Cheap and secure metatransactions on the blockchain using hash-based authorisation and preferred batchers

Downloaded from: https://research.chalmers.se, 2026-04-04 18:50 UTC

Citation for the original published paper (version of record):
Hughes, W., Magnusson, T., Russo, A. et al (2023). Cheap and secure metatransactions on the blockchain using hash-based authorisation and
preferred batchers. Blockchain: Research and Applications, 4(2).
http://dx.doi.org/10.1016/j.bcra.2022.100125

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Blockchain: Research and Applications 4 (2023) 100125

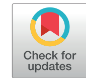
Contents lists available at ScienceDirect

Blockchain: Research and Applications

journal homepage: www.journals.elsevier.com/blockchain-research-and-applications

Research Article

Cheap and secure metatransactions on the blockchain using hash-based authorisation and preferred batchers



William Hughes [a,*], Tobias Magnusson [b], Alejandro Russo [b], Gerardo Schneider [a]

[a] Dept. of Computer Science and Engineering, University of Gothenburg, 41296, Gothenburg, Sweden
[b] Dept. of Computer Science and Engineering, Chalmers University, 41296, Gothenburg, Sweden

ARTICLE INFO

Keywords:
Ethereum
Domain-specific language
Interpreter
Gas optimisation

ABSTRACT

Smart contracts are self-executing programs running in the blockchain allowing for decentralised storage and execution without a middleman. On-chain execution is expensive, with miners charging fees for distributed execution according to a cost model defined in the protocol. In particular, transactions have a high fixed cost.

We present MultiCall, a transaction-batching interpreter for Ethereum that reduces the cost of smart contract executions by gathering multiple users' transactions into a batch. Our current implementation of MultiCall includes the following features: the ability to emulate Ethereum calls and create transactions, both from MultiCall itself and using an identity unique to the user; the ability to cheaply pay Ether to other MultiCall users; and the ability to authorise emulated transactions on behalf of multiple users in a single transaction using hash-based authorisation rather than more expensive signatures. This improves upon a previous version of MultiCall. Our experiments show that MultiCall provides a saving between 57% and 99% of the fixed transaction cost compared with the standard approach of sending Ethereum transactions directly.

Besides, we also show how to prevent an economic attack exploiting the metatransaction feature, describe a generic protocol for hash-based authorisation of metatransactions, and analyse how to minimise its off-chain computational and storage cost.

1. Introduction

Distributed ledger technology (a type of database spread across multiple sites or participants), and blockchain in particular, promise to revolutionise the industry. The blockchain is the platform on top of which programs (smart contracts) run that allow mutually distrustful parties to transact and contract without a middleman. Such programs interact among themselves, as well as with users, via transactions that are transparent, traceable, and irreversible.

There are many different blockchain platforms and smart contract languages. Although our results may be applicable to different blockchain platforms (see Section 9), the discussion that follows and the rest of the paper focus on Ethereum [1].

In Ethereum and similar public blockchains, users pay a transaction fee for the execution of smart contracts to compensate for the cost incurred to process and validate them. This cost is calculated in Ethereum based on the computations performed by the program (the instructions executed by the program), and it is expressed in a unit of account called *gas*. Gas is charged for each transaction included in a block, each byte uploaded and Ethereum Virtual Machine (EVM) instruction executed. The gas used per block and the rate of block creation are limited by the protocol —currently to around 30 million and about once every 13 s, respectively [2]. On-chain processing capacity is valuable, expensive, and strictly limited; miners charge users via transaction fees proportional to the gas used. This cost is significant, several dollars for the fixed transaction cost alone [2]. The cost of the computation of each instruction (fees) is regularly updated in Ethereum's yellow paper [1] (see Appendix G of Ref. [1]).

Let us consider a subset of the smart contract Token (see Fig. 1) as an example. The smart contract implements a ledger of tokens that depositors can transfer to each other. While simple, the program is not trivial: it is very close to real implementations of the popular ERC-20 [3]

* Corresponding author.
E-mail addresses: hughes@chalmers.se (W. Hughes), tobmag@chalmers.se (T. Magnusson), russo@chalmers.se (A. Russo), gersch@chalmers.se (G. Schneider).

https://doi.org/10.1016/j.bcra.2022.100125
Received 15 August 2021; Received in revised form 16 December 2022; Accepted 18 December 2022
2096-7209/© 2023 The Authors. Published by Elsevier B.V. on behalf of Zhejiang University Press. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

```
contract Token {
 ...//Other variables
 mapping(address => uint)accounts;
 function xfer(address recipient, uint amount) public {
  uint senderAmount = accounts[msg.sender];
  if (senderAmount < amount)revert();
  accounts[recipient] += amount;
  accounts[msg.sender] = senderAmount - amount;
 }
 ...//Other methods }
```

Fig. 1. The smart contract Token.

interface[1]. Balances of the Token token are stored in the accounts object, a mapping from Ethereum addresses to 256-bit unsigned integers. The contract supports an xfer method, which depositors can use to pay other recipients integral amounts of the token.

The standard way for users to interact with smart contracts today is to send an individual transaction corresponding to a method (or function) call. Executing the function xfer, for instance, costs approximately 34000 gas[2]. Every transaction has a fixed cost of 21000 gas, which is around two-thirds of the cost of the xfer operation. To see the order of magnitude of the cost of executing a transaction, it is useful to contrast it with other operations. For instance, a JUMP instruction costs 8 gas, while an ADD instruction costs 3 gas.

Consider a user that wishes to transfer many tokens to different recipients, such as the operator of an exchange or mining pool. It is clear that in such a case, the user will have to pay a fee proportional to the number of transactions executed. Therefore, reducing their gas costs by even a small proportion may provide significant monetary savings.

In the example above, one can see that a large part of the execution cost comes from the fixed per-transaction cost. Our aim in this paper is to find a practical and systematic way to reduce smart contract gas consumption. We do so by proposing an architecture to do transaction batching, and we provide a proof-of-concept implementation for the blockchain Ethereum.

Our approach provides more than a small cost reduction. Indeed, for the example above, it reduces the marginal cost of a token transfer by 67.7% (10912 gas), or the total cost by 59.6% when making 10 transfers (the solution has an overhead of 27326 gas, which can be amortised by making more transfers).

To achieve that, in Ref. [4], we have presented MultiCall, an Ethereum smart contract that reduces the gas cost of on-chain execution by emulating multiple transactions using a single one (a process known as batching). Batching is not novel; existing batchers emulate multiple calls from a single sender using Solidity, the most popular Ethereum smart contract programming language. MultiCall differs from prior batchers in that it is a proper multi-instruction interpreter, and its instruction set can emulate functionality equivalent to an arbitrary block of transactions in a single transaction.

The interpreter presented in Ref. [4] was quite efficient: the volatile memory accesses, arithmetic, and branching required for interpretation are much cheaper in the Ethereum cost model than accessing the ledger state or verifying signatures. The overhead of interpretation when batching transactions was therefore manageable.

In this work, we present a revised and extended version of Ref. [4]. The main contributions of this paper are:

i) An architecture to improve gas consumption based on a middleware (MultiCall and its off-chain API code) that can emulate arbitrary sequences of transactions (Section 3).
ii) A new cryptographic protocol, HBAuth, is designed to reduce the cost of metatransactions using hash-based authentication (Section 4).
iii) A technique to protect metatransaction batchers from an economic attack that invalidates their transactions (Section 5).
iv) A proof-of-concept implementation of the MultiCall Ethereum smart contract, including the new features from ii) and iii) (Section 6).
v) An evaluation of our approach to show its feasibility and advantages. Our evaluation of MultiCall's performance relative to unbatched transactions shows a saving of between 57% and 99% of the fixed per-transaction cost compared to sending individual transactions. We also compare MultiCall's performance to two other existing batchers (one of academic origin and the other industrial), with favourable results. Finally, we evaluate our new hash-based metatransaction mechanism and find it reduces the overhead of metatransactions by approximately 40% (Section 7).
vi) An analysis of the optimal off-chain caching scheme for HBAuth based on existing work on efficient hash-chain traversals. We also found the optimal choice of the hash-chain length under varying conditions (Appendix A).

Contributions ii), iii), and vi) are new to the extended version. Contribution i) has been extended with a description of the design of the new functionality provided by ii) and iii). Contribution iv) has been extended with a description of the code generation DSL in which MultiCall was written. Furthermore, we added detail on how the DSL was used to implement the functionality of MultiCall, including both the original functionality and that provided by ii) and iii). In contribution v), the comparison with the iBatch batcher [5] and the evaluation of the new hash-based metatransaction feature are new to the extended version. Finally, we added a section on the security of MultiCall (Section 8) and extended the discussion (Section 9).

In Section 2, we present some preliminaries on Ethereum, Solidity, metatransactions, and the uses of hashes in cryptography. In Section 3, we describe the design decisions underlying the MultiCall smart contract. In Sections 4 and 5, we describe the new HBAuth protocol, which permits hash-based metatransactions, and the preferred batcher feature, which protects batchers from economic attack, respectively. In Section 6, we explain the low-level implementation of MultiCall and its new features in more detail. In Section 7, we evaluate the contract's performance; security is discussed in Section 8. We discuss the overall results and future work in Section 9. Related work and our conclusion are presented in Sections 10 and 11. In Appendix A, we analyse how to minimise the off-chain cost of the HBAuth protocol.

2. Background

The fundamental purpose of the Ethereum ecosystem is to enable parties to transact: to enter into contracts, to interact with those contracts (for example by making choices or executing clauses) and to make payments. The contracts entered into using distributed ledger technology are computerised and self-enforcing; such contracts are termed smart

[1] ERC-20 is a technical standard for smart contracts that record token balances, requiring that they expose certain standard methods and that the token balances obey certain rules. Compliant token contracts can be accessed in a uniform manner, enabling code sharing and facilitating trading between multiple tokens.

[2] 33817 to be exact, but it could cost less if the address of the recipient contains zero bytes; the addresses we used did not. Note that we used Solidity compiler version 0.5.16 (the default for Truffle v5.5.19). The private chain fork used is Muir Glacier; Ganache does not yet (as of v7.2.0) support the London or Berlin forks.

Table 1
Ethereum's layers of abstraction.

Layer	Identities	Payments	Contract initiation	Contract execution
Abstract	Users	Payment	Offer to counterparties	Choice, complaint etc
Solidity	Addresses	.transfer method	Constructor call	Method call
Primitive	Addresses	Call txn/ instruction	Create txn/ instruction	Call txn/ instruction

contracts.[3]

Like many computer systems, the Ethereum ecosystem can be thought of as implemented in multiple layers of abstraction (see Table 1). The highest is what might be called the abstract layer, which consists of payments and contracts that the parties wish to make. It is implementation-agnostic, so it could, for instance, be implemented as a centralised ledger or using scalable decentralised solutions such as state channels [6].

The most popular means of implementing abstract contracts on the Ethereum platform is using the smart contract programming language Solidity [7]. Solidity is a statically typed object-oriented language that lets the user write Ethereum smart contracts as objects that expose methods and contain persistent states. Abstract arrangements, such as an escrow agreement or a new issue of tokens, then correspond to one or more Solidity contract objects (or states within such objects). Users offer new arrangements to counterparties by creating Solidity contracts and interact with those contracts using method calls. Ether payments are treated as a special .transfer method.

The Solidity layer is in turn translated into primitive Ethereum transactions. Transactions are an indivisible unit of interaction with the blockchain; each block contains a sequence of transactions. There are two Ethereum transaction types, create and call. Solidity constructor calls (which instantiate a new contract) are translated into create transactions, and method calls are translated into call transactions. When appended to the blockchain, transactions modify the distributed ledger by performing a call or create action, respectively, from the signatory key's account.

There are two types of accounts on the Ethereum ledger: externally owned accounts (EOAs) and smart contracts. EOAs are controlled by an Ethereum private key, and their address is the corresponding 160-bit public key. When transactions are added to the blockchain, they cause calls or creates to be performed from EOAs. Calling an EOA transfers the Ether value specified in the call from the caller to the callee account. Smart contract accounts contain additional states: immutable bytecode and a mutable persistent storage space. When a smart contract is called, in addition to optionally effecting an Ether payment, the contract's bytecode is interpreted by the EVM. The EVM is a Turing-complete stack machine with instructions specialised for the blockchain environment. In particular, the EVM supports instructions for querying the bytestring *calldata* sent in the call, querying the address of the caller, and performing calls and creates with equivalent effect to transactions. Solidity method identifiers and arguments are encoded as calldata, the msg.sender expression is compiled to the EVM CALLER instruction, and method and constructor calls in the text of a contract compile to EVM CALL and CREATE instructions.

Payments in the native cryptocurrency Ether are a special case of call transactions with empty calldata. Ether payments may be translated directly from the abstract layer to transactions or may be viewed as the special .transfer Solidity method call. Payments in user-issued tokens such as those managed by ERC-20 contracts are translated to method calls to that contract.

Let us see how the above works through an example. When Alice wishes to perform the abstract action of giving Bob some Token tokens (see Fig. 1), they specify an Ethereum public key BobAddr controlled by Bob and instruct their Ethereum client software to make a payment of Token to that address. The action is translated in their client software to a Solidity method call to the token contract's address TokenAddr and then to an Ethereum call transaction, signed by some private key controlled by Alice whose corresponding public key is AliceAddr. Finally, the transaction is broadcast to the Ethereum network and ultimately mined and included in the blockchain. That modifies the distributed Ethereum ledger state, effecting a call action from the EOA at address AliceAddr to the smart contract at address TokenAddr. This process is illustrated in Fig. 2.

We are interested in reducing the gas cost of execution on the Ethereum blockchain. To optimise execution on the Ethereum ledger, one must first understand its capabilities and cost model. A detailed description of Ethereum's cost model and semantics can be found in the Ethereum Yellow Paper [1]. We will not present Ethereum's cost model in detail but rather what is relevant to understanding how MultiCall works, namely, the following key facts:

1. Call transactions cost 21000 gas, not including the cost of contract execution.
2. Call instructions executed by a contract (which have a practically equivalent effect) cost significantly less.[4]
3. Create transactions cost 53000 gas: the fixed transaction cost and an additional creation cost of 32000 gas.
4. Create instructions executed by a contract (which have a practically equivalent effect) cost only 32000 gas.
5. Compared to instructions that modify the persistent ledger state (such as calls and storage writes), the arithmetic and control flow EVM instructions needed for interpretation are very cheap.
6. The cost of uploading data (such as scripts) is also relatively low, at 16 gas per nonzero byte and 4 per zero byte.
7. Signature verification costs 3700 gas, not counting the 65 bytes of signature that must be uploaded. Hashing costs only 30 gas, plus 3 gas per 256-bit EVM word hashed.

2.1. *Metatransactions in brief*

Metatransactions [8] are authorisations to perform some actions that are verified and executed in smart contracts. They are analogous to native transactions but are attractive to use instead because they allow users to send transactions without having to first hold the blockchain's native cryptocurrency. They also allow more effective batching. To be secure, transactions (meta and otherwise) must be associated with some cryptographic proof that the principal has approved them. Also, when authorising a non-idempotent action such as payment, they must be *replay-protected*. Ethereum implements the former feature using the ECDSA signature scheme and a nonce associated with each account controlled by a private key [1]. The conventional approach to implementing metatransactions in Ethereum smart contracts is to use the same signature scheme as the Ethereum protocol (helpfully provided as a primitive) and analogously associate users with nonces.

[3] The term refers both to the entirety of self-enforcing arrangements, including off-chain components, and to individual on-chain program objects on the blockchain. The relation between smart contracts in the general and narrow sense is analogous to that between programs consisting of multiple OS processes and each individual OS process, which is also a program.

[4] Gas costs frequently change as Ethereum protocol developers seek to defend the network against denial of service attacks. The cost of calls has recently changed to cost 2600 gas the first time a contract is called in a transaction and 100 gas subsequent times. Our evaluation tool does not yet support this new cost model, but we expect it to improve MultiCall's performance.

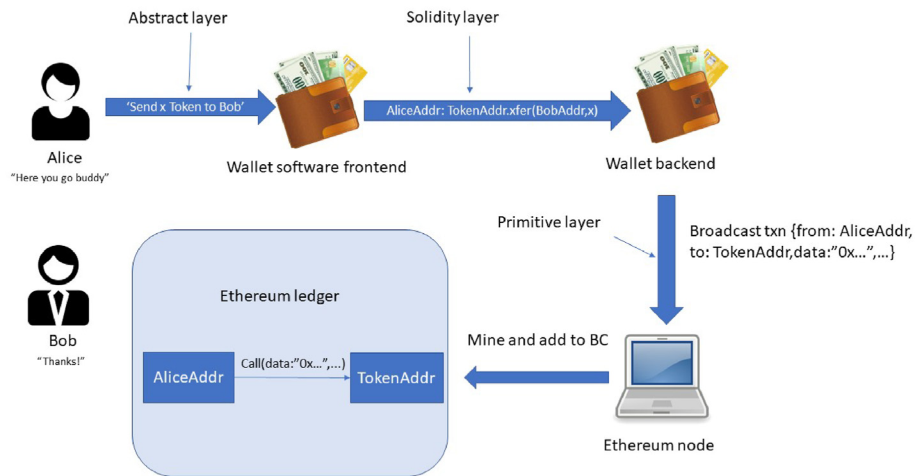


Fig. 2. An illustrated payment of a user-issued token.

