to increment the nonce passed to the instruction for each invocation. Furthermore, each invocation's gas cost seemed to vary randomly. I tested it using the function `api.signed(address,deadline,tip,nonce,script)`, which constructs a `signed` instruction with a signature created by the private key of `address`, to which the Ethereum node must have access. Observing that the cost varied only by multiples of 64 (the difference between the data cost of zero and nonzero bytes), I conjectured that the variation in gas cost was due to variation in the number of zero bytes in the signature. Subtracting the data cost from the total gas cost of sequences of dummy signed scripts of the form
`api.signed(address,deadline,1,nonce,api.iset.stop())`, I was proven correct: the execution cost per instruction was always 9723. All variation in cost was therefore due to the data cost, and only the signature bytes varied in the instruction; the cost variation was therefore due to variation in the number of zero bytes in the signature. Averaging the data cost of 255 `signed` instructions yielded 5044 (rounded up). The average gas overhead of a `signed` instruction should therefore be approximately 14777, which can usefully be rounded to 14800. That is 70.5% of the fixed transaction cost. In other words, the savings provided by any two of the other instructions profiled exceeds the overhead imposed by `signed`. Furthermore, single non-paying rolodex-using calls, rolodex-using deposits and create instructions

do so as well. I conclude that the overhead of signed scripts is low enough to merit use instead of transactions for most use cases, provided there is a service which accumulates signed scripts from multiple signatories into a single transaction.

6.2.2 Hypotheses

H1 is confirmed through observation: the interpreter provides instructions for calling other addresses and creating contracts, and provides a unique identity to each user via the Proxy feature. Via the Signed feature, multiple signatories can share a single call to MultiCall.
H2 is confirmed by the above profiling; emulating calls, creates and payments using MultiCall is significantly cheaper than sending them directly, by 48 to 98 percent. Calling the interpreter imposes a fixed cost of 32222 gas, but for the instructions profiled the savings from executing 4 or more in a single transaction will exceed the cost. If multiple signatories share a transaction, they must each also pay the 14800 gas overhead of the `signed` instruction. The savings from any 2 of the instructions profiled exceeds the overhead of `signed`. I conclude that the shortest transaction sequence which it is not efficient to batch is no more than 4, which is below the limit of 50 set in the definition of H2.

6.3 Discussion

Though I have stated that the design objective of MultiCall is minimizing gas usage, divergence between the gas cost model and real computational costs poses a threat to the long-term usefulness of a practical interpreter smart contract. Certain gas cost optimizations encouraged by the protocol may not correspond to reduced computational cost for miners, rendering them more akin to tax dodges than optimizations and thus subject to crackdowns in the form of alterations to the gas model. It is possible that the savings provided by MultiCall do not correspond to real computational cost savings, and that if it achieves widespread adoption the protocol will be changed to render it worthless.
Advancements in hardware and optimization techniques may reduce the cost of certain contracts below their gas cost. A way for clients and miners to circumvent the gas model is for miners to accept lower gas prices when transactions invoke code which may be executed more efficiently than the gas model suggests. In such a scenario the gas model would still be useful for preventing denial of service attacks, but the fees paid for distributed computation would correspond more closely to the real cost. How to optimize smart contracts to minimize real computational cost is an open question.
Planned changes to the Ethereum protocol may soon reduce the interpreter's value: support is being added for a new and more efficient bytecode, eWasm [13]. The sharding feature planned for the Ethereum 2.0 protocol change [16], which divides the Ethereum blockchain into many sub-chains, may reduce redundant computation. The progress of Ethereum development is somewhat unpredictable, but may lower the computational cost of distributed execution and thus reduce the value of

optimizing it. Constant development of new interpreters in response to protocol changes may be required to stay relevant.

6.3.1 Generality of results

Evaluation shows H1 and H2 are fulfilled, but only for an instance of the batching interpreter implemented on Ethereum. One may ask whether the idea is generally useful, or is simply an artifact of arbitrary elements of Ethereum's cost model. There is reason to believe that there is a real overhead for transactions, since they contain some metadata and are each given a separate entry in a Merkle tree when they are included in a block. That entry also contains data on their effect. For blockchains which store their transactions submitted per block in a Merkle, submitting them instead as a single script may reduce costs. Whether a batching interpreter is efficient for an environment depends on its programming and cost model. First, it must be possible to emulate the effects of multiple transactions with smart contract functionality. That is not the case for Bitcoin, for example. Secondly, it must be cheaper to interpret and emulate a transaction than to send it. If a protocol natively supports batching, a batching interpreter is likely to be superfluous.