2.2. *Hash-based cryptography*

Collision-resistant hashes are a fundamental component of cryptography in general and distributed ledger technology in particular. Beyond being used for Bitcoin's iconic proof-of-work protocol [9], hashes are also used to make immutable, constant-size references to particular data. Any hashes contained in the data can in turn serve as immutable references, forming a graph. An extreme example of the compression provided by hashes is that entire blockchain histories of many terabytes can be referred to using a 256-bit hash. Hashes' compressive property is used in HBAuth to commit to a sequence of metatransactions to perform using its hash.

A third use of collision-resistant hashes, highly relevant to our work, is as a one-time password. By choosing a random number and then making its hash or the last element of a long chain of hashes produced from it public, a party gains the ability to send a signal that no one else can by revealing a preimage of the published hash. That is used in atomic swaps [10] to exchange cryptocurrency across different blockchains.

3. MultiCall design

Our solution provides value by batching transactions. To do so, we need to modify the Ethereum transaction submission workflow in the wallets and on the Ethereum ledger. Instead of converting an action into a transaction and sending it immediately, it is converted to a MultiCall instruction and saved. Multiple instructions can be concatenated and signed to create a metatransaction [8].

The MultiCall metatransactions can then be sent to a batching server, which sends their concatenation to MultiCall in a single transaction. MultiCall acts "under the hood", so the user need not be aware of it. As shown in Fig. 3, MultiCall and its associated off-chain code work as middleware between the wallet frontend and the backend and add some additional machinery to aggregate instructions before delivering them to Ethereum nodes.

One of the key features of MultiCall is the ability to batch transactions from multiple users in a single call while providing each user with a unique identity. Consider the Token example again: each user should be able to transfer tokens from their account and only their account. This is done by querying the caller address in the xfer method, msg.sender (see Fig. 1), and deducting from their account. Using the caller address to authenticate and identify users is a common smart contract pattern. Because smart contracts often deal with the transfer and use of assets, method calls that require user authentication are a key part of smart contract functionality. MultiCall would therefore not be very useful if it did not provide each user with a unique identity; it does so through proxies. Proxies are a well-known Ethereum development pattern: puppet contracts that perform calls and create actions when commanded to do so by their controller. That may be achieved by checking the address of the proxy's caller or by using metatransactions. In the case of MultiCall, its proxies check that MultiCall is the caller. Each proxy belongs to a

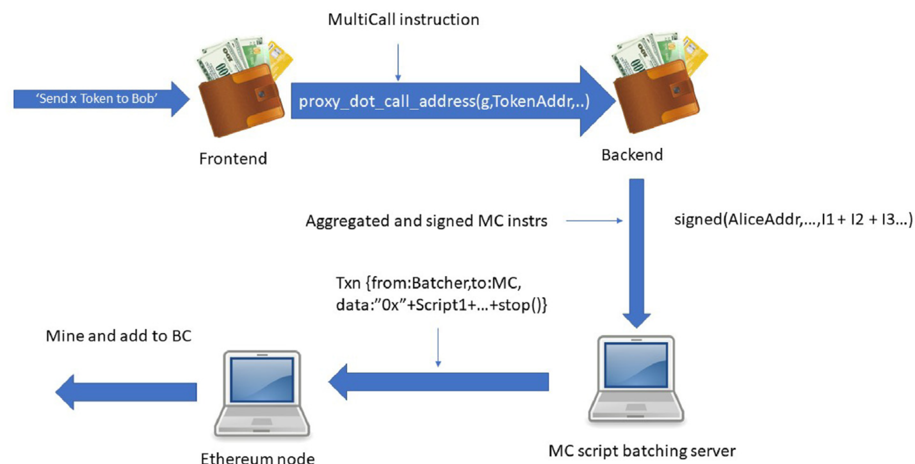


Fig. 3. An illustration of the modified workflow using MultiCall. In this case, the client submits a signature-based metatransaction to their batcher. The batcher later sends a transaction containing the metatransaction to MultiCall, causing the metatransaction to be run.

single MultiCall user (represented as an Ethereum address); MultiCall will only command a proxy to create a contract or call an address when executing on behalf of the address that owns it (we explain how that works in more detail later).

The MultiCall contract is written in a low-level EVM code-generating DSL rather than Solidity. That made it easier to achieve an efficient data layout for instruction arguments, as we avoided the inefficient abstractions and calling convention of Solidity; the DSL is described in more detail in Section 6.1.

3.1. Improved metatransactions

As an extension of earlier work [4], metatransaction functionality has been improved. Alongside the conventional signature-based approach where users simply sign the metatransactions they wish to approve, support has been added for hash-based metatransactions. Such metatransactions are implemented using a novel protocol we call HBAuth, described in detail in Section 4. The protocol is illustrated in Figs. 4–6: first, users sign their metatransactions as normal, then the batcher (an off-chain server that aggregates metatransactions for a fee) commits to perform those particular metatransactions, and finally, the users reveal preimages to hashes stored in their accounts to approve the committed metatransactions. The process is summarised in Fig. 7. HBAuth replaces signature verification with a check that a preimage corresponds to a hash, significantly reducing the metatransaction verification cost.

To prevent an economic attack where malicious parties could use metatransaction replay protection to invalidate batcher transactions, a new feature called preferred batchers has been added to MultiCall. It is described in more detail in Section 5.

3.2. Instruction set

MultiCall is a smart contract implementing a specialised interpreter that, when called, interprets its bytestring argument as a sequence (or script) of MultiCall instructions and executes each sequentially. Its instruction set is designed purely for transaction batching, emulating the functionality of several transactions in one. Unlike interpreters such as the JVM or EVM, MultiCall is deliberately not Turing-complete in order to ease design and future formal verification. It has thirteen principal instructions of interest to the user (described below), not counting admin-only instructions to transfer ownership of MultiCall. MultiCall executes on behalf of a single Ethereum address at a time, charging any Ether costs incurred when executing instructions to the user's account.

Note that the "CA" mentioned in Fig. 4 and the names of instructions refers to committed actions. Batchers commit to performing actions (batches of user metatransactions) as part of the HBAuth protocol, which is described in more detail in Section 4.

1. call_address(gas, address, eth_value, data) performs an EVM CALL with the given arguments directly from MultiCall; it is useful for Ether payments to users and method calls that do not require authentication.
2. proxy_dot_call_address(gas, address, eth_value, data) instructs the user's proxy to perform the call instead.
3. create(eth_value, data) creates a contract directly from MultiCall.
4. proxy_dot_create(gas, eth_value, data) creates a contract from the user's proxy.
5. deposit_address(address, eth_value) credits the given address' account in MultiCall's persistent storage; it is a cheaper way of paying a MultiCall user Ether than making a call.
6. createProxy(eth_value) creates a new proxy with the given Ether endowment.
7. setBatcherTo(batcherID) lets the user choose which batcher they want to batch their metatransactions. Setting batcherID to 1 indicates no batcher is desired.
8. becomeBatcher() is how new batchers are made: the instruction converts a client account to a batcher, allocating a new ID for it.
9. setCA(hash) lets batchers commit to performing a particular script of metatransaction instructions; it is used in the HBAuth protocol.
10. startCA(script) takes a set of MultiCall instructions as its argument. If the user is a batcher and their committed hash matches the script, the script is run, and metatransactions in it can be executed. Its complement endCA() disables metatransactions again.
11. startMetatxn(client, tip, script) performs the MultiCall instructions in script on behalf of the client and pays the tip to the batcher to incentivize them to run the metatransaction. Its complement endMetatxn() returns MultiCall to execute on behalf of the caller.
12. setImage(hash) is used by clients to set the hash whose preimage they reveal as part of using hash-based metatransactions.
13. endCall() ends the call to MultiCall.

Intuitively, every Ethereum transaction conceived to create or call a contract can be mapped into one of MultiCall's instructions create, proxy_dot_create (if creator authentication is required), or call_address, proxy_dot_call_address, respectively. Furthermore, Ether payments may either be translated to call instructions or a deposit_address instruction. Multiple instructions may be concatenated into scripts before being signed and sent to a batching server. The batching server, in turn, collects

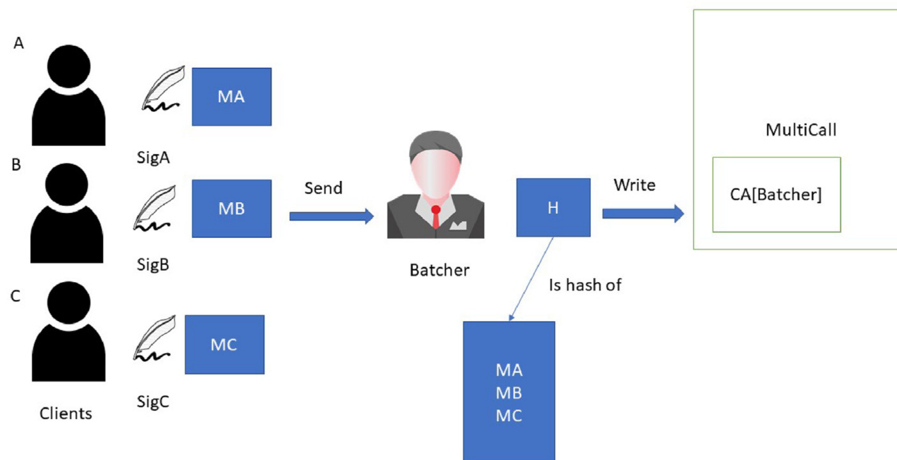


Fig. 4. The first phase of HBAuth: clients sign metatransactions and send the metatransactions and signatures to the batcher server. The batcher commits to running the metatransactions by storing their hash on-chain.

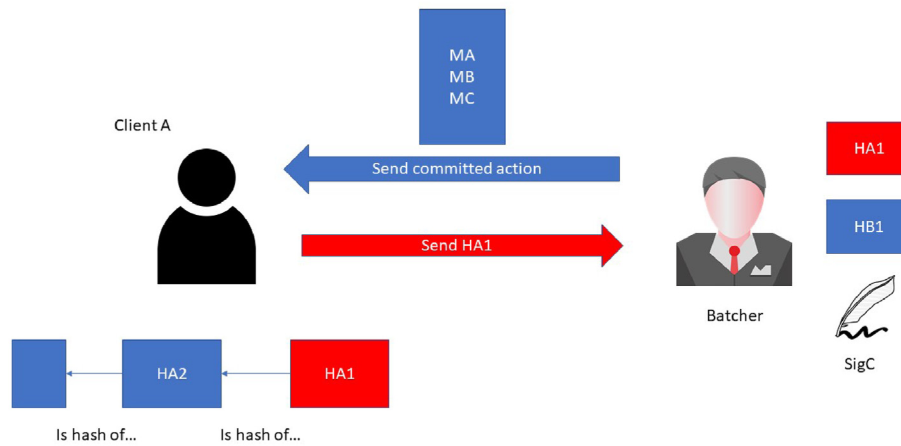


Fig. 5. The second phase of HBAuth: the batcher shows the committed metatransactions to each user, receiving the preimage to the image in the user's account if the client is satisfied. Each HBAuth user stores a long secret chain of preimages; in A's case, HA1 = hash(HA2), then HA2 = hash(HA3) and so on. The hash of HA1 (let us call it HA) is stored on-chain in the user's MultiCall account record and not off-chain by the user. Note C has not sent their preimage; perhaps they're offline, or just uncooperative.

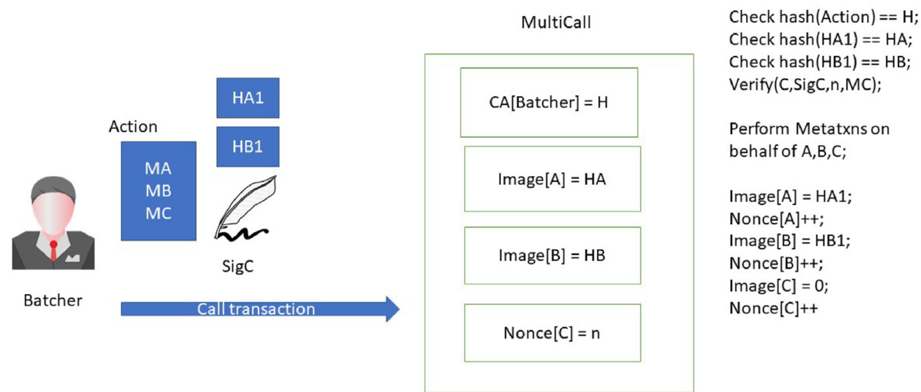


Fig. 6. The third and final phase of HBAuth: the batcher uploads the metatransactions, using revealed preimages to authorise them when possible and signatures otherwise. We refer to the images stored in A and B's accounts as HA and HB, respectively. At this stage, the batcher has verified off-chain that they are the hash of HA1 and HB1.

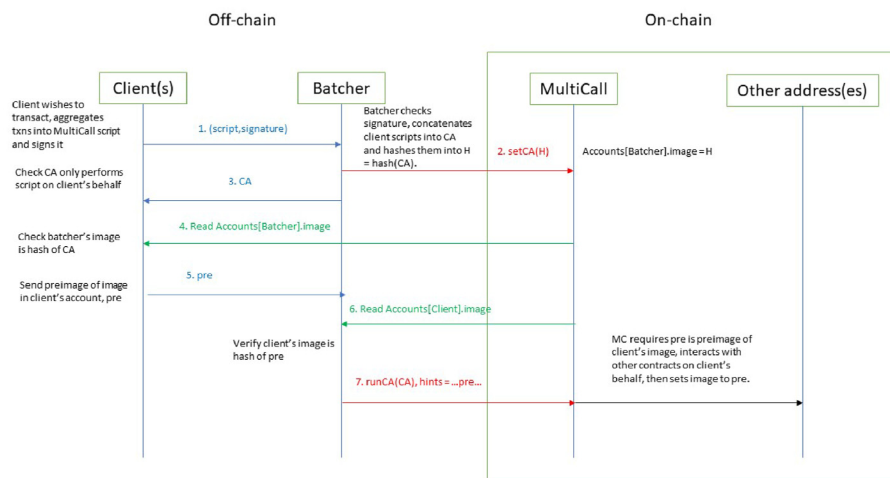


Fig. 7. The three phases are summarised in a sequence diagram. Blue arrows indicate off-chain messages, red on-chain transactions, and green off-chain reads of the blockchain state. The latter require no gas. Note that the storage write and fixed transaction costs of setCA and runCA can be amortised by running the last committed action and committing a new one in the same transaction. The preimage pre is the last element of the chain of secret hashes the client stores, having created them earlier.

multiple scripts and then broadcasts a call transaction (signed by its own key), which calls the MultiCall interpreter smart contract. The calldata of the transaction contains the scripts and cryptographic proofs that the senders authorised them. When mined, this transaction effects a call to MultiCall, which then performs the actions authorised by the users, such as making method calls and payments. Like Token, MultiCall contains a mapping from Ethereum addresses to accounts in its persistent state; Ether payments performed on behalf of a user are deducted from the user's account.

3.3. Revisited example

Suppose Alice wishes to make payments of Amount1 to Amount10, respectively, of the token tracked in the Token contract to 10 different recipients, Bob1 to Bob10. Each payment is translated into a MultiCall instruction

proxy_dot_call_address(G, TokenAddr, 0, C(BobN, AmountN))

where G is some reasonable gas limit chosen by Alice or the wallet and C(BobN, AmountN) is the calldata corresponding to the method .xfer(BobN, AmountN). BobN and AmountN refer to one of the recipient addresses and the number of tokens to pay them, respectively. Note that a proxy call is appropriate because Token's method .xfer requires authentication to spend from the caller's account. The wallet could at this point wait for more instructions to be added before signing them, but let us suppose it does not. It then selects an appropriate deadline and tip. A reasonable choice would be the current time plus 1 h and the cost of the given instructions at the current gas price. The wallet then appends the instruction endMetatxn to the list of instructions and signs the resulting script S with their account's nonce, deadline D, and tip T. Alice then sends the metatransaction startMetatxn(Alice, T, S) and signature to their batcher. Alice can then optionally use HBAuth to reduce the cost of their metatransaction, in return for increased latency.

Another choice would be for Alice to send the instructions to MultiCall herself without going through a batching server. In that case, profiling shows it costs 136516 gas, as opposed to 338170 to send the payments individually[5]. Considering the fixed cost of the transaction to send the script, 59.6% of the fixed transaction cost was eliminated relative to sending 10 transactions individually. Blocks may contain several hundred transactions (and when using MultiCall you could effectively make many more within the block gas limit), so fixed overheads would be negligible in practice if using a batching server.

3.4. Deployment

To use the MultiCall smart contract, an instance of it must first be uploaded to the blockchain with a created transaction. For each individual user to gain access to the full functionality of MultiCall, they must allocate an account and create a proxy. Allocating an account is achieved by depositing Ether to MultiCall with a call transaction or receiving a deposit from another user and then setting the image used by HBAuth in a signature-based metatransaction using the instruction setImage. Creating the user's proxy is done by running the createProxy instruction. Depositing funds is necessary to be able to pay batcher tips on-chain; off-chain payments could also be used without modification to MultiCall but would require some engineering effort by the parties.

4. Hash-based authorisation

Hash-based authorisation (HBAuth) is a cryptographic protocol developed by the first author for authorisation of user transactions on distributed ledgers. It is intended to provide a lower-cost complement to the standard signature-based approach. Verifying the transactions of external users is a fundamental and frequently performed task of distributed ledgers, so we hope that reducing its cost will bring significant benefits. The gist of the protocol is that instead of providing a signature of the transaction they wish to approve, the user reveals the preimage of a hash they chose earlier and associated with their account. Verifying a preimage is a much cheaper operation than verifying a signature (requiring only a hashing and equality check), allowing the cost of transaction authorisation to be significantly reduced. By keeping their preimage secret, the user can ensure that only they may approve transactions from their account.

However, HBAuth comes with a downside: unlike a signature, the preimage committed in advance of choosing what transaction to authorise cannot contain any reference to the chosen transaction. Revealing or not revealing the preimage can be viewed as communicating one bit of information: approval or disapproval. What transaction the approval refers to must be chosen some other way prior to preimage revelation.

Thankfully, we have a distributed ledger at our disposal to which the transaction to be executed can be committed. HBAuth dictates that the API for committing the transaction prevents timely modification of it once committed. That allows the user to safely reveal their preimage, confident that it will only be used to approve their chosen transaction. We describe our protocol in more detail below.

4.1. On-chain state and API required

Rather than being inherently tied to MultiCall, HBAuth is implementable by any on-chain system containing the requisite state and exposing the following API. The conventional approach for APIs on Ethereum would be to implement each function as a Solidity method; instead, we have implemented each as one or more MultiCall instructions.

There are two types of users in the protocol: clients and batchers. Batchers commit and run metatransactions on the ledger, and clients approve them. In MultiCall, there can be multiple batchers, and each user chooses one. However, that is not strictly part of the HBAuth protocol: one could also choose to have a single central batcher or have users mine for the right to batch clients using proof of stake, for example. The significance of choosing a batcher is described in more detail in Section 5.

4.1.1. Required state

In MultiCall, all the information pertinent to each user (client or batcher) is stored in its own storage slot in the MultiCall contract's persistent storage. However, the HBAuth protocol does not fix the storage layout. For example, the state could be stored in a Merkle tree or in separate contracts in another implementation; what matters is that it exists. The necessary data are listed below.

- Per batcher: a mutable variable CA[batcher], either zero or containing a hash.
- Per client: a mutable variable image[client], either zero or containing a hash; a mutable numeric variable nonce[client].

In fact, we store client images and batcher committed actions (CAs) in the same field in their account struct, image. Its meaning is overloaded based on a bit indicating whether the account is a batcher or a client.

4.1.2. Required API

The HBAuth protocol uses the following API. The functions setCA, runCA and setImage must be available to users; runMetatxn is internal and used by runCA.

- setCA(h), run by the batcher to commit the hash of an action (in other words, a set of metatransactions).
- runCA(action, proofs), run by the batcher. The argument action is a sequence of metatransactions; if the hash of action matches the

[5] The cost may vary slightly due to the number of zero bytes in addresses; a nonzero byte in calldata costs 16 gas, while a zero byte costs only 4.

```
setCA(h):
 require caller is a batcher
 if CA[caller] == 0:
  CA[caller] = h

runCA(action):
 require caller is a batcher
 if hash(action) == CA[caller]:
  perform action
  CA[caller] = 0

setImage(h):
 if run inside metatransaction
       for client:
  image[client] = h
  if run by the client directly:
   if the client has no batcher:
    image[client] = h
   else:
    require(!image[client])
    image[client] = h
```

```
runMetatxn(client,script,tip,proof):
 //Called as part of runCA
 if proof is (signature,n):
  require(ecrecover(signature,script,n,tip)
          == client)
  require(nonce[client] == n)
  image[client] = 0
 else proof is preimage:
  require(image[client] == hash(preimage))
  image[client] = preimage
 nonce[client]++
 pay tip from client to batcher
 perform script on behalf of client
```

Fig. 8. Key instructions used by HBAuth and their behaviour.

committed hash CA[caller], then they are executed using runMetatxn. The proofs are used to authorise the committed metatransactions; the metatransactions themselves do not contain the requisite proofs.

- runMetatxn(client, script, proof) performs the actions in script on behalf of client given proof of its validity. The proof may be either a signature and nonce or a preimage. The action argument to runCA consists of data specifying invocations of runMetatxn; in the case of MultiCall, it is simply a string of MultiCall instructions.
- setImage(h), run by the client (in a metatransaction or directly), setting their image variable to a new value. This is used to upload a new hash to their account.

Fig. 8 below sketches the behaviour of instructions. The signature expiration check in runMetatxn and some minor details have been elided. The runCA and runMetatxn functions are in fact implemented using two instructions each (start- and end- CA and Metatxn), across which their logic is split. That is because MultiCall contains no subroutine calls for efficiency, instead performing only direct jumps.

4.2. Off-chain protocol

Because metatransactions must be committed before approval, the off-chain protocol is more complex than with signature-based metatransactions, which can be sent in one step. Each batcher is intended to communicate with many clients. One instance of the protocol between client and batcher is outlined below.

4.2.1. Preparation

Clients start without an image in their account; when they have no image, they must use a signed metatransaction to set it. First, they must generate a random value X and compute a long chain of hashes $(\mathtt{hash}(X), \mathtt{hash}(\mathtt{hash}(X)),\ldots,\mathtt{hash}^N(X))$. The final hash is set as the first preimage. Let us call the sequence of hashes the client maintains images. Clients must also acquire a batcher. In the case of MultiCall, the owner of MultiCall (initially its creator) is the default batcher.

The protocol in the normal case.

1. The client wishes to send a metatransaction. They sign it and the nonce in their account using their Ethereum private key, sending the signature and metatransaction to their batcher.
2. The batcher verifies the signature against the script and the client's nonce. They also check that the client has sufficient funds to perform the transaction successfully (and to pay a tip to the batcher) and that the tip paid is sufficient for the gas used. If satisfied, they schedule the metatransaction for inclusion in a committed action.
3. Later, the metatransaction is included in a committed action. The batcher runs setCA(h), where h is the hash of the committed action. The batcher then sends the committed action to the clients whose metatransactions were included.
4. They client checks that the committed action sent to them matches the hash which the batcher has committed on-chain. The client then verifies that the committed action does what they want on their behalf and nothing more. If satisfied, they reveal the preimage of the image in their account to the batcher. The preimage is obtained by popping off the last element of the sequence images.
5. When receiving a preimage from a client, the batcher verifies that it matches the image stored in the client's accounts. Finally, the batcher runs the committed action using runCA, using preimages when clients have chosen to reveal them and signatures when they have not.

4.3. Edge cases

In real systems, the failure of a node and the loss of its state in memory is always a possibility. To protect against the possibility of batchers forgetting their committed action and thus being deadlocked, there is a mechanism for batchers to reset CA. There is a warning period W_1 starting from when the action is committed, intended to allow clients to respond to attempts to reset the CA. The warning period is necessary because otherwise any revealed preimages would be a *carte blanche* to drain the revealing clients' accounts. If the client sees that the batcher is not running the CA and the warning period for reset is coming to an end, they must themselves escape and end the client-batcher relationship before the batcher can reset the CA. That in turn has a warning period W_2 to prevent clients from being able to invalidate transactions, so they must initiate the process in time (within W_1–W_2 of the last CA commit).

When clients have only one preimage left, they must also set a new image alongside uploading their last preimage. How long a chain of hashes to use to minimise gas, storage and compute cost is explored in Appendix A.

4.4. Design considerations

The protocol has been designed to prevent misbehaviour by either the client or the batcher. One may ask, for example, why a signature of the metatransaction is required in Step 1 of the off-chain protocol when it is

not used in the ideal case. That is because the batcher may not run a committed action that attempts to do something on a client's behalf without a cryptographic proof of the client's approval. If a signature were not requested in advance, the batcher would be subject to a denial of service attack where a client requests the batcher add a metatransaction but then does not approve it, preventing the batcher from running their committed action for the warning period W_1.

The batcher is able to commit metatransactions from multiple clients at once. That is for performance reasons, as storage writes are very expensive. Even if storage were optimised by using a Merkle tree, the cost of uploading a separate committed action hash per user would be significant.

By default, the preimage used to authorise the metatransaction becomes the next image. That halves the number of images that need to be uploaded.

5. Preferred batchers

Each client in MultiCall has a particular "preferred batcher" user that may batch their metatransactions. In the earlier version of MultiCall [4], anyone could run anyone's metatransactions. On the face of it, that seems a desirable property: users would not need to spend gas choosing a batcher, and there would not be a need for a separate class of users that might attempt to censor transactions or misbehave. However, replay protection in combination with free execution of metatransactions enables an economic attack, as explained below.

5.1. The attack

MultiCall relies on metatransactions to enable transactions from multiple users to be emulated in one, saving users a significant proportion of the gas cost. However, that is all for naught if malicious parties can invalidate transactions containing metatransactions. And indeed, our earlier implementation of MultiCall [4] suffered from an attack exploiting this issue. The attack is as follows:

- The victim (a transaction batcher) broadcasts an Ethereum call transaction to MultiCall, containing a script performing multiple metatransactions.
- The malicious third party sniffs the transaction out of the mempool. They then create their own call transaction to MultiCall, which runs one of the metatransactions M. To preempt the victim, they set a higher gas price before broadcasting the transaction.
- The malicious party's transaction is run before the victim's. The nonce in the account of M's signatory is incremented, preventing M from being replayed.
- When the victim's transaction is mined, M has already been run, causing the attempt to rerun it to revert the transaction. The victim's transaction will have no effect, except to cost them Ether for gas used.

Since MultiCall's design encourages batchers to emulate a full block's worth of transactions in one, such an attack could be very costly for the victim. Since the batcher cannot progress if their transactions revert, the attack could be performed repeatedly and totally halt batching of metatransactions.

5.2. The solution

The problem arises from multiple parties being able to modify the same account in a way that causes legitimate transactions to fail. One mitigation would be to skip metatransactions with failing verification rather than revert the entire transaction, but that is only a mitigation: gas is still wasted uploading the metatransaction and performing the verification. We opted instead to associate each account with a single batcher user that has privileged access to it. In particular, the preferred batcher has the exclusive right to perform metatransactions on behalf of their client. Actions by other users that would invalidate the metatransactions of the batcher's clients are prevented by MultiCall. That includes clients spending Ether from their accounts (which may render an account unable to pay for a metatransaction) and setting the image to a new value. The feature is optional; clients may also become independent, in which case they can access their account directly but cannot benefit from metatransactions until they choose another batcher.

In MultiCall, the association of clients to batchers is achieved by storing a preferred batcher ID in the client's account. Clients can choose their batcher using the setBatcherTo(id) instruction. Batchers instead store their own ID; new IDs are allocated when clients register as batchers using the becomeBatcher() instruction. When running metatransactions, a check is performed to ensure that the caller is a batcher and that they are the preferred batcher of the clients on whose behalf they are running metatransactions.

5.3. Design considerations

On-chain space is limited and very expensive. Even if one uses a Merkle tree to store the state off-chain, it must be re-uploaded when used, which is costly. As such, it is desirable to minimise the state used by smart contracts.

The preferred batcher feature requires an additional state to be added to client accounts: the identity of their preferred batcher. That corresponds to a 160-bit Ethereum address, but that would be costly to store directly—for one thing, it would not fit into a 32-byte storage slot alongside the client's 160-bit image field. Instead, a 23-bit shorthand is used, with new batcher IDs allocated when users register as batchers. As an aside, we propose shorthands as a generally useful technique for programming smart contracts, as they can be much smaller than Ethereum's 20-byte incompressible addresses. Direct use of addresses is in fact extravagant, as they cost several USD cents to upload at typical gas prices [2].

A practical and decentralised crypto protocol must consider the possibility of misbehaviour by all parties. In the case of preferred batchers, that means refusing to run a client's metatransactions. To prevent the client's funds from being locked forever if their preferred batcher misbehaves, the setBatcherTo instruction allows them to leave their current batcher without the batcher's approval. However, immediately ending the client-batcher relationship could be used to attack the batcher by invalidating their attempt to run a metatransaction by the client. Consequently, there must be a warning period after the instruction is run before it takes effect. That is implemented using a field in the client's account which records when the preferred batcher relationship will end if nonzero. When it does, the client can access their account independently but must choose another batcher if they wish to use metatransactions instead (which is more efficient).

6. Implementation

The implementation of the interpreter consists of five main components[6]:

- A volatile state;
- An account array mapping from user addresses to account structs, containing all information relevant to a user;
- A jump table of MultiCall instructions;
- interpreter initialization code which sets the volatile state on entry, and
- an instruction set which defines the available instructions.

We discuss each of these below. Each of the features has been implemented using our EVM code-generating DSL, the C monad. To

[6] The source code can be made available via the PC chairs.

provide context to low-level details of the implementation, we first provide a description of it.

6.1. The C monad

MultiCall is defined in Haskell using a datatype which can be interpreted to generate EVM code. The datatype, called C (for code generator; it has no relation to the C language) encodes actions which modify an environment tracking the evolving state of the program. For example, one can output a byte of EVM code using the operator byte. The datatype is a monad, meaning it supports a pair of operators (return and (≫=), known as bind), which allow one to return values from the action and bind actions together. The result of binding the C action (a :: C a) to the function (f :: a –> C b) is a compound action (a ≫= f) :: C b. When interpreted, it first performs the state updates of a, passes its return value (let us call it *x*) to *f*, and then performs *f x*, returning its return value. Haskell provides a syntactic sugar for monadic operations called do notation, which allows one to write such actions in an imperative style. For example, the following C monad expression

```
do old <- newVar (sload 0)
   new <- newVar (calldataload 0)
   sstore 0 (dupVar new)
   returnExpr (dupVar old)
```

generates EVM code for a smart contract which loads a value from persistent storage, saves it to a stack variable, loads the first word of calldata and overwrites the old value in storage, and finally returns the old value to the caller.

The functions newVar, sload et cetera are defined in terms of basic operations such as byte. This allows very low-level control of EVM code generated, in contrast to Solidity where the inefficient ABI calling convention (with its single method per EVM call and loose packing of arguments) is built in. In addition, the compositional approach (where more advanced constructs are defined as functions in terms of more basic ones) allows one to program at a higher level using reusable combinators. An example of this is whilenot:

```
whilenot :: C() -> C() -> C()
whilenot cond body = do
  loop <- jumpdest
  exit <- label
  jumpi cond exit
  so <- getSO
  body
  so' <- getSO
  if so /= so' then
   error "Invalid␣while␣body"
   else void
  jump loop
  place_jumpdest exit
```

The whilenot combinator generates EVM code which repeatedly executes body while cond is false. Such combinators allow one to program in a structured style. Once a combinator is defined, it can be freely reused; for example, one could write

```
whilenot (iszero (dupVar n)) (do
   accum *=: dupVar n
   n -=: 1)
```

to define a factorial function (given two stack variables *n* and accum).

Combinators which implement code generation patterns turned out to be very useful for generating efficient low-level code, in particular when attempting to implement the pattern manually would be tedious and error-prone. For example, MultiCall's jump table, instruction argument parsing and dispatch code was automatically generated using C combinators.

The C DSL is a so-called state monad, which tracks the code as its being generated; C actions can read and update the state. This is the definition of its state type:

```
data Env = Env {label_ctr :: Int,

                text_offset :: Int,
                syms :: Syms,
                future_syms :: Syms,
                prog_text :: Program,
                stackOffset :: Integer
                }
         deriving (Eq,Ord,Read,Show)
type Program = [Integer]
type Syms = [(Label,Integer)]
data Label = LNamed String | LAnon Int
  deriving (Eq,Ord,Read,Show)
```

Expressions of type (C a) modify a state of type Env and return a value of type a when they are interpreted using the function interp of type Env –> C a –> (a, Env). The function interpret :: C –> (a, Env) passes in an initial Env state and uses a lazy functional programming technique called circular programming ("tying the knot" [11]) to let the monad observe the value of labels before it defines them. The field label_ctr is used to track the next anonymous label index; it is incremented for each anonymous label allocated. The text_offset field tracks the number of bytes of program text generated, and is used to place labels. The field syms is prepended to during interpretation to add new label-integer mappings; a lazy value which will eventually become the final value of syms is passed into the initial environment's future_syms field by interpret. The field prog_text is the accumulated program text, stored in reverse so that bytes can be appended to the program text in constant time using the cons operator. The stackOffset field tracks the stack effect of EVM instructions, and is used to implement stack variables.

The primitive API of the C monad is:

```
label :: C Label --Allocates a new anonymous label
(=:) :: Label -> Integer -> C () --Defines a label
byte :: Integer -> C () --Append byte to
                        --program text
offset :: C Int --Get current program text offset
SetTextOffset :: Int -> C ()
lookupLabel :: Label -> C Integer --Lookup val of
                                  --label
getSO :: C Integer --Gets the stack offset
incSO :: Integer -> C () --Increments stack offset
```

alongside the monad interface of bind and return. From these, more complex C expressions can be defined. For example,

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

places a label at the current text offset. EVM instruction generation is defined by executing the code generation for their argument expressions (which push values to the stack), outputting the instruction's opcode using byte, and adjusting the stack offset using incSO. A combinator has been defined for each EVM instruction. EVM generation code for arithmetic operations and numeric constants is given syntactic sugar using Haskell's Num typeclass.

The power of DSLs lies in their extensibility; from the base operators shown above, more complex code patterns can be defined as functional combinators, allowing easy reuse of custom patterns. That provides the programmer with the ergonomics of a high-level language, while retaining the option of low-level control of code generated. For instance, we have created combinators for structured programming, stack and global variables, structs and arrays.

6.2. Volatile state

When called, MultiCall uses 15 one-word global variables as its volatile state; we describe 4 key ones now. The remainder will be introduced

as needed, when describing the features which they are used to implement.

- The program counter pc is used to track the next MultiCall instruction to be executed from the calldata; pc is stored on the stack, the rest in memory.
- Because MultiCall may batch transactions from many users in a single call, a mutable variable is required to track on whose behalf it is executing: signatory tracks the Ethereum public key of that user.
- The variable balance caches the credit of the current user, deducted for instructions which spend Ether such as calls, creates and deposits.
- The variable stashedBalance is used to remember the balance of the caller when they execute a metatransaction on behalf of a client.