6.3.2 MultiCall and transaction auditability

In order to verify incoming payments and analyze activity on the Ethereum blockchain, users have an interest in cheaply scanning it for transactions matching a certain description. While each block contains a list of transactions and a corresponding list of *receipts* recording the effect of the transaction, the receipt does not include a trace of the calls made. By emulating multiple call transactions with a single transaction, MultiCall makes proving that payments or calls occurred somewhat more expensive. My examiner has suggested that an interpreter smart contract could be used to obfuscate transactions in this manner, and thus to attack the auditability of the blockchain. The key players involved are *light clients* and full nodes. Light clients are programs which do not run a full node recording the entire blockchain state but which can observe the chain by requesting cryptographic proofs about it from full nodes. Full nodes can construct a proof that a transaction in a block has a certain effect, of size proportional to the amount of blockchain state accessed by the transaction. The light client then verifies the proof by running the transaction with the given data, which is cryptographically proven to be valid. If a full block of transactions is batched, a light client requesting proof a specific call occurred would need to evaluate the entire block. While batching transactions does not make proving payments occurred intractable, it may increase computational costs enough to render doing so via less powerful machines such as Raspberry Pi impractical. This bears investigation.

The problem can be mitigated for MultiCall by exploiting its properties: transaction success (meaning full execution rather than reversion) is recorded in the receipt, and an invocation of MultiCall with a script does not succeed unless all of its instructions do. To prove a payment occurred within a MultiCall invocation with a given script, the light client could parse the script to find the payment rather than running the

entire call. For a contract whose purpose is obfuscation, cryptographic techniques such as those used by Zcash [6] could more reliably be applied to provide anonymity and payment secrecy combined with access to the Ethereum blockchain.

7
Future work

I intend to use the QuickCheck [8] Haskell property-based testing library to test the intended properties of the MultiCall interpreter, and use a formal verification tool to prove the contract's compliance with those properties once low-hanging fruit bugs have been eliminated with QuickCheck.
Interpreters targeting other Turing-complete blockchains may provide similar savings; that bears investigation.
Improvements to the code generator such as automatic efficient allocation of volatile and persistent variables and convenient definition of datatypes would be of interest.
MultiCall significantly reduces the gas cost of accessing contracts; further optimizations may be found in the contracts themselves. In the course of the thesis I have identified a number of techniques which could be used to reduce gas costs, such as cohousing multiple "virtual contracts" within a single contract, and caching persistent values for multiple accesses. An Erlang-like [5] language that expresses contracts in terms of processes could provide such optimizations alongside a natural way of expressing multi-phase contracts with restricted storage mutation limiting the ways in which things may go wrong.
A Turing-complete interpreter, while more difficult to implement, would provide exciting new capabilities. In particular, programs could be stored and then cheaply loaded instead of being resent as transaction data. With noninterference guarantees, multiple user-submitted programs could interact within the memory space of a single contract without the overhead of a `CALL` for each message transfer. A virtual smart contract could also benefit from reduced byte code size, as a higher-level interpreter may provide shared libraries. In addition, a Turing-complete interpreter could be used to implement Turing-complete state channels. A state channel [26] is a technique for reducing the cost of peer-to-peer cryptocurrency transactions by using a verification contract on-chain which holds funds in escrow on behalf of one or more pairs of users. A cryptographically verifiable ledger between two users is then maintained by the peers; they update it by sending signed messages. The funds can later be withdrawn by sending the ledger state to the blockchain, where it is verified. If one user sends an out-of-date ledger, the other user may complain and override it within a certain time limit before funds are withdrawn. Currently state channels significantly improve the speed and cost of peer-to-peer payments, but do not support smart contracts. By using an on-chain verification contract capable of Turing-complete interpretation, peer-to-peer ledgers containing Turing-complete programs could be verified. While fate channels [29] provide state channels that support specific casino games, I have not found an example of general-purpose Turing-complete state channels.

8

Conclusion

In this thesis I have described what is to my knowledge the first batching multi-instruction interpreter capable of running on the Ethereum blockchain. I have shown that, perhaps counterintuitively, an interpreter can be used to significantly reduce the gas cost of distributed computation by combining multiple transactions into one. To implement the interpreter, I designed and implemented an EDSL to generate EVM bytecode; it has proven to be a practical means of writing performant EVM code with low-level control of instructions in a convenient syntax. I used it to implement a prototype version of MultiCall and evaluated the savings provided by using it to emulate transactions (relative to sending transactions directly). I found them to be substantial: between 50 and 98%. I hypothesized that it was possible to implement an interpreter on the Ethereum blockchain which both emulates the full functionality of any typical sequence of transactions (H1), while reducing gas costs compared to sending those transactions individually (H2). In Evaluation, I confirmed those hypotheses. The potential commercial value of an Ethereum interpreter is significant, though further development is required for practical use.

Bibliography

[1] 2017. URL: https://github.com/TrueBitProject/lanai.

[2] URL: https://litecoin.org.

[3] URL: https://multisender.app/.