The program counter is accessed using the SVar datatype:

```
data SVar = SVar{originalOffset::Integer,
                 varOffset::Integer}
             deriving (Eq,Ord,Show,Read)
--API:
declareVars :: Integer -> C [Svar]
dupVar :: SVar -> C ()
swapVar :: SVar -> C ()
newVar :: C () -> SVar
```

In each instruction, the program counter is declared to be already on the stack using declareVars. Values of type SVar contain enough information to locate a stack variable, even as the code pushes and pops values; the functions dupVar and swapVar perform an EVM DUP and SWAP of it respectively. The function newVar takes an expression and allocates a new stack variable containing it. This feature is made possible by the C monad's static stack offset tracking.

The in-memory global variables are implemented using the GVar datatype:

```
data GVar = Memory Integer | Storage Integer
    deriving (Eq,Ord,Read,Show)
--API, works for both memory and storage globals
load :: GVar -> C ()
store :: GVar -> C () -> C ()
```

An example use in the Haskell definition of MultiCall is the definition balance = Memory 32, which defines the in-memory location of the balance global. The balance is then modified using the load and store functions, and using combinators which use them (such as (+ =), which has the obvious meaning).

6.3. Account array

Each user (identified by an Ethereum address) has an entry in the array accounts in persistent storage, which contains Account structs. Since struct access code is error-prone to write using manual bit twiddling, this required the DSL be extended with typed expressions:

```
newtype Expr t = Expr {untypeExpr :: C ()}
newtype TVar t = TVar {untypeTVar :: SVar}
newtype TGVar t = TGVar {untypeTGVar :: GVar}
```

The type Expr wraps an untyped C monad code block, tagging it with the type of value it pushes. The types TVar and TGVar are typed equivalents of stack and global variables respectively. The parameter t symbolizes an EVM type, not a typical Haskell type. The type Uint (n :: Nat) is commonly used in our code. We also defined a type of structs (parameterized by the types of its contents), struct fields and various accessor combinators. A typical use of structs found in the code is: (acc, accTime) .= 0. The (.=) combinator takes a pair of a typed stack variable containing a struct and a named field of that struct type, as well as an untyped expression (in this case zero) and assigns the expression to the given field of the struct.

Readers familiar with Solidity may wonder what the difference is between an account array and a mapping. Arrays are contiguous areas of storage slots, whereas Solidity mappings are hash maps. Arrays are cheaper to access, but in contrast to mappings support only keys smaller than an EVM word. Since addresses fit within a word, an array is the more efficient choice. The array accounts is a sequence of 2^{160} storage slots; since only nonzero slots are stored by Ethereum nodes, such vast but mostly empty data structures are viable.

To implement it in the DSL, we defined a type of static arrays with typed content:

```
data STA ix a = STA {staOffset :: Integer}
indexSTA :: STA ix a -> Expr ix -> Expr a
writeSTA :: STA ix a -> Expr ix -> Expr a -> C ()
```

And then we defined the accounts array as

```
accounts = STA 2 :: STA Address Account
```

That means the array starts at storage index 2, permits the entire range of addresses as indices, and contains values of type Account.

Limiting accounts to a single word is vital for performance, since persistent storage loads are very expensive and only load a single one-word storage slot at a time. In the previous version of MultiCall [4], account structs were simple: just a 48-bit balance and a 16-bit nonce. The implementation of HBAuth and the preferred batcher features has made them more complex.

```
//Account struct pseudocode
struct Account {uint32 balance,
                uint16 nonce,
                uint24 time,
                uint24 batcher,
                uint160 image}

--In Haskell code
type Account = Tup (Eth,Nonce,ShortTime,
                        Batcher,Hash)
type ShortTime = Uint 3
type Batcher = Uint 3
type Hash = Uint 20
type Eth = Uint 4
type Nonce = Uint 2
```

In particular, the image field rendered design of the struct nontrivial due to its size: EVM storage slots consist of 32 bytes, of which 20 are used for the image. Fitting the other fields in the remaining 12 bytes was a challenge. It was achieved in two ways: by overloading the meaning of fields based on whether the user is a client or batcher, and shaving a byte each off the balance, time, and batcher fields.

Shaving bytes off fields has a cost in the accuracy and range of representable values; an alternative approach would be to store some or all of the fields in a hash. That would increase the cost of accessing the account by at least 320 gas (the cost of uploading 20 bytes) and complicate the off-chain logic required, but would be an interesting avenue to explore. Storing all accounts in a single Merkle tree would also be of interest for future work, but is beyond the scope of this paper.

6.3.1. Balance

The balance variable records the user's Ether balance in terms of the monetary unit, a number of wei (the minimum unit of Ether). The monetary unit determines the precision of the Ether values accounts can store and payment instructions can transfer, as well as the maximum representable balance size. Originally having a precision of 1 gwei (a billionth of an Ether) and maximum value of 281474 Ether, balances now have a precision of 10000 gwei and maximum value of only 42949 Ether. That corresponds to a precision of 3 USD cents and a maximum of 128 million dollars at current prices. It is possible that the precision should be improved at the expense of the maximum value; that is trivial, just a question of adjusting a constant.

How much accuracy is required is an open question, but we believe

that the significant cost of paying calls (on the order of dollars even if using MultiCall [2]) renders small on-chain payments inefficient. MultiCall can provide significant savings, but there is no question of it providing micropayment functionality—for micropayments layer 2 solutions are more appropriate, with on-chain payments best used for settlement. As such, a large granularity of payment and balance values imposes a relatively small proportional granularity to efficient payments (i.e., those with a low gas cost relative to value transferred).

6.3.2. Nonce

The nonce field is used for replay protection in metatransactions. It remains unchanged from the original MultiCall. While it currently does not permit overflow (preventing any account from ever sending more than 65535 metatransactions), modifying it to overflow would not pose a security problem since the signatures used for HBAuth have been enhanced with an expiry time chosen by the user after which the signature will not be accepted. By choosing a reasonable expiry time (such as 24 h after the signature is created), users can ensure that old signatures for low nonce values do not remain valid after nonce overflow.

6.3.3. Time

The variable time is used to record the end of the warning period W_1 (when the batcher can reset their CA) or W_2 (when the client ends the client-batcher relationship) for batchers and clients, respectively. The EVM provides a TIMESTAMP instruction which allows smart contracts to query the “current time” (i.e., the timestamp of the block in which the transaction calling them is included). The timestamp is the 32-bit UNIX epoch time in seconds; since time is only 24 bits, it records times with a granularity of 256 s. Precision could be improved by taking into account that only times until January 19, 2038 (when the UNIX time overflows) need be represented, but that would have an additional gas overhead. By transferring a byte from time to balance, one could increase the latter’s accuracy to a hundredth of a cent at the cost of reducing time’s accuracy to about 18 h.

6.3.4. Batcher

The batcher field contains a 23-bit batcher ID in its most significant bits and a bit indicating whether the user is a client (0) or batcher (1) respectively. If the user is a client, the batcher ID refers to their preferred batcher; if the user is a batcher, it refers to the user’s own batcher ID. Eight million IDs is expected to be sufficient, as most users do not need to be batchers. However, it renders the contract more vulnerable to a denial of service attack where a malicious party deliberately exhausts the space. Adding a small fee paid to the contract’s creator to register as a batcher could serve as a mitigation; if an attacker paid a dollar per ID to exhaust the space, we would be rather pleased.

6.3.5. Image

If the user is a batcher, the image field contains their committed action (CA). If the user is a client, it contains the image which the user reveals the preimage of when using HBAuth to send a hash-based metatransaction.

6.3.6. Design considerations

A consequence of the limited space available is that a desirable field was dropped: the desiredBatcher field would have allowed clients to specify which batcher they wanted to switch to when they announce they are leaving their current batcher without that batcher’s approval. That way, they could transition directly from one batcher to another; instead, they must become independent and then choose the next batcher via a direct call, which is more expensive.

6.4. Jump table

MultiCall’s jump table consists of an array of EVM code entries, each of which is a JUMPDEST instruction marking a valid jump destination followed by a jump to a constant address (the address of the instruction code). The interpreter’s one-byte opcodes are used as byte offsets into the table; the dispatch code simply jumps into the table using the opcode as a byte offset. The jump table is therefore only 256 bytes long, and because each entry is 5 bytes (one byte for JUMPDEST, 4 for a constant jump), it can fit at most 52 instructions. Thankfully that is sufficient for MultiCall’s functionality, but future interpreters may require a different dispatch scheme. MultiCall’s dispatching mechanism is efficient enough for our purposes, costing only 41 gas compared to 3 gas for an add or push EVM instruction. Its cost is negligible compared to the cost of executing transaction-emulating instructions, as shown in Section 7.

Manually constructing the jump table from the set of instruction definitions would be tedious and error-prone; instead, we defined a combinator jumptable_mc:

```
jumptable_mc :: [C ()] -> C ()
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]
```

The combinator places each instruction from the list of C () code blocks it receives, tracking their offset in the code. It then constructs a jump table to those offsets, and binds a named label “JUMPTABLE_MC” to the start offset of the table. The code of MultiCall as a whole is defined as.

```
do init_hbamc
    defProxy --Defines the code of proxies
    jumptable_mc [setBatcherTo,becomeBatcher, ...]
```

6.5. Initialization code

When MultiCall is invoked, MultiCall sets the in-memory variables signatory (the address on whose behalf MultiCall is executing) to the caller, balance to the number of monetary units (10,000 gwei) deposited by the caller, and the program counter to 2. It also parses the 2 first bytes off the calldata and sets the hint pointer to that value. The hint is described in more detail in Section 6.7. MultiCall then dispatches, entering the first instruction.

```
init_hbamc = do
    store balance $ callvalue / monetaryUnit
    store signatory caller
    --Parse 2 bytes off calldata to get
    --hint pointer
    store hint (calldataload 0 ‘shr‘ 240)
    store desiredBatcher (0 - 1)
    --Push PC = 2 to top of stack
    2
dispatch_mc
```

6.6. Instruction set

For efficiency, there is no separate interpreter loop; each instruction dispatches to the next using the C expression dispatch_mc, which functions as a macro.

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

The dispatch code duplicates the program counter, which is on the top of the stack, uses it to load a word of calldata, selects the first byte of it and uses that as an index into the jump table.

Each MultiCall instruction consists of a contiguous block of EVM bytecode. MultiCall instructions modify the state of the interpreter and perform some side effects (such as internal state changes or EVM calls) until an instruction throws an exception or the endCall instruction exits the interpreter. We describe the implementation of a subset of key MultiCall instructions below.

6.6.1. *Making payments*

Paying instructions such as calls, creates and deposits perform a side effect which may cost Ether: performing an EVM call, creating a contract, and crediting another user's account, respectively. Such instructions deduct the payment from the volatile balance variable, rather than directly from the account of the signatory on whose behalf MultiCall is executing. That saves gas since persistent storage writes are significantly more expensive than memory writes.

6.6.2. *Stopping execution*

There are two points at which execution of instructions on behalf of the current principal (identified by the signatory variable) is stopped and the volatile balance is settled against their account: in endMetatxn and endCall. In both endMetatxn and endCall, checks are performed that are deferred until the principal's account is loaded into memory. That is achieved by storing the details of the checks to perform in global variables such as nonce. Checks and modifications of the user's account (such as modifying the balance) are deferred in order to reduce the storage accesses required to one read and write per user.

The endMetatxn instruction ends a metatransaction and performs some checks relevant to authorisation. If the nonce global variable is nonzero, then a signature was used, and the nonce is compared to the nonce field in the client's account. If it is zero, then a 20-byte preimage is parsed off the hint pointer from calldata, hashed and compared to the image field of the client's account. The endCall instruction checks that MultiCall is not executing a committed action or metatransaction and then checks that the caller was authorised to batch on behalf of any users whose metatransactions were run. In both cases, the volatile balance variable is settled against the balance field stored in the user's account; if there is an overflow or underflow, the transaction fails and has no effect.

The endCall instruction is the only means of ending a call to MultiCall; if the script passed in the calldata does not contain an endCall, then the interpreter will loop until it runs out of gas and throws an exception, reverting any desirable side effects.

6.6.3. *Proxies*

Each user controls a proxy contract, whose address can be computed from the user's address. Proxies are allocated with the createProxy instruction, which creates a proxy using the EVM instruction CREATE2. The instruction behaves like CREATE, except that the address of the created contract is deterministically computed from the creator address (in this case, the address of MultiCall), the initialization code and a salt. The salt used is the address of the signatory. That eliminates the need to store the proxy's address in the user's account, as the address can be recomputed when needed by the proxy_dot_create and proxy_dot_call_address MultiCall instructions.

All proxies created by the same instance of MultiCall have the same bytecode, which checks whether the caller is its creator (MultiCall) and the calldata are ended by the magic 32-bit number indicating a proxy call or create command. If that is the case, the proxy performs the commanded call or create. The trivial way for a proxy to store its creator's address would be to place it in persistent storage and read it on each call, but that would be inefficient as storage reads are expensive. Instead, during contract creation, the creator's address (which can then be compared to the CALLER) is written into a push instruction in the in-memory bytestring, which is returned as the final code of the proxy. The end result is that fetching MultiCall's address in order to compare the caller's to it costs only 3 gas.

6.6.4. *Metatransactions*

MultiCall metatransactions consist of a startMetatxn instruction, containing a number of MultiCall instructions terminated by an endMetatxn instruction. As stated in Section 4, there are two ways of authorising a metatransaction: signatures and preimages. Signature verification is achieved by calling the primitive contract ECRECOVER, which recovers the public key to the signatory given a hash and an Ethereum signature of the hash. The metatransaction, the tip, a nonce and deadline field are included in the hash. The nonce and a deadline chosen by the user are uploaded alongside the signature and are used for replay protection and to ensure that signatures eventually expire. Preimage verification is achieved by comparing the submitted preimage's hash to the image stored in the user's account. Metatransactions execute on behalf of their creator if authorisation succeeds. What executing "on behalf of" a user means that the signatory variable is set to that user's public key. The value of signatory is restored to the caller upon the next stop instruction. The balance of the caller is also saved to the stashedBalance variable, and the signatory executes with a new balance. The tip is deducted from the new balance and credited to the stashed balance. When signed execution stops (at the next endMetatxn instruction), then the signatory's balance is settled, the stashed balance is restored, and MultiCall reverts to executing on behalf of the caller. Since there is only one stashedBalance variable rather than a stack of balances and addresses, nested metatransactions are disallowed. Note that if the metatransaction is terminated with an endMetatxn prematurely, it is the caller that pays for subsequent instructions. If it is not terminated at all, the batcher is free to append their own instructions when running the signed script, enabling the theft of any Ether in the account and any ERC-20 tokens controlled by the account's proxy contract.

An attractive feature of MultiCall metatransactions is that they allow the authorisation cost to be shared across multiple authorised actions.

6.6.5. *Batcher-specific instructions*

The becomeBatcher instruction is the means by which new batcher accounts are created; all users start as clients. When a user runs becomeBatcher, the wantToBecomeBatcher volatile variable is set, which triggers a deferred conversion to batcher when their account is settled. That amounts to allocating a new batcher ID from the persistent counter variable batcherIDCounter, setting the most significant 23 bits of the user account's batcher field to that value, and setting the least significant bit to 1.

The setCA(h) instruction sets the user's image field to the given 160-bit hash h if they are a batcher. The write is deferred by storing the hash in the desiredCA volatile variable. It also sets the time field to the current time plus a warning period; the committed action may not be changed until then. If there is already a committed action in the variable and the time field is greater than the current time, setCA fails.

The startCA(script) instruction runs the given script if the hash of the script matches the committed action in the batcher's image field. The script immediate argument is merely a 16-bit length field and a bytestring of MultiCall instructions of that length. If the script matches the committed action, then startCA sets the inCA flag and sets the program counter to the start of the bytestring. Metatransactions require inCA is set to run.

6.6.6. *Client-specific instructions*

The setImage(h) instruction performs a deferred write to the client's image field. It is performed after the preimage comparison used by HBAuth; otherwise, an attacker could authorise any metatransaction.

The setBatcherTo(id) instruction performs a deferred write to the most significant 23 bits of the client's batcher field, assuming it is run in a metatransaction. If run directly in a call, the time field in the user's account is set to the end of the warning period. When the current time exceeds the time field, the client becomes independent. The next time

their account is accessed, its batcher ID will then be written to 1, indicating no batcher. The client will then be able to spend from their account directly but not use metatransactions.

6.7. Hinting

In MultiCall, the proofs of approval passed to runMetatxn instructions are not in fact in their immediate arguments. Instead, there is a separate pointer global variable hint from which they are parsed. That is useful, since the runMetatxn instructions must be committed by the batcher before the preimage they use is revealed. The preimage can therefore not be an immediate argument included in the commitment. One can think of the script as not being complete by itself but rather being parameterisable or dependent on help from the batcher. The batcher serves as an untrusted "kernel", providing untrusted data in response to user requests, allowing the user to only verify rather than compute hinted data. The batcher in turn must predict the hints required in the course of transaction execution and upload them as part of the transaction calldata. We call this technique hinting and consider it of interest for future study.

7. Performance evaluation

Note: gas cost evaluation was performed using the popular Ethereum tools Truffle [12] and Ganache [13], which allow smart contract testing in a private blockchain but do not yet support the rule changes in the recent Berlin hardfork of Ethereum. One change relevant to MultiCall was made in the Berlin fork: EIP-2929 [14], which increases the cost of accessing contracts and storage indices for the first time in a transaction ("cold load") but makes them much cheaper to access subsequent times ("hot load"). We expect this to enhance the cost savings provided by MultiCall and batchers in general since batching many transactions together should allow more of their calls and storage accesses to pay the lower hot load cost.

We will showcase the cost savings provided by MultiCall with micro-benchmarks as well as by revisiting the example Token from Section 1, where MultiCall saves 59.6% of the gas required for token-transfer costs. We also compare the performance of MultiCall with a preexisting batcher, MultiSend. Finally, we compare MultiCall to iBatch, a batcher described in the academic literature [5].

7.1. Micro-benchmarks

To evaluate the gas cost savings provided by MultiCall, we ran a number of MultiCall instructions in sequence and measured their marginal cost. We uploaded MultiCall to a private test chain run using Ganache version 7.2.0 [13] and tested it using truffle version 5.5.19 [12], a development framework for Ethereum that can provide an interactive JavaScript console to the private chain. The savings provided by using transaction-emulating instructions relative to transactions as a proportion of the fixed transaction are shown in Table 2. To clarify, if an instruction which emulates a call transaction has a cost of X, its savings are reported as $(21000-X)/21000$, rounded to the nearest tenth of a percentage point. To calculate the savings of instructions that emulate create transactions, we first deduct the 32000 gas cost that is paid either way and then perform the above calculation.

Table 2
MultiCall gas costs and savings *vs.* unbatched transactions.

MultiCall action	Gas cost	Savings
Ether-paying call	8060	61.6%
Non-paying call	1348	93.6%
Ether-paying proxy call	8968	57.3%
Non-paying proxy call	2256	89.3%
Create	32206	99.0%
Proxy create	33233	94.1%
Deposit	6405	69.5%

The calldata used for calls and creates were empty; the contracts called were dummies which returned immediately upon being called, in order to eliminate any gas cost confounding from contract execution. There is a negligible overhead of approximately 3 gas per word for copying calldata into memory when batching, rather than sending transactions directly. The fact that MultiCall creates instructions cost more than the fixed transaction cost of 21000 gas and yet still provides savings may be surprising; note that the 32000 contract creation cost is paid by create transactions as well. Create transactions have a minimum gas cost of 53000, which includes both the 32000 gas cost of creation and the 21000 fixed transaction cost. The proportional cost saving of batching is computed as a fraction of the 21000 gas fixed transaction cost saved, not the 53000 gas cost, which includes the create. We conclude that batching via MultiCall provides significant cost savings.

7.2. Metatransactions

The version of MultiCall described in this paper provides two forms of metatransaction: signature and hash-based. Both use the runMetatxn instruction, which takes a script of MultiCall instructions and executes it on the client's behalf. The difference lies in whether the batcher provides a signature or a preimage as proof of authorisation. The instruction can be used to allow multiple clients to share a single call transaction for their batching. However, runMetatxn has some overhead—modifying the user's nonce and image requires a storage write, and both signature verification or hashing cost some gas. If runMetatxn cost more than 21000 gas, it would of course not be of any use—then users might as well send transactions separately.

To evaluate the runMetatxn instruction, it was profiled by sending varying numbers of metatransactions with empty scripts (containing only an endMetatxn instruction). Profiling is somewhat more complicated than for other instructions. Different clients must be used for each metatransaction in the call to avoid amortisation of the write to the client's account; repeated writes to the same storage slot cost less in the gas cost model. Also, HBAuth requires some additional off-chain logic to use which the normal instructions do not. For that reason, runMetatxn was profiled separately from the other instructions.

In short, the gas cost of each runMetatxn varies slightly due to variation in the number of zero bytes in the signature (which cost 12 gas less to upload in transaction calldata than nonzero bytes), but signature-based metatransactions appear not to cost on average 12472 gas. Hash-based metatransactions appear to cost on average 7489 gas, for a saving relative to signature-based of 4983 gas. The end user may assume they save approximately 13500 gas using a hash-based metatransaction to share a call compared to calling MultiCall themselves. A noteworthy implication is that it is cheaper to send a MultiCall metatransaction that contains only a single Ether transfer than to use an unbatched transaction.

Signature-based metatransactions now cost approximately 500 gas more than they did in the earlier version of MultiCall [4]. That is because this version has not been optimised for the signature-based case; they are the "failure case" of hash-based transactions, intended to be used only when the peer-to-peer protocol fails or when the user must submit their first hash. However, hash-based metatransactions are significantly cheaper regardless.

Hash-based metatransactions require the batcher to commit the metatransactions to be run in advance. That requires a storage write, which costs around 5000 gas; perhaps that is a threat to their usefulness? Thankfully, the batcher can run a set of metatransactions, commit a new set and add client tips to their balance in the same storage write. The fixed cost of the batcher making a call transaction which runs a set of metatransactions and committing a new set is below 27900. A user sending a hash-based metatransaction performing only a single paying call saves over 5000 gas; it would take only 6 such metatransactions in a call to overcome MultiCall's fixed cost. Since blocks may contain hundreds of transactions and individual users may make many payments in a

```
xfer = to => api.iset2.proxy_dot_call_address(
 20480,
 T.address,
 0,
 util.abi_method("xfer(address,uint256)")+
  util.abi_addr(recip(1))+api.abi_uint(1));
web3.eth.sendTransaction({
 from:accs[0],
 to:MC,
 data:api.withHints(xfer(recipient1) + ... +
  xfer(recipientN),
 gas:1000000})
```

Fig. 9. Batching contract calls using MultiCall.

single metatransaction, MultiCall's fixed cost is not expected to be a significant problem.

7.3. Token transfers

Consider once more the example of Alice making one token payment each to *N* recipients, using the method .xfer(address, uint) of the Solidity contract Token—recall Section 1. The gas cost of doing so in one transaction via MultiCall compared to sending *N* transactions individually is shown in Fig. 10 for *N* ranging from 1 to 10. We can see in the figure that the savings are significant, i.e., up to 59.6% for 10 payments, and the proportional savings improve as the fixed overhead of MultiCall is amortised.

On the one hand, we launch *N* transfers with truffle by simply making *N* standalone JavaScript method calls T .xfer(to, amt). Such calls are then translated into call transactions to the address T.address, with calldata containing the 32-bit method identifier for xfer followed by the 160-bit address argument to (left-padded to 32 bytes) and 256-bit token amount amt. When making *N* unbatched Solidity method calls, each transaction executes independently and costs the same amount of gas: 33817. The total cost is therefore $N \cdot 33817$.

In contrast, when making *N* payments using MultiCall, only a single call transaction to MultiCall is sent; the calldata consists of the concatenation of *N* proxy_dot_call_address instructions. Each proxy call instruction calls the user's proxy, which in turn calls the Token contract instance T.address, with the same calldata as used in a standalone .xfer transaction. The JavaScript code for making the call to MultiCall from truffle is shown in Fig. 9. The cost of making a single payment via MultiCall is 38238. After that, the marginal cost of making an additional payment is always the same: 10912 gas. That is, a 67.7% saving compared to sending token payment transactions individually. The shared fixed cost for the sequence of payments is 27326 gas, corresponding approximately to the fixed transaction cost and the cost of a storage write. Since batched Token payments are implemented using a proxy call which does not make an Ether payment, one would expect from the micro-benchmarking that approximately 2256 of the 10912 gas cost is the overhead of MultiCall, and the remaining 8656 is the .xfer method's execution cost. That is surprisingly low, since the method performs two storage writes, which typically cost 5800 gas each. However, repeated writes to the same storage slot are cheaper in the cost model. We believe that since the proxy's balance is deducted repeatedly, the marginal cost of updating it is reduced. The ability to amortise the cost of storage writes made during contract execution makes batching transactions even more attractive than the micro-benchmarks would indicate.

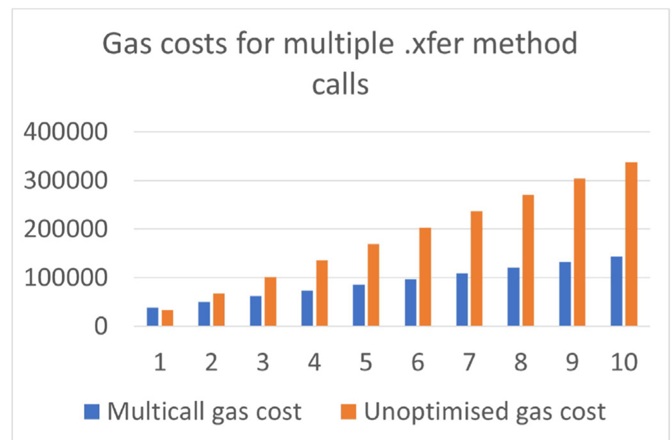


Fig. 10. The gas cost of 1–10.xfer method calls to the contract Token shown in Section 1, compared to the cost of unbatched EVM method calls.

7.4. MultiCall vs. MultiSend

In this section, we will show that MultiCall is more performant than MultiSend[7], a batcher implemented in Solidity assembly with the purpose of reducing gas costs. Before going into the details of our comparison, we summarise the main result.

For the only comparable feature (call batching), MultiSend costs 200 more gas per batched call. Since the upload cost of each MultiCall call instruction is about 200 gas lower than its MultiSend equivalent, we speculate that the difference lies in MultiCall's highly optimised instruction argument packing and parsing, generated by a DSL combinator. Specifically, MultiSend stores each immediate argument in a separate word, while MultiCall packs them densely and then uses shifting to efficiently unpack them into separate stack slots. Such shifting is tedious and error-prone to perform manually, so it is ideal for automatic generation. We do so using a combinator unpackLeft :: [ArgType] –> C [SVar]. The ArgType values indicate the types (and thus byte sizes) of values to unpack; the monad returns a list of stack variables containing the unpacked values. Access to combinators that can be reused to generate efficient bytecode is an advantage of the DSL over Solidity assembly.

7.4.1. Profiling details

The cost of paying and non-paying MultiSend calls was evaluated the same way as MultiCall calls were, using calls with the same calldata (the empty bytestring) to the same contract (a dummy which stops immediately). Batches of 1–20 calls of each type were made, and the marginal cost of a call inferred by calculating the gas cost delta. Marginal gas costs varied in a range of 1464–1610 for non-paying calls and 8176 to 8311 for paying calls, without a clear downwards trend. On average, non-paying calls cost 1557 gas and paying calls 8268. Both are approximately 200 gas more expensive than their MultiCall equivalent, approximately 1% of the fixed transaction cost (see Table 3). As MultiSend is a popular batcher written in assembly with the purpose of reducing gas costs compared to plain Solidity, that is an encouraging result.

7.4.2. Comparison discussion

It bears noting that comparing the cost of calls does not factor in the additional features required to use MultiSend that are built into MultiCall. Because MultiSend does not support metatransactions or have multiple account records for different users, each user is expected to separately delegate to it from a different contract. Payments between users of MultiSend also require at least one call, while multiple users

[7] https://github.com/gnosis/safe-contracts/blob/8443cfaa410bfb197cc708b1c5e06ffa0c49c217/contracts/libraries/MultiSend.sol.

could make deposits to other users without performing a call in MultiCall. Baking many features into the same contract reduces the overhead of context switching; interpreters like MultiCall are a promising avenue for doing so.

In summary, the results of the evaluation are promising and suggest that greater adoption of batching would be advisable. MultiCall's interpreter design does not appear to impose unacceptable overhead and could be reused in future smart contracts.

7.5. MultiCall vs. iBatch

We also evaluated iBatch, a batcher smart contract described in the academic literature [5][8]. Like MultiSend, iBatch is written in Solidity and supports batching of calls but not creates; however, it also uses signatures to allow multiple principals to share a transaction. By using a shared nonce, iBatch cleverly avoids the need for each user to update a separate one. As with MultiCall, the off-chain batching server collects calls from multiple users. Once enough have been collected, the server concatenates them and sends them to each user, who then signs the combined calls and shared nonce. The calls are then submitted alongside the nonce and signatures; the script is hashed and for each call the user's signature is checked against it when making a call.

The authors claim this allows iBatch to offer replay protection without any in-contract state accesses; sadly, an implementation error means that is not the case. The submitted nonce is not checked in the code, meaning batched calls can be trivially replayed by a malicious party making another call with the same calldata. There is also another error: when signing the script, the arguments to the batched calls are not included in the hash. That means they're malleable, which is unacceptable from a security perspective: the recipient of an ERC transfer method could be changed, for example. Fortunately, the two errors would be simple to solve: to prevent a replay attack, one could keep a nonce variable in storage for a fixed cost of around 5000 gas per invocation of iBatch, and call argument malleability could be prevented by also hashing the arguments for a slight increase in gas cost. All in all, the two errors mean that the comparison to MultiCall is biased slightly in favour of iBatch.

Instead of providing a proxy contract for each user, iBatch passes their address as the first argument of any call.[9] Calls made from iBatch may take 1-4 additional whole-word arguments; to make the profiling of iBatch as similar as possible to that of MultiCall and MultiSend, we sent 1–20 calls from it with a zero-valued one-word argument to a dummy contract that stops immediately.

The difference in gas costs between batching N and $N+1$ calls is the marginal cost of batching a call; it averages 10200. When averaging the marginal costs of 21–100 calls, we found a slightly higher average: 10234. That is not surprising, since hashing of the calls is done by copying all of them into memory at once, and the cost of memory allocation in the EVM has a quadratic component. It appears that the overhead of batching a call in iBatch is significantly higher than that for either MultiCall (1348) or MultiSend (1557). Note that iBatch does not support Ether-paying calls, so the cost for non-paying calls is compared.

It should be noted that signature verification is integrated with each batched call in iBatch. That is inefficient if users wish to make several calls, but let us assume they do not. Then, a fair cost comparison for an iBatch call is not to a MultiCall call instruction but to a metatransaction containing only a single call. We profiled single-call metatransactions in MultiCall and found they cost 8803 gas, about 1400 less than calls in iBatch.

Table 3
MultiCall gas cost *vs.* MultiSend.

Action	MultiCall cost	MultiSend cost (average)
Ether-paying call	8060	8268
Non-paying call	1348	1557

We were surprised that the reduction in per-call overhead in iBatch compared to the fixed transaction cost (51%) was lower than the 61% saving as a proportion of the total transaction cost advertised in the iBatch paper. However, the mystery was resolved when we profiled iBatch for calls to a contract which made a storage write. Repeatedly accessing a storage slot within a single transaction is significantly cheaper than accessing it for the first time, so all batchers provide additional savings above elimination of the fixed transaction cost when making calls that perform such accesses.

A significant part of iBatch's per-call overhead is signature verification (costing around 4740 gas for signature upload and the verification computation); by sharing that across multiple calls like MultiCall does the overhead could be reduced.

The origin of the around 3000 gas cost that remains unaccounted for after considering the calldata upload cost and contract cost remains mysterious; we suspect that using Solidity assembly to optimise memory accesses could reduce it significantly. While iBatch is not optimally efficient, we believe the nonce-sharing approach used is a sound and interesting idea.

In fact, it is such a good idea that it is a potential threat to the validity of HBAuth. Hash-based authorisation requires a hash in storage to be updated for each user that uses it; in MultiCall that costs 7489 gas, whereas a signature verification without a state write costs only around 4740 gas. Does that render HBAuth obsolete? We believe the answer is no: while iBatch touts its lack of storage use as an advantage, contracts have state for a reason. The user account in MultiCall contains a nonce and hash for HBAuth but also an Ether balance. That enables users to make Ether payments directly from MultiCall and deposit Ether in other users' accounts without making a call; with iBatch a proxy contract would have to be used. Most of the cost of HBAuth (around 5000 gas) is the update to the user's storage, which can be amortised if the user also spends Ether, requiring their balance be deducted. One might object that with the proliferation of ERC tokens, Ether payments could in the future become infrequent enough that such amortisation would be rare. In our view, that is a valid point, and the answer is to make the in-batcher state richer (for example, by adding ERC balances). Doing more within the batcher itself should in general be cheaper than accessing the external state. One could also use Merkle trees to store user balances, thus requiring only one storage slot access to update the accounts of many users. We believe that HBAuth remains most efficient when there is cause to update the state in the batcher contract (allowing the storage write required by HBAuth to be amortised), but shared-nonce signature verification would be a useful complement when there is not. Integrating the two approaches in a single batcher contract which also uses Merkle trees would be an interesting subject for future work.

8. Security

We provide a simple demonstration of a security feature and an outline of our reasoning used when designing MultiCall to be secure. Rigorous testing and formal verification of MultiCall using a proof assistant would be interesting future work.

8.1. Demonstration of preferred batcher feature

We have written Javacript API code, which runs in the truffle console, meant to automate the process of running the HBAuth protocol when

[8] Retrieved June 21, 2022 from https://github.com/syracuse-fullstacksecurity/iBatch-offchain-sim. iBatch does not compile with the latest the latest Solidity compiler; we had to downgrade the compiler version to 0.4.25 and make two trivial changes to the program, but they do not affect the semantics or results.