[4] URL: https://medium.com/@howekuo/send-erc20-tokens-to-thousands-of-addresses-out-in-1-minute-ca3551497cfb.

[5] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. ISBN: 193435600X, 9781934356005.

[6] Eli Ben-Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2014, pp. 459–474. ISBN: 978-1-4799-4686-0. URL: http://dblp.uni-trier.de/db/conf/sp/sp2014.html#Ben-SassonCGOMTV14.

[7] Manuel Chakravarty. *Keynote - Rethinking Blockchain Contract Development*. Accessed: 2019-06-05. 2019. URL: https://www.youtube.com/watch?v=RGJ4FvsUQzo.

[8] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. URL: http://doi.acm.org/10.1145/351240.351266.

[9] Conal Elliott, Sigbjørn Finne, and Oege de Moor. "Compiling Embedded Languages". In: *Journal of Functional Programming* 13.2 (2003). Updated version of paper by the same name that appeared in SAIG '00 proceedings. URL: http://conal.net/papers/jfp-saig/.

[10] Etherscan.io. *Ethereum (ETH) Blockchain Explorer*. Accessed: 2019-06-05. 2019. URL: https://etherscan.io/.

[11] ethgasstation.info. *Eth Gas Station | Consumer oriented metrics for the ethereum gas market*. Accessed: 2019-06-05. 2019. URL: https://ethgasstation.info/index.php.

[12] Shane Fontaine. *Introducing Batched Transactions on Authereum*. 2019. URL: https://medium.com/authereum/introducing-batched-transactions-with-authereum-f82dac9b62e7.

[13] The Ethereum Foundation. *ewasm Design Overview and Specification*. Accessed: 2019-06-05. 2019. URL: https://github.com/ewasm/design.

[14] The Ethereum Foundation. *List of Known Bugs*. Accessed: 2019-06-05. 2019. URL: https://solidity.readthedocs.io/en/v0.5.9/bugs.html.

[15] The Ethereum Foundation. *LLL Introduction*. Accessed: 2019-06-06. 2019. URL: https://lll-docs.readthedocs.io/en/latest/lll_introduction.html.

[16] The Ethereum Foundation. *Sharding introduction R&D compendium*. Accessed: 2019-06-05. 2019. URL: https://github.com/ethereum/wiki/wiki/Sharding-introduction-R&D-compendium.

[17] The Ethereum Foundation. *Solidity documentation*. Accessed: 2019-06-05. 2019. URL: https://solidity.readthedocs.io/en/v0.5.9/.

[18] Austin T. Griffith. *bouncer proxy*. 2018. URL: https://github.com/austintgriffith/bouncer-proxy.

[19] Austin T. Griffith. *Ethereum Meta Transactions*. 2018. URL: https://medium.com/@austin_48503/ethereum-meta-transactions-90ccf0859e84.

[20] Truffle Blockchain Group. *Truffle Overview*. Accessed: 2019-06-05. 2019. URL: https://www.trufflesuite.com/docs/truffle/overview.

[21] Truffle Blockchain Group. *trufflesuite/ganache-cli: Fast Ethereum RPC client for testing and development*. Accessed: 2019-06-05. 2019. URL: https://github.com/trufflesuite/ganache-cli.

[22] David A. Harding. *Saving up to 80% on Bitcoin transaction fees by batching payments*. 2017. URL: https://bitcointechtalk.com/saving-up-to-80-on-bitcoin-transaction-fees-by-batching-payments-4147ab7009fb.

[23] A. R. Hevner et al. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004), pp. 75–106. URL: http://search.ebscohost.com/login.aspx?direct=true&db=bsh&AN=12581935&loginpage=Login.asp&site=ehost-live.

[24] Billy Markus and Jackson Palmer. *Dogecoin*. 2013.

[25] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2009. URL: http://bitcoin.org/bitcoin.pdf.

[26] Joseph Poon and Thaddeus Dryja. *The bitcoin lightning network*. Accessed: 2019-06-05. 2016. URL: https://lightning.network/lightning-network-paper.pdf.

[27] Pablo Lamela Seijas et al. "Marlowe: implementing and analysing financial contracts on blockchain". In: ().

[28] The ZILLIQA Team. *The ZILLIQA Technical Whitepaper*. 2017. URL: https://github.com/Zilliqa/docs/blob/master/whitepaper.pdf.

[29] FunFair Technologies. *State channels in disguise?* Accessed: 2019-06-05. 2019. URL: https://funfair.io/state-channels-in-disguise/.

[30] Philip Wadler. "Monads for Functional Programming". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 24–52. ISBN: 3-540-59451-5. URL: http://dl.acm.org/citation.cfm?id=647698.734146.

[31] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger EIP-150 REVISION (759dccd - 2017-08-07)*. Accessed: 2019-06-05. 2017. URL: https://ethereum.github.io/yellowpaper/paper.pdf.