[9] The authors propose to reengineer smart contracts to treat iBatch as acting on behalf of the first argument for the purpose of authorization.

sending hash-based metatransactions. It does not implement the full protocol in the sense of allowing a client to communicate with a remote batcher, but rather performs the computations both parties would have performed. Nonetheless, it is enough to test MultiCall. We show interactions in the console (saved in the course of testing) to demonstrate that only the client's chosen batcher (in this case, MultiCall's creator) may batch their metatransactions.

```
truffle(development)> api.account2(MC,acc1)
    { balance: 0,
      nonce: 20,
      time: 0,
      batcher: 0,
      hash: 'ea6c72d005b4ac33434c
             47dbdc30b86d0e160ace' }
```

First, we check the account of the address acc1, a variable which was bound earlier. MultiCall has been created, and its address is bound to MC. The function api.account2 fetches and parses MultiCall account structs. As we can see, 19 metatransactions have already been run for this user, and they have an image set.

```
truffle(development)> api.runMetatxns(MC,acc0,
                                        [[c1,0,""]])
```

Then, we run a function which performs a metatransaction on behalf of acc1. The value c1 is a client object which contains acc1's hashes. The output of this command is verbose (consisting of an Ethereum transaction receipt object) and not shown.

```
truffle(development)> api.account2(MC,acc1)
    { balance: 0,
      nonce: 21,
      time: 0,
      batcher: 0,
      hash: '40528f978901b7b96bdb
             03e89dda968b2384220b' }
```

As we can see, a hash-based metatransaction has run; the nonce has been incremented, and the hash is the preimage of the old hash (using the hash function api.hbaHash). Now, we will simulate the attack from Section 5 and attempt to batch a metatransaction made by acc1 from accs [4], a user we converted to a batcher earlier.

```
truffle(development)> api.runMetatxns(MC,accs[4],
                                        [[c1,0,""]])
```

The output has been truncated (and our own debugging messages elided) to show the relevant part: when unauthorised batchers attempt to run metatransactions, they are prevented from doing so.

8.2. Informal security argument

To be secure, HBAuth in combination with the preferred batcher feature must ensure that batchers can run their committed metatransactions. It must also protect clients from metatransaction forgery. In what follows, we present an informal argument on why our solution prevents the attack presented in Section 5, (protecting batchers from transaction invalidation) and protects clients from theft or indefinite lockup of funds.

8.2.1. HBAuth security

Because the first step of the off-chain protocol is to give the batcher a signature of the client's desired metatransaction, the batcher will always have the requisite cryptographic proof to run metatransactions they commit, regardless of whether the client subsequently refuses to reveal their preimage. The batcher must also ensure that the metatransaction can be run successfully—that the client does not spend more Ether than they have, or use too much gas[10]—before committing it. The client can therefore not attack the batcher by causing them to commit to running invalid metatransactions.

To authorise a metatransaction on behalf of a particular client, one requires a cryptographic proof of authorisation in the form of either the preimage to the hash in their account or a signature from their private key. As long as the client keeps their chain of hashes secret until it is time to reveal one and keeps their secret key secret and does not provide signatures to malicious parties, forging such an authorisation is intractable as long as the underlying cryptographic primitives (SHA3 and Ethereum's signature scheme) remain secure against collision and forgery, respectively.

8.2.2. Preferred batcher security

There are four ways to modify a client account which could invalidate a metatransaction: ending the client-batcher relationship, incrementing the nonce, modifying the image, and decrementing the balance. Of the four, the first can only be done with a delay allowing the batcher to run their committed action first; the other three can only be done within a metatransaction (that is, with the batcher's approval) unless the client has no batcher. Therefore, malicious parties cannot invalidate metatransactions once a batcher has committed them.

At any point in time, each client has at most one user that is their preferred batcher. That is because their account specifies a particular batcher ID that is their preferred batcher, and each batcher has a unique batcher ID. Batcher IDs are unique because they're allocated from a 23-bit counter which may only increment, and which disallows further allocations when it would overflow. The special batcher ID 0 (which indicates the default batcher all accounts start with) can only be held by one account at a time, since MultiCall starts with only the creator holding the special ID, and granting it to another batcher account requires allocating a new non-special ID for the sender. Transferring the special ID cannot be used to invalidate a batcher's committed action by changing their batcher ID, since the recipient must collect the ID themselves; the act of granting the ID only gives them the right to collect it, it does not forcibly modify the recipient's account.

Clients can end a client-batcher relationship within a limited period of time W_2, becoming independent and capable of spending from their own account via a direct call to MultiCall. Therefore, a malicious batcher cannot lock up user funds indefinitely. There is a gas overhead to escaping from a batcher, but it is a constant cost on the order of the fixed transaction cost and can be managed through off-chain contracts or batcher reputation. The interest cost of user funds being locked up for W_2 can be mitigated by selecting a W_2 less than a day. We recommend arbitrageurs absolutely dependent on access to their funds within a blocktime to not use a batcher at all.

[10] The astute reader may observe that predicting computational resource usage in a Turing complete system with unpredictable state is far from trivial. Fortunately, the EVM allows the gas usage of the CALL instruction to be bounded by passing a parameter to it. Direct call, proxy call and proxy create instructions are implemented using a CALL instruction. They take a gas limit parameter set by the client and pass it to the CALL, bounding the gas they can use. That makes them safe for the batcher: by setting a transaction gas limit higher than the total gas bounds plus the cost of executing instructions in MultiCall, the batcher can guarantee their transaction will succeed. Note that safety means only that the submitted metatransactions will complete without reverting the committed action; if clients set the gas bounds of their instructions too low the instructions will fail but the batcher will still get paid! Native Ethereum transactions have the same problem; it's inherent to making calls to Turing-complete smart contracts. Consequently, we don't consider it a fatal design flaw for MultiCall. For direct creates there is no native means of limiting the gas usage of the initialization EVM code; there static analysis by the batcher to establish an upper bound would be required. MultiCall instructions themselves are executed in sequence and consist of non-looping code; establishing a gas bound for them should be feasible.

8.3. Security assumptions

As MultiCall is an application built to run on Ethereum, it naturally relies on the security of the underlying system. However, it also places additional requirements on the capabilities of MultiCall users. To determine the degree of exposure to cryptographic vulnerability risk and whether the usage of MultiCall is suitable for a particular client, it is useful to make those assumptions and requirements explicit. They are as follows:

1. We assume Ethereum's signature scheme (which is used to sign metatransactions) is secure against forgery.
2. We assume the EVM's SHA3 hash, truncated to 160 bits, is sufficiently collision resistant to render finding any collision intractable.
3. We assume both clients and batchers can detect on-chain events on Ethereum, send a transaction in response to it, and have it accepted within some period W_1–W_2 and W_2, respectively.

Assumption 1 is trivial; if it did not hold, the Ethereum protocol itself would be broken.

Hash-based authorisation depends on the difficulty of finding a preimage to the image stored in the client's account to prevent metatransaction forgeries and on the difficulty for batchers of finding a new malicious action whose hash matches their committed action. In other words, the hash function used must be collision resistant to be secure—that is assumption 2. In both cases, the same hash function is used: the native SHA3 function provided by the EVM [1], truncated to the most significant 160 bits out of 256 returned. While truncation is necessary to fit the image into 256-bit user accounts alongside other fields, a hash size of precisely 160 bits was chosen because it coincides with the length of an Ethereum address. We assume that the hash is sufficiently long to prevent adversaries from finding collisions. Ethereum addresses are themselves the result of truncating the output of the EVM's native SHA3 function, using the ECDSA public key as input for externally owned accounts. Individual EOAs may hold billions of USD worth of Ether[11]; our reasoning is that if 160-bit SHA3 hashes were insufficiently collision resistant, the Ethereum protocol itself would be vulnerable. A difference from the Ethereum address hashing function is that our hash uses the leftmost rather than rightmost 160 bits, but we know of no result indicating they are more predictable for SHA3.

Both the hash-based authorisation and preferred batcher features require users to respond in a timely manner to actions taken by another user in order to avoid losing funds - that users are capable of doing so is assumption 3. Batchers may reset their committed action within W_1 seconds of setting it, which requires that any of their clients that had revealed their preimages end the client-batcher relationship within W_1–W_2 seconds. Clients may end the client-batcher relationship by calling MultiCall directly; that has a delay of W_2 seconds to give the batcher time to respond. If a batcher has committed an action which includes a metatransaction from that client, they must run the committed action within W_2 or be forced to reset their committed action (forcing clients to end the client-batcher relationship in turn, costing them money in transaction fees). Both cases require the users (client and batcher) to be able to observe events on the Ethereum blockchain and send a transaction that gets accepted within W_1–W_2 and W_2, respectively.

We assume that is possible; as such HBAuth is inappropriate for clients which will be offline for the entirety of the W_1–W_2 period after revealing a preimage. However, the task of freeing the client if the batcher misbehaves could be delegated to a watchdog node by sending it an Ethereum transaction which does that. That batchers must be responsive is less of a problem, as they must be to aggregate metatransactions. Currently, W_1 and W_2 are set to small numbers to aid testing, but in a deployed version of MultiCall appropriate values will need to be selected to ensure that the assumption is correct. W_1 could be set to a large value such as 1 month and W_2 to a small value such as 1 h; since the batcher is intended to be a sophisticated party and its capacity for malfeasance is more serious, it is reasonable to place a higher burden on it.

[11] https://etherscan.io/address/0x9BF4001d307dFd62B26A2F1307ee0C0307632d59.

9. Discussion

This work has focused primarily on introducing the concept of a batching interpreter, a prototype on-chain implementation, and improving that prototype's performance and security. Verification of the security of MultiCall with the help of existing tools such as Ref. [15] is a subject for future work. While the gas usage of the prototype has been profiled with good results, a number of challenges stand between MultiCall and real-world adoption.

9.1. Adoption

First of all, the question arises why batching has not yet achieved mass adoption, despite preexisting batchers that could also provide significant gas savings. We speculate that for individual users who do not frequently send many transactions at once, the effort of learning the user interface of an existing batching tool is not worth the money saved. MultiCall's ability to share calls between multiple users via metatransactions may ease the adoption of batching, as it makes even single calls from one signatory cheaper to batch than send directly. However, changes to wallet user interface software would be required to make batcher use effortless for the end user. The problem of aligning incentives of wallet client providers and batcher developers is beyond the scope of this paper.

9.2. HBAuth performance

The performance improvement of hash-based metatransactions relative to conventional signed ones is exciting. While a 40% cost reduction is impressive by itself, it bears noting that 5800 gas or 77% of the remaining cost is due to the account update rather than metatransaction verification. Excluding the account update cost for both hashed and signed metatransactions shows that the verification cost has been reduced by 75%! The obvious next step is to use Merkle trees rather than an array for account storage. Doing so could replace the storage write per user with a single shared write to the Merkle root, potentially bringing the overhead of a metatransaction below 2000 gas. That would be of great interest for future work. It bears noting that if signature-based transactions used a shared nonce as iBatch does (for example, the nonce in the batcher's account) rather than a separate one for each client, then signature-based transactions could be cheaper than hash-based as long as the client does not spend Ether from their account or otherwise modify it. However, that would be rendered moot by using Merkle tree storage, as long as accesses to it are dense enough to reduce the cost per element below 4740 gas (the cost of uploading and verifying a signature).

9.3. HBAuth limitations

While hash-based metatransactions provide a significant cost improvement, they have some disadvantages. Before they can be run, hash-based metatransactions must first be committed by the batcher. For optimal efficiency, that commitment should be done alongside the execution of the previous committed metatransactions in order to amortise the storage write. That means that if it takes a batcher T to collect enough metatransactions to send in a transaction, a client that sends a metatransaction at a random time must wait $1.5T$ beyond Ethereum's transaction acceptance latency (first $0.5T$ waiting for the metatransactions it will be sent alongside, then they are committed, then they are executed only when another set of metatransactions is ready to

be committed). Signature-based metatransactions need only wait 0.5T, because they do not need to be committed. That is due to unintended but beneficial behaviour: while each time a batcher runs metatransactions, they must reveal a committed set of them, they do not need to terminate them with an endCA() instruction. Then, the revealed metatransactions can be followed with signature-based ones chosen after commitment. In future versions, the endCA() instruction will be removed, as it serves no purpose. Another, more significant challenge to hash-based metatransactions is that in the event of a chain reorganisation the client's revealed preimage will be a carte blanche for the batcher to steal the client's entire account contents. Mitigating that requires waiting enough blocks after commitment to render a reorganisation very unlikely before revealing the preimage. However, it is our understanding that the planned transition from proof of work to proof of stake for Ethereum will make reorganisations very costly for stakers, so they will be improbable after a small number of blocks.

9.4. Use of watchdogs

An external watchdog server could be used to allow the client to fire and forget hash-based metatransactions by sending the watchdog a transaction which ends the user's client-batcher relationship. The watchdog would then be tasked with monitoring the blockchain and sending that transaction if and only if the batcher does not run the committed action within $W1 - W2$ less some buffer. There are two forms of malfeasance available to the watchdog: sending the transaction when they should not, and not sending the transaction when they should.

The first imposes a limited expense on the client but cannot be used to attack the batcher if it runs its committed action within $W2$ of committing it. Giving the watchdog a transaction signed by the client is simple from an engineering viewpoint, but has a problem: since the client can also send the transaction, the watchdog cannot be held accountable for sending it when it should not - after all, the client might have sent it themselves. Attribution could be enabled by extending MultiCall's instruction set with an "exit with attribution" instruction, which takes a signed request to end the client-batcher relationship from the client, ends it and records that the caller served the request.

Ideally, watchdogs would be incentivised via a smart contract to behave in the correct manner - that is, that they send or do not send certain transactions based on the state of the blockchain. A smart contract would then store locked funds and burn them or assign them to the parties based on proofs of contractual compliance or violation submitted by the parties. In theory, it would be possible for smart contracts to inspect Ethereum transactions sent by receiving a Merkle path from one of the last 256 block hashes (available via the EVM blockhash instruction). However, that would both require a significant engineering effort and might in the end be more costly than just using on-chain state to record actions (since one would need to explore a Merkle list full of unrelated data). A simpler solution would be to record contractually relevant actions such as "watchdog W sent an exit request for client C at T" directly in MultiCall's storage.

The second form of malfeasance (not sending an exit transaction when the watchdog should) is more serious than the first since it can lead to the user's entire balance and identity being stolen. One could increase redundancy by using multiple watchdogs, incentivising each to send an exit transaction by giving each an exclusive time window in which to do so (giving a bounty to the first watchdog that does so when required). Watchdogs could also have significant stake that is burned if they fail to perform.

However, if a significant amount of value is stored in MultiCall accounts (either directly in the balance or via token contracts that credit the user's proxy), it might be worth it for a malicious batcher to bribe the watchdogs in question. By selecting one or more high-value targets, preparing to perform denial of service (DoS) attacks on them and using DoS or bribes to silence their watchdogs, such an attack might be viable. Further work to investigate watchdog mechanisms and protocols as well as on-chain measurement of performance would be of interest.

It is possible that the catastrophic risk of an attack by a malicious batcher is not worth leaving to watchdogs to prevent; if so, it may be prudent to simply prevent batchers from resetting the committed action without running it (effectively setting $W1$ to infinity). Being unable to run the committed action due to its invalidity or losing the data would then require the batcher to allocate another account and could cause an exodus of clients. While expensive, that is not in the same order of magnitude as a major blockchain theft.

9.5. Generalisability

One might ask what general applicability HBAuth and the preferred batcher pattern have. After all, if they're dependent on some idiosyncrasy of MultiCall for viability then they would not be of theoretical value. The attack described in Section 5 should be present in any system where metatransactions are replay-protected, failing transactions cause fees to be wasted, and multiple users can batch the same metatransaction. We argue that both solutions could be ported to other smart contracts without much difficulty, as the state and code they use have been described and are independent of MultiCall's design. A mapping from users to account data is a common pattern, and extending accounts with the state used for HBauth or preferred batchers is trivial. While we have not tested this, we expect the features could also be implemented on other blockchains that support smart contracts such as the UTXO-based blockchain Cardano[12]. However, preferred batchers would not be as useful on UTXO-based blockchains since transactions cannot be invalidated the same way as in account-based blockchains: if any of the inputs to the transaction are used, the transaction will not be mined at all and will not cost the sender transaction fees. We expect HBAuth will be more widely applicable, and could serve as a generic replacement for signatures in both native and metatransactions.

9.6. Decentralisation

One potential objection to the approach of using an off-chain batching server would be that it imposes centralisation on a system designed to be decentralised, which might compromise censorship resistance. However, we argue that allowing multiple competing batching servers would render them analogous to mining pools. While it is true that a system without more entral nodes such as mining pools or exchanges would be more decentralised, in practice, the current arrangement is "decentralised enough" because all or most of them would have to collude to successfully impose transaction censorship. However, a drawback of the preferred batcher feature is that clients can only use one batcher at a time and must spend transaction fees and wait to switch. While a series of uncooperative batchers would be unable to indefinitely prevent the user from sending transactions, they could cause them delay and expense. We have considered two possible countermeasures to this: performance contracts using verifiable message delivery and randomisation of batchers. In the former solution, the batcher would agree in return for payment to batch the user's metatransactions within some maximum latency. An off-chain protocol and intermediaries would then be used to force the batcher to confirm a timestamped receipt of the client's metatransactions. That receipt could then be submitted alongside proof of non-performance of the batcher's obligation to send a metatransaction for the user (attainable by inspecting the user's nonce and seeing it is too low) to a smart contract which penalizes the batcher for non-compliance. While a protocol for verifiable message delivery could be of general use in peer-to-peer protocols (for example, to deter Lightning nodes [6] from censoring), designing it would be a significant effort.

The second approach is simpler but has a higher gas overhead. Randomisation of batchers would let users choose a pool of batchers rather

[12] https://cardano.org/.

than a single batcher. Each round (some fixed period), a random batcher from the pool would be selected to be the preferred batcher for the clients of the pool. If the number of clients grows large, it may be appropriate to appoint multiple batchers at once, assigning each to a subset of users. Through this method, clients would cycle through different batchers without having to spend gas themselves, protecting them from denial of service by batchers. The gas cost of randomly appointing batchers each round could be amortised over many clients.

9.7. Hinting

The technique we used to pass proofs of authorisation to runMetatxn, which we call hinting, has the potential to simplify user scripts by leaving the details of how to implement them to the batcher.

For example, consider an upgraded MultiCall-like batching interpreter smart contract where multiple user balances can be cached in memory. The instruction set would need to allow the batcher to load and store user accounts, but ideally user metatransactions need not be aware of that, analogous to the cache and virtual memory being transparent in off-chain systems. Consider a new user instruction pay(recipientAddress, amount): by receiving an index to the recipient's cached balance as a hint, the pay instruction can use caching while being oblivious to it.

Hinting can also be used to accelerate on-chain computation by shifting everything but verification of the result off-chain. For example, a lookup in an array of key-value pairs ordered by key can be reduced to $O(1)$ from $O(\log(N))$ by hinting the index of the relevant key. NP-complete problems are of course even more amenable to hinting, being reduced from exponential to polynomial on-chain work. We suspect that it may also be useful for accelerating memory management and garbage collection in a higher-level interpreter smart contract.

9.8. Hash-chain optimisation

In Appendix A, we made use of earlier work on efficient hash-chain traversals [16–20] to analyse how to minimise the cost of HBAuth's off-chain work.

If the storage cost per block-time for one hash U and computational cost for computing a single hash H are equal, then the existing algorithms for hash-chain traversal provide almost optimal caching schemes.

However, for $U \neq H$, we showed that one can in certain cases use Kim's scalable hash-chain traversal algorithm [19] to find a total cost less than that provided by Yum-Seo-Eom-Lee's algorithm [20] (representing the state-of-the-art when $H = U$).

Given a choice of caching scheme, we also computed the optimal choice of chain length under the assumption that the client uses MultiCall for a long period of time. This turned out to be approximated well by a convex optimisation problem, which was straightforward to solve.

In conclusion, we found that for a number of typical real-life estimations of the storage and computational cost (sourced from AWS EC2 for computation and AWS S3 for storage), it is optimal for the client to use a single chain until they reach around 10^9 hashes. That is enough for at least 422 years of maximal use, and hence clients should in practice plan to use one hash chain for all of their interactions with MultiCall.

10. Related work

MultiCall is a technology intended to reduce the cost of payment and contracting in the Ethereum ecosystem. Distributed ledger cost reduction techniques may broadly be categorised into two types: layer 1 optimisations and layer 2 scaling solutions. The former reduce the cost of blockchain (layer 1) execution, while the latter use off-chain distributed data structures secured by the layer 1 parent chain (layer 2) to move computation off the chain entirely. MultiCall is a layer 1 optimisation; to gain an overview of the general cost reduction problem, it is useful to first compare it with layer 2 solutions.

10.1. Layer 2 scaling solutions

Layer 2 scaling solutions move transaction processing and storage to another distributed ledger. That ledger does not need to independently guarantee consistency or availability, as it is secured by a layer 1 parent chain. Reliance on the parent chain for security allows the number of redundant nodes that maintain the child ledger to be drastically reduced, enabling simpler and cheaper consensus protocols to be used safely. Consequently, the cost of layer 2 computation, storage and consensus can be significantly lower than on layer 1.

The prototypical scaling solution is the state channel [6]. By locking on-chain assets in a contract controlled by a closed set of parties, those assets can be transferred between them off-chain. Off-chain transfers are effected by the parties all signing a ledger state, distributing ownership of the locked assets across them. Later ledger states invalidate earlier ones. By full consensus among the parties, the assets may be withdrawn from the state channel immediately. To prevent assets from being locked indefinitely if some parties refuse to cooperate, one may also submit a signed ledger state to the on-chain state channel contract. If no party submits a later ledger state within a certain warning period, that state becomes the definitive one, and the state channel is finalised. On-chain intervention invalidating the attempted action is only required if the attempt is fraudulent; that pattern may be termed counterfactual or optimistic verification, and is a fundamental tool of scaling solution design. State channels were first deployed on the Bitcoin blockchain in the form of payment channels; on Turing complete blockchains, more complex contracts may be entered into the off-chain ledger.[13]

While state channels render the computational cost of off-chain payments negligible, they have a significant disadvantage: only the parties to the channel can safely receive off-chain payments from it. That imposes a capital cost, as funds must be locked up and thus rendered unavailable for payments to arbitrary recipients. Plasma is a scaling solution designed by Poon and Buterin to address this issue [21]. Like state channels, Plasma provides an off-chain ledger. However, any number of users may have deposits in and receive payments through it. Instead of being performed fully off-chain, ledger updates require the hash of the ledger to be uploaded. By verifying the ledger, users may be assured that the hashed state is valid. After a certain time has elapsed, the uploaded state becomes final. Prior to that, users are able to appeal fraudulent state transitions. If the maintainer or maintainers are unwilling to make the off-chain state available, users can request the withdrawal of their balances; to block the withdrawal, it must be proven that the withdrawals are fraudulent.

Plasma ledgers may be considered an intermediate payment solution between state channels and on-chain payments: they may be used to pay an open set of recipients but have a latency of at least a blocktime. Plasma may have a significantly lower transaction cost than the main chain, but final withdrawal incurs a delay. Ledger updates require an on-chain access, but that can be amortised over many users' transactions. Consequently, Plasma may be used as a complement to state channels, for larger and less frequent payments and intermediate settlement to defer final on-chain settlement.

Layer 2 scaling is a field of active research; other scaling technologies include zero knowledge rollups[14], optimistic rollups [22] and TrueBit [23]. What layer 2 solutions have in common is that they rely on their parent chain for security: to ensure the consistency and availability of the layer 2 data structure and to ultimately enforce off-chain payments between users in terms of on-chain assets. Establishment, state commitment and ultimate settlement of the layer 2 state therefore requires interaction with the main chain. Such interaction can be optimised using layer 1 solutions. Consequently, layer 2 scaling solutions are complements rather

[13] https://funfair.io/a-reference-implementation-of-state-channel-contracts/.

[14] https://pandax-statics.oss-cn-shenzhen.aliyuncs.com/statics/1221233526992813.pdf.

than competitors to layer 1 optimisations. More concretely, any layer 2 interactions with the Ethereum blockchain require transactions to be sent, and those transactions can be batched. As such, they are not a threat to the validity of MultiCall or any other batcher.

10.2. Layer 1 gas optimisation

MultiCall is a layer 1 optimisation intended to reduce the execution cost of on-chain contract calls and creation. Layer 1 optimisation solutions can be divided into two groups: micro-level and macro-level optimisation. Micro-level optimisations optimise individual contracts to reduce their creation and execution costs without changing their externally observable behaviour. Macro-level optimisations save gas by restructuring smart contracts, changing their API and potentially the transaction workflow. Essentially, they optimise systems of contracts. MultiCall and other batchers are of the latter sort. The approaches are complementary: micro-level optimisations can be applied to contracts after their structure and API have been designed. However, there is reason to think macro-optimisation can provide larger savings: making a particular contract use less gas with the same behaviour will not enable it to use fewer transactions, for example. Other expensive operations, such as storage writes, may also be easier to eliminate by varying the design of multi-contract systems than optimising single contracts.

10.2.1. Micro-optimisation

Chen et al. [24] developed GASPER, a tool which searches for inefficient patterns in EVM bytecode. Applied to all contracts on the blockchain as of 2016, it showed a significant proportion of contracts were under-optimised. One example inefficient pattern was fetching a storage word in a loop, which is optimised by fetching it once and caching it.

In a spiritual sequel to Ref. [24], Chen et al. introduced GasReducer [25], a tool which finds more inefficient patterns and performs bytecode-to-bytecode optimisation. GasReducer is evaluated by tracing the EVM code execution of all transactions as of 2017. The evaluation showed that 9 billion gas was wasted on inefficient code patterns detected by GasReducer, vindicating the approach.

That Chen et al. scanned existing contracts and showed there are significant savings (in monetary terms) to be made is interesting; not only does it show the value of gas optimisation, it is an inspiring approach to evaluating on-chain artefacts. Since the cost model is formalised and the actual transaction history is publicly available, obtaining real and accurate performance data is much easier than on physical machines. Augmenting evaluation of MultiCall with real transaction history could be a subject for future work.

Albert et al. created a super-optimising tool for the EVM, which finds the optimal code for straight-line segments containing arithmetic and bitwise instructions by exhaustive search [26]. Using a data set of transactions to the 128 most-called smart contracts, they obtain a potential gas optimisation of 0.59%. We suspect that the dominance of the cost of instructions that access the ledger state compared to arithmetic is responsible for the small saving relative to that provided by batchers. Nonetheless, any gas cost reduction is welcome.

MultiCall consists of manually optimised EVM assembly and already uses techniques such as caching storage words, but it would be interesting to apply automatic optimisation to it. Optimising MultiCall was already a significant effort; manually optimising more complex interpreters with additional functionality may quickly become infeasible as they grow.

10.2.2. Macro-optimisation

The patterns used in MultiCall's design (batching, proxies and metatransactions) are well established, but the manner in which they have been combined is novel.

MultiCall is, to our knowledge, unique in being expressly designed to batch a full block of any type of transactions in one and is the first application of an interpreter smart contract to batching. Aside from its interpreter design and programming language used, the features provided by MultiCall differ in two main ways. First, it combines account structs in the batcher with metatransactions to allow multiple signatories to control a single batcher smart contract (MultiCall) in a single call. Secondly, MultiCall and the proxies it controls allow the batching of create transactions as well as calls (both directly from MultiCall and via a proxy).

10.2.2.1. Transaction batching, metatransactions and proxies. 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. Different batching techniques used for airdrops (mass transfers of a token intended to boost adoption of it) were studied and compared in Ref. [27]. Several payment batching contracts on the blockchain[15, 16], allow the caller to make payments in a single currency to a number of recipients. The company Autherium also provides wallet proxy contracts written in Solidity, which can batch calls on behalf of their owner[17]. Such contracts can receive an array of metatransactions (specifying a call to make) signed by the contract's owner, verify the signature of each and then execute them. MultiCall allows signature verification to be shared across a sequence of actions, which is more efficient. Storing batching logic in individual users' wallet contracts is also expensive because code storage costs gas.

The batcher MultiSend[18] (mentioned in Section 7) solves the code size issue by allowing user wallet contracts to delegate to a shared library using the EVM instruction DELEGATECALL, which executes the code of the callee in the storage context of the caller.

Existing batchers could be retrofitted with most of MultiCall's functionality by combining them: one batcher such as MultiSend can be used to send metatransactions to wallet contracts such as Authereum's, which accept them. Wallets that can only perform calls can be retrofitted with the ability to create contracts by calling a simple contract which just creates a contract using the given calldata as its creation code. We do not consider this a threat to MultiCall's validity: as calls are costly, a monolithic approach is efficient. MultiCall is a prototype of that approach.

Transaction batching is also used in Bitcoin: transactions natively support sending to multiple outputs, which can be used to reduce transaction costs by up to 80% [28].

In June 2021 (shortly after the publication of our first paper on MultiCall), Wang et al. described their transaction batching solution iBatch [5], which uses signed metatransactions to make calls on behalf of multiple users in one transaction. Unlike MultiCall, iBatch does not support batched creation of contracts nor hash-based metatransactions. It also lacks support for proxies (preventing it from granting a unique identity to each user); instead, the authors propose that the Ethereum protocol be forked to retrofit existing smart contracts with support for iBatch's own authentication scheme. Compared to our work with MultiCall, more focus has been placed on the off-chain component of batching: Wang et al. empirically analyse the interaction between latency (caused by waiting for more transactions to batch) and gas cost using historical transaction data from the Ethereum blockchain. For a 2-min delay, they find gas cost savings between 14.6% and 59.1%. That would appear to be worse than MultiCall's savings numbers for calls, but it is an apples-to-oranges comparison: Wang et al. measured the total cost reduction including fixed costs, whereas we measured the reduction in

[15] https://multisender.app/.
[16] https://etherscan.io/address/0x2f6321db2461f68676f42f396330a4dc4a8f49df#code.
[17] https://github.com/authereum/contracts/blob/master/contracts/account/AuthKeyMetaTxAccount.sol.
[18] https://github.com/gnosis/safe-contracts/blob/8443cfaa410bfb197cc708b1c5e06ffa0c49c217/contracts/libraries/MultiSend.sol.

marginal cost. We profiled iBatch and MultiCall's and found that MultiCall has a lower gas cost per batched call; evaluation of MultiCall on the same workloads as those used in the iBatch paper (or other realistic workloads) would be an interesting avenue for future work.

10.3. On-chain interpreters

Other smart contracts that implement interpreters have been developed for Ethereum in order to scale contract execution. The Optimistic Virtual Machine (OVM) is an interesting example, used by the Optimistic Ethereum scaling solution[19], an optimistic rollup chain. Optimistic rollup allows users to perform verification of transaction data off-chain, reducing the transaction cost. The rollup chain is secured by enabling users to verify its execution and prove fraud via a verifier contract on the Ethereum chain. To enable Ethereum smart contract execution on the rollup chain, the verifier contract implements an EVM code interpreter called OVM. Its purpose is quite different: where MultiCall instructions are run to perform actions on the blockchain, OVM is used to verify off-chain computations in the event of dispute. To be useful, MultiCall must be run, whereas the OVM need only be *available* to be run in the optimistic case. Moving computation off-chain can yield significant savings, but because off-chain scaling still requires transactions on the main chain to be secure, on-chain gas optimisation is not obsolete. An interpreter for a simple virtual machine called Lanai is also used to verify off-chain computations[20] as part of the TrueBit scaling solution [23].

10.4. Hash-based cryptography

Collision-resistant hashes have long been used as a fundamental building block of cryptographic protocols. Ironically, while we use hashing to avoid the use of digital signatures, the world's first digital signature scheme, invented by Leslie Lamport in 1975, relies on a one-way function (for which a hash function suffices) [29]. The Lamport signature scheme functions by creating pairs of committed hashes and revealing a preimage of one to communicate a bit of the content to be signed (or its hash).

Lamport later developed a protocol for authenticating computer users using one-time passwords [30] – that is, a chain of preimages. HBAuth closely resembles it, with a few key differences. Like HBAuth, Lamport's one-time password scheme uses a chain of hashes to authenticate the user. However, the protocols differ in their application domain and capabilities. Lamport's protocol contains no on-chain components (having been invented decades before the emergence of blockchains) and can be used by a computer user to authenticate themselves with a remote server. However, the server cannot prove to other parties that the user authorised a particular action. HBAuth, on the other hand, is intimately tied to distributed ledgers: it depends on actions committed on the blockchain to allow anyone to confirm that the client approved a particular action.

Hash commitment and revelation are also used to implement atomic swaps of cryptocurrency between different blockchains. Originally part of informal crypto developer knowhow, atomic swaps were studied more rigorously by Herlihy [10].

10.5. Hash-chain optimisation

We see in Appendix A that optimising the caching scheme for the usage of HBAuth is equivalent to finding efficient hash chain traversals. Efficient hash chain traversal has been studied extensively by Jakobsson [16], Coppersmith-Jakobsson [17], Sella [18], Kim [19], and Yum-Seo-Eom-Lee [20]. The algorithm by Yum-Seo-Eom-Lee is as far as we can tell the current state-of-the-art, assuming that the storage and computational costs are equal.

Kim's algorithm, building upon Sella's algorithm, has the particular strength of letting the client select the maximum number of hashes to perform per step. It can therefore be optimised accordingly. The total cost provided by Kim's algorithm coincides with the total cost provided by Jakobsson's algorithm when the computational budget is maximal. In certain cases, Kim's algorithm can provide a lower total cost than Yum-Seo-Eom-Lee's algorithm. Kim's algorithm, however, depends on having a certain lower bound on the computational cost in terms of the hash chain length, which made it impossible to use in our AWS-based examples.

Given a caching scheme, the problem of choosing an optimal chain length turned out to be approximated well by minimising a certain convex function (representing the sum of the costs per chain). This is a very a mature area of research (as demonstrated by the abundance of textbooks on the subject), and nothing novel was involved in the computation of the result.

11. Conclusions

We have implemented MultiCall, an interpreter for the Ethereum blockchain whose instruction set is designed for batching transactions. We demonstrated significant savings for both micro-benchmarks as well as a typical token-transfer smart contract, and MultiCall's performance is compared favourably to two preexisting batchers, one of which was also written at a low level of abstraction in order to maximise performance.

Our idea of deploying batching interpreter smart contracts with rich instruction sets has been shown to be an effective and systematic mechanism to substantially save gas.

MultiCall's metatransaction and deposit features allow multiple users to operate the batcher in a single call and to interact with other users. It is an example of how allowing safe resource sharing between mutually distrustful parties (the signatories) can help reduce costs.

Hash-based metatransactions have been implemented and evaluated for efficiency; they are significantly cheaper than their signature-based equivalents.

An economic attack on metatransaction batching has been conceived of and prevented; the prevention method has been systematically described to allow reuse by other systems.

As future work, we plan to upgrade MultiCall with Merkle tree storage to reduce costs further. We also wish to investigate whether it is possible to amend Kim's algorithm in such a way that its total cost coincides with the total cost of Yum-Seo-Eom-Lee's algorithm when the budget is maximal.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

Partially supported by the Swedish Research Council (Vetenskapsrådet) under grant No. 2019-04951 (X-LEGAL: Smart Legal Contracts).

[19] https://optimism.io.

[20] https://github.com/TrueBitProject/lanai.

Appendix A. Optimising the off-chain caching scheme for HBAuth

To use HBAuth, the client must generate and then use a chain of hashes. In the implementation of their wallet code, one could choose to only store the seed k, and compute $(k, h(k), \ldots, h^i(k))$ when $h^i(k)$ as needed, or one could compute the entire chain $(k, h(k), \ldots, h^n(k))$, and then store it for the duration of its use (requiring no re-computations).

Neither of these options are necessarily optimal. Indeed, they can be seen as being on opposite ends of a spectrum – the former using as little storage as possible, and the latter performing as little computation as possible.

If the client instead wants to minimise the *sum* of the computational costs and storage costs, they can choose to store (cache) only a subset of the hashes, say

$$\{h^{i_1}(k), \ldots, h^{i_j}(k)\},$$

and compute $(h^{i_1+1}(k), \ldots, h^{i}(k))$ when $h^i(k)$ is needed. The question then arises:

$$\text{How to pick the subsets so that the sum of the computational costs and the storage costs (total cost) is minimal, and how long should the chain be?} \tag{A.1}$$

In this section, we answer this question by relating it to hash-chain traversals. This has been studied extensively by Jakobsson [16], Coppersmith-Jakobsson [17], Sella [18], Kim [19], and Yum-Seo-Eom-Lee [20], among others. The cited works all provide algorithms that select the subsets so that the total cost is $O(N\log(N))$, where N is the length of the hash chain[21]. Coppersmith-Jakobsson have in particular proven that any algorithm which solves the problem gives a total cost that is $\Omega(N\log(N))$.

Appendix A.1. The model

We assume that a hash is used for authorisation every $T \in \mathbb{R}_{>0}$ seconds, that it costs $H \in \mathbb{R}_{>0}$ dollars to compute a hash, and that it costs $S \in \mathbb{R}_{>0}$ dollars per second to store a hash. We further assume that the time required to compute a hash is negligible.

Suppose now that we want to use $N \in \mathbb{Z}_{\geq 1}$ hashes (that is, one initialization, and $N-1$ authorisations), following the below procedure.

Algorithm 1. Computing hashes and caching some of them.

```
randomly generate k ∈ Z_{≥0};
let C_N = {1, ..., N};
compute h^l(k) for l ∈ C_N;
initialise MultiCall with h^N(k);
let S_N be subset of {k, h(k), ..., h^{N−1}(k)} including k;
store S_N in memory;
for N − 1 ≥ i ≥ 1 do
    let j = max{j : h^j(k) ∈ S_{i+1}};
    let C_i = {j + 1, ..., i};
    compute h^l(k) for l ∈ C_i;
    wait for h^i(k) to be used for authorisation;
    use h^i(k) for authorisation;
    let S_i be a subset of {h^{j+1}(k), ..., h^{i−1}(k)} ∪ S_{i+1}, including k;
    flush S_{i+1} from memory and store S_i;
end
```

With the same notation as above, we let

$$\sigma(N)(C_1, \ldots, C_N, S_1, \ldots, S_N) = \sum_{1\leq i\leq N} (H|C_i| + ST|S_i|).$$

In what follows, we will use the shorthand notation $\boldsymbol{C} = (C_1, \ldots, C_N)$ and $\boldsymbol{S} = (S_1, \ldots, S_N)$ and refer to a choice of $(\boldsymbol{C}, \boldsymbol{S})$ as an N-configuration. We will also use the shorthand notation $U = S \cdot T$ and $\alpha = U/H$.

Appendix A.2. Cheap N-configurations

Let N be a fixed positive integer. The question (A.1) then translates to finding

$$\min\nolimits_{(\boldsymbol{C},\boldsymbol{S})}\sigma(N)(\boldsymbol{C}, \boldsymbol{S}). \tag{A.2}$$

Using the terminology of Refs. [16–20], we refer to the memory slots where the hashes in S_i are stored as pebbles, and the number $\max_i |C_i|$ as the budget.

The hash-chain traversal algorithms that yield the lowest total cost are Kim's algorithm and Yum-Seo-Eom-Lee's algorithm. They differ in that Kim's algorithm allows for a flexible budget, whereas Yum-Seo-Eom-Lee's algorithm has a fixed budget. However, if the computational and storage costs are equal, Yum-Seo-Eom-Lee's algorithm provides (at the time of writing) the lowest total cost.

The total cost provided by Kim's algorithm can be optimised with respect to the budget. One obtains an optimum m_0 that can be explicitly expressed in terms N, α, and Lambert's W_0-function. The derivation is provided at [31].

[21] Where the constant is approximately equal to the sum of the storage cost for one memory unit and one second, and the cost for one computational unit.

We want to determine when Kim's algorithm yields a lower total cost than Yum-Seo-Eom-Lee's algorithm. This requires solving a two inequalities for the optimum m_0.

Due to the presence of W_0, this is not feasible to approach analytically, and therefore we opted for a numerical approach. Using SciPy (the code can be found at [31]), we computed ranges of α where Kim's algorithm provides a lower total cost than Yum-Seo-Eom-Lee's algorithm, for $\lceil N \rceil$ where

$$N \in \frac{30.4375 \cdot 24 \cdot 3600}{T} \cdot 10^6 \{1, \ldots, 12, 24, 36\},$$

where T is the average block-time. This corresponds to the amount of hashes used in 1 month, 2 months, and so on up to a year, and then for two and three years[22]. At the time of writing, T was approximately 13.32 s [2]. The result is presented in Table A.4. We remark that $\alpha_{\min} \to 0$, and $\alpha_{\max} \to 0.1265$ as $N \to \infty$.

Table A.4
Ranges of α for which Kim's algorithm provides a total cost lower than Yum-Seo-Eom-Lee's algorithm. The time is in months.

Time	$\alpha_{\min}$	$\alpha_{\max}$	Time	$\alpha_{\min}$	$\alpha_{\max}$
1	0.1805	0.1821	8	0.1468	0.1731
2	0.1678	0.1787	9	0.1452	0.1727
3	0.1611	0.1769	10	0.1439	0.1723
4	0.1567	0.1758	11	0.1427	0.1720
5	0.1533	0.1749	12	0.1416	0.1717
6	0.1507	0.1742	24	0.1334	0.1694
7	0.1486	0.1736	36	0.1290	0.1682

Appendix A.3. Optimal chain lengths

Let us now suppose that we need to use M hashes, for a total duration of $M \cdot T$ seconds. We then ask: how do we choose the amount of chains n_c and the chain lengths N_i so that $\sum_{i=1}^{n_c} N_i T = MT$ and such that

$$\sum_{i=1}^{n_c} (\sigma(N_i)(\boldsymbol{C}_i, \boldsymbol{S}_i) + R_{\text{set}}),$$

is minimal? Here $(\boldsymbol{C}_i, \boldsymbol{S}_i)$ denotes an N_i-configuration, and R_{set} is the cost of starting a new chain (that is, running setImage). We ignore the cost of uploading the very first hash, since this only occurs once per client. Furthermore, we will assume that the same algorithm is used for all computations.

In practice, we use either Kim's algorithm or Yum-Seo-Eom-Lee's algorithm, which means that we want to minimise

$$\sum_{i=1}^{n_c} (f(m_i^*)(N_i) + R_{\text{set}}) \quad \text{or} \quad \sum_{i=1}^{n_c} (g(N_i) + R_{\text{set}}), \tag{A.3}$$

where $f(m)(N)$ is the total cost for Kim's algorithm with budget m and $g(N)$ is the total cost for Yum-Seo-Eom-Lee's algorithm, and where m_i^* is a minimal point for $f(m)(N_i)$.

The expressions (A.3) can be minimised using essentially the same techniques. For the computations, we refer to our appendix [31].

One finds that the global minimum is attained when $N_i = M/n_c$ for all i, and when

$$\frac{M}{n_c} = (1 + \Delta)\frac{R_{\text{set}}}{\beta},$$

where $\beta = \log(2)^{-1}(H/2 + U)$ if Yum-Seo-Eom-Lee's algorithm is used, and

$$\beta = (H + U(\exp(s_0 + 1) - 1)) \Big/ (s_0 + 1), \quad s_0 = W_0\left(\frac{\alpha^{-1} - 1}{e}\right)$$

if Kim's algorithm is used, and where $\Delta \to 0$ as $M \to \infty$.

We may assume that M is very large and thus we may approximate the optimal chain length with $\lceil R_{\text{set}}/\beta \rceil$.

Appendix A.4. Real-life estimations of optimal chain lengths and costs

We will now compute a few real-life approximations of α and the corresponding optimal chain lengths. We source our costs from AWS and will

[22] Note that setting T equal to the block-time implies metatransaction authorization once per block. That is very frequent but may be realistic for sophisticated actors such as financial institutions, exchanges or miners. Sending transactions less frequently increases T and consequently the storage cost relative to computation. Fortunately, increasing T does not threaten our conclusion - that it is optimal to use a chain so long that it would not feasibly run out in the probable lifetime of MultiCall.

assume that the hashes are stored on an S3 instance [32], and that they are computed on an EC2 instance [33].

For an EC2 instance with a computational price per hour of p_c \$/h, a hash rate of r H/s (hashes per second), we use the approximation

$$H = \frac{p_c}{3600 \cdot r} \$/\mathrm{H}.$$

Similarly, for an S3 instance with a monthly storage price of p_s for 1 GiB, we use the approximation

$$U = \frac{20 \cdot p_s \cdot T}{30.4375 \cdot 24 \cdot 3600 \cdot 1024^3} \$ \Big/ (\text{hash stored for } T \text{ seconds}).$$

Hence, with a monthly storage price of p_s, and computational price of p_c \$/h, we obtain

$$\alpha = \frac{20}{30.4375 \cdot 24 \cdot 1024^3} \cdot \frac{T \cdot p_s \cdot r}{p_c}.$$

We collected data from four EC2 instances, g3s.xlarge, g4dn.xlarge, p2.xlarge, and p3.2xlarge, all located in the region US East (Ohio) [34]. Furthermore, we used the S3 storage price of 0.023 $ for one month and 1 GiB of storage [35].

In Table A.5, we present our corresponding approximations. We remark that the optimal chain lengths for Kim's algorithm in conjunction with the estimated αs violate the conditions required by Kim's algorithm. Hence, in all of the computed cases, Yum-Seo-Eom-Lee's algorithm should be used instead.[23]

In particular, we notice that in the cases we have considered, the optimal chain length is on the order of 10^9, which corresponds to at least 422 years. Therefore, the client should plan to use a single chain of hashes, allocating as many as they will ever use from the beginning.

Table A.5
Approximated values of α for four different EC2 instances. Here Yocl means optimal

EC2	p_c	$r \cdot 10^{-6}$	$\alpha \cdot 10^5$	Yocl $\cdot 10^{-9}$	\$/chain
g3s.xlarge	0.225	2.64	9.1657	5.6469	2.2731
g4dn.xlarge	0.1578	22.72	112.4717	69.1501	2.5403
p2.xlarge	0.27	1.83	5.2946	3.2622	2.1393
p3.2xlarge	0.918	78.01	66.3819	40.8507	2.4068

Chain length with Yum-Seo-Eom-Lee's algorithm, $T = 13.32$ s and $p_s = 0.023$ \$/month for 1 GiB.

References

[1] G. Wood, ETHEREUM: A Secure Decentralised Generalised Transaction Ledger, Istanbul Version 80085f7 – 2021-07-11, 2021. URL, https://ethereum.github.io/yellowpaper/paper.pdf.
[2] Etherscanio, Ethereum (ETH) blockchain explorer, accessed: 2021-08-12. URL https://etherscan.io/.
[3] EIP-20, ERC-20 token standard, accessed: 2021-01-29, https://eips.ethereum.org/EIPS/eip-20, 2015. URL.
[4] W. Hughes, A. Russo, G. Schneider, Multicall: a transaction-batching interpreter for Ethereum, in: Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, ACM, 2021, pp. 25–35, https://doi.org/10.1145/3457337.3457839.
[5] Y. Wang, Q. Zhang, K. Li, Y. Tang, J. Chen, X. Luo, T. Chen, ibatch: saving ethereum fees via secure and cost-effective batching of smart-contract invocations, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2021, pp. 566–577, https://doi.org/10.1145/3468264.3468568.
[6] J. Poon, T. Dryja, The bitcoin lightning network, accessed: 2021-01-29, https://lightning.network/lightning-network-paper.pdf, 2016. URL.
[7] Solidity documentation, The ethereum foundation, accessed: 2021-01-29, https://solidity.readthedocs.io/en/v0.8.1/, 2021. URL.
[8] I.A. Seres, On blockchain metatransactions, in: Proceedings of 2020 IEEE International Conference on Blockchain (Blockchain), IEEE, 2020, pp. 178–187, https://doi.org/10.1109/Blockchain50366.2020.00029.
[9] S. Nakamoto, Bitcoin: a peer-to-peer electronic cash system, URL, http://bitcoin.org/bitcoin.pdf, 2009.
[10] M. Herlihy, Atomic cross-chain swaps, in: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC '18, ACM, 2018, pp. 245–254, https://doi.org/10.1145/3212734.3212736.
[11] R.S. Bird, Using circular programs to eliminate multiple traversals of data, Acta Inf. 21 (3) (1984) 239–250, https://doi.org/10.1007/BF00264249.
[12] Accessed: June 2022. [link]. URL https://trufflesuite.com/truffle/.
[13] Accessed: June 2022. [link]. URL https://trufflesuite.com/ganache/.
[14] V. Buterin, M. Swende, Eip-2929: gas cost increases for state access opcodes, accessed: 2021-08-15 (9 2020). URL https://eips.ethereum.org/EIPS/eip-2929.
[15] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, G. Rosu, Kevm: a complete formal semantics of the ethereum virtual machine, in: Proceedings of the 2018 IEEE 31st Computer Security Foundations Symposium (CSF), IEEE, 2018, pp. 204–217, https://doi.org/10.1109/CSF.2018.00022.
[16] M. Jakobsson, Fractal hash sequence representation and traversal, in: Proceedings IEEE International Symposium on Information Theory, IEEE, 2002, p. 437, https://doi.org/10.1109/ISIT.2002.1023709.
[17] D. Coppersmith, M. Jakobsson, Almost optimal hash sequence traversal, in: M. Blaze (Ed.), Financial Cryptography. FC 2002, Springer, Berlin, Heidelberg, 2003, pp. 102–119, https://doi.org/10.1007/3-540-36504-4_8.
[18] Y. Sella, On the computation-storage trade-offs of hash chain traversal, in: Financial Cryptography. FC 2003, Springer, Berlin, Heidelberg, 2003, pp. 270–285, https://doi.org/10.1007/978-3-540-45126-6_20.
[19] S.-R. Kim, Improved scalable hash chain traversal, in: J. Zhou, M. Yung, Y. Han (Eds.), Applied Cryptography and Network Security, Springer, Berlin, Heidelberg, 2003, pp. 86–95, https://doi.org/10.1007/978-3-540-45203-4_7.
[20] D.H. Yum, J.W. Seo, S. Eom, P.J. Lee, Single-layer fractal hash chain traversal with almost optimal complexity, in: M. Fischlin (Ed.), Topics in Cryptology – CT-RSA 2009, Springer, Berlin, Heidelberg, 2009, pp. 325–339, https://doi.org/10.1007/978-3-642-00862-7_22.
[21] J. Poon, V. Buterin, Plasma: scalable autonomous smart contracts, White paper (2017) 1–47. URL, https://www.plasma.io/plasma-deprecated.pdf.
[22] J. Adler, M. Quintyne-Collins, Building Scalable Decentralized Payment Systems, 2019 arXiv preprint arXiv:1904.06441.
[23] J. Teutsch, C. Reitwießner, A Scalable Verification Solution for Blockchains, 2019, 04756 arXiv preprint arXiv:1908.
[24] T. Chen, X. Li, X. Luo, X. Zhang, Under-optimized smart contracts devour your money, in: Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 442–446, https://doi.org/10.1109/SANER.2017.7884650.
[25] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, X. Zhang, Towards saving money in using smart contracts, in: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results, ACM, 2018, pp. 81–84, https://doi.org/10.1145/3183399.3183420.
[26] E. Albert, P. Gordillo, A. Rubio, M.A. Schett, Synthesis of super-optimized smart contracts using max-smt, in: S.K. Lahiri, C. Wang (Eds.), Computer Aided Verification, Springer, Cham, 2020, pp. 177–200, https://doi.org/10.1007/978-3-030-53288-8_10.
[27] M. Fröwis, R. Böhme, The operational cost of ethereum airdrops, in: C. Pérez-Solà, G. Navarro-Arribas, A. Biryukov, J. Garcia-Alfaro (Eds.), Data Privacy Management, Cryptocurrencies and Blockchain Technology, Springer, Cham, 2019, pp. 255–270, https://doi.org/10.1007/978-3-030-31500-9_17.

[23] We remark that if the storage price would be considerably higher (say 2.3 $/month), this is no longer true. In this case, Kim's algorithm provides a lower cost than Yum-Seo-Eom-Lee's algorithm. See also the functions alpha_lower_bound_check and alpha_upper_bound_check in the online appendix [31].

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

[29] L. Lamport, Constructing digital signatures from a one way function, Tech. rep. (October 1979). URL, https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/.

[30] L. Lamport, Password authentication with insecure communication, Commun. ACM 24 (11) (1981) 770–772, https://doi.org/10.1145/358790.358797.

[31] W. Hughes, T. Magnusson, Multicall, optimization appendix, URL, https://tobiasmagnusson.com/notes/multicall/, 2021.

[32] Amazon, Amazon s3, URL, https://aws.amazon.com/s3/, 2021.

[33] Amazon, Amazon ec2, URL, https://aws.amazon.com/ec2/, 2021.

[34] Amazon, Amazon ec2 on-demand pricing, accessed: 2021-08-12. URL https://aws.amazon.com/ec2/pricing/on-demand/.

[35] Amazon, Aws pricing calculator, s3, accessed: 2021-08-12. URL https://calculator.aws/#/createCalculator/S3?nc2=h_ql_pr_calc.