usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Hamilton: A High-Performance Transaction Processor for Central Bank Digital Currencies

James Lovejoy, *Federal Reserve Bank of Boston;* Madars Virza and Cory Fields, *MIT Media Lab;* Kevin Karwaski and Anders Brownworth, *Federal Reserve Bank of Boston;* Neha Narula, *MIT Media Lab*

https://www.usenix.org/conference/nsdi23/presentation/lovejoy

This paper is included in the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Hamilton: A High-Performance Transaction Processor for Central Bank Digital Currencies

James Lovejoy
Federal Reserve Bank of Boston

Madars Virza
MIT Media Lab

Cory Fields
MIT Media Lab

Kevin Karwaski
Federal Reserve Bank of Boston

Anders Brownworth
Federal Reserve Bank of Boston

Neha Narula
MIT Media Lab

Abstract

Over 80% of central banks around the world are investigating central bank digital currency (CBDC), a digital form of central bank money that would be made available to the public for payments. We present Hamilton, a transaction processor for CBDC that provides high throughput, low latency, and fault tolerance, and that minimizes data stored in the transaction processor and provides flexibility for multiple types of programmability and a variety of roles for financial intermediaries. Hamilton does so by decoupling the steps of transaction validation so only the validating layer needs to see the details of a transaction, and by co-designing the transaction format with a simple version of a two-phase-commit protocol, which efficiently applies state updates in parallel. An evaluation shows Hamilton achieves 1.7M transactions per second in a geo-distributed setting.

1 Introduction

Central banks are increasingly investigating general-purpose central bank digital currency (CBDC), a digital currency that would be broadly available to users making retail payments, could provide interoperability and programmability depending on how it is designed, and, because it would be a direct liability of the central bank, reduces risk [7, 8, 14, 15, 19, 22, 23, 24, 37, 47, 65].

Figure 1 summarizes the different properties of a CBDC as compared to other forms of payment instruments [9]. A CBDC could help address public policy objectives such as ensuring public access to central bank money, fostering payment competitiveness and resilience, supporting financial inclusion, and offering privacy-preserving digital payments [3, 7, 15, 42, 70].

Technical designs for CBDC vary depending on specific policy requirements and goals. For example, a CBDC could provide low value payments in an anonymous, peer-to-peer fashion, or be distributed and accessed only through accounts at approved financial institutions. To better inform policy discussions, central banks are recognizing the importance of technical experimentation in understanding the implications and trade-offs of different CBDC models and other policy choices. Importantly, the feasibility, operating performance and impact of different CBDC policy choices are dependent upon the technical design of the underlying transaction processor.

This paper presents a collaboration with a major central bank in the design of Hamilton, a high-performance transaction processing system flexible enough to support experimentation with different choices around data storage, programmability, and intermediation. Hamilton processes payments from users (or financial institutions) who address and sign transactions using cryptographic keys stored in digital *wallets*. Wallets submit transactions to the Hamilton transaction processor to move *unspent funds*—a representation of money containing an amount and the rules required to spend it (in our case, a public key indicating ownership). Our initial goals (set by the central bank) were a centralized transaction processor for CBDC with high performance and geo-replicated resiliency. Three additional goals emerged within the collaboration:

Intermediary and custody flexibility. An open question in CBDC design is that of the role of the central bank and other intermediaries, and determining how (if at all) CBDC access can be moderated. These roles will likely vary by jurisdiction, due to policymaker decisions and consumer preferences. Currently, people who want to digitally store funds and make payments must open accounts with financial institutions or payment service providers which are linked to the identity of the owner and are responsible for processing transactions on behalf of their customers, interfacing with payment networks, and safeguarding customer funds. In contrast, cash can be held directly by the public and used to conduct transactions without the need for a financial institution to process the payment on their behalf. A CBDC could be designed to offer similar functionality to cash and provide users the ability to spend their own funds without the need for an account provider or custodian to generate transactions, it could be designed more like existing digital payment

The views expressed in this paper are those of the authors and do not necessarily reflect the views of the Federal Reserve Bank of Boston or the Federal Reserve System.

Property	Cash	Bank deposits	Cryptocurrency	Central bank reserves	CBDC
Electronic		✓	✓	✓	✓
Central-bank issued	✓			✓	✓
Universally accessible	✓	✓	✓		✓

Figure 1: Table describing the properties of various monetary instruments, summarized from Graph 3 in [9].

systems, or it could even support a combination of the two models [17].

Interoperability and programmability. Many payment processor systems provide high throughput and low latency, but unfortunately, they generally provide limited application programming interfaces (APIs) and do not natively *interoperate*. For example, if a Venmo user wants to pay a user who only has Square Cash, they would need to transfer their funds outside the application into bank accounts and send the money via traditional banking rails (inside the United States using ACH or FedWire; across borders, even in the same currency, using SWIFT and correspondent banking), which incurs bank and application fees and could take days to complete. Central bank Real-Time Gross Settlement Systems (RTGS), or instant payment systems, help reduce interoperability latency but do not preclude the requirement for multi-step transfers between applications and bank accounts (where fees are charged) or provide interoperability and programmability, especially between payment applications. Contrast these systems with cryptocurrencies which do not scale well but natively provide interoperability and programmability.

Preserving privacy and minimizing data retention. There is strong user demand for financial privacy [37] and central banks would prefer not to collect and store user-identifying information or sensitive transaction details.

Challenges. Building Hamilton to achieve these goals required addressing the following challenges. First, we had to build a flexible platform that could support multiple designs without explicit policy requirements or well-defined tradeoffs. For example, it is unclear what balance to target between end-user privacy and data storage requirements for users at the central transaction processor. We take a layered approach with a design where additional functionality can be built outside the core transaction processor. Our design supports a range of intermediary roles including one where users custody their own funds. Hamilton does not store personally identifiable information (PII), transaction addresses or amounts in the core of the system.

The second challenge is in providing strong consistency, geographic fault tolerance, high throughput, and low latency, all with a workload that consists of 100% read/write, multi-server transactions. Since Hamilton is unaware of the mapping between users and unspent funds, we cannot rely on user locality for partitioning, which is often exploited by traditional database systems to make workloads predominantly single-partition transactions.

Key ideas. We address these challenges in Hamilton by carefully co-designing the transaction format, data model, and distributed transaction commitment protocol to achieve the above goals while getting good performance. This involves three parts: First, we decouple transaction validation from fund existence checks; only user wallets and a *validating layer* see transaction details. Hamilton only ever stores funds as opaque 32 byte hashes, in an *Unspent funds Hash Set*, or UHS [41] (§3.3). This hides details about the funds (like amounts and addresses) from the UHS storage, reduces storage requirements, and creates opportunities to improve performance, described below.

Next, we next create a UHS-designed transaction format (§3.4), which is extensible and secure against double spends, inflation attacks, replay attacks, and malleability, and also has the benefit of supporting future layer 2 designs for even higher throughput in the future. It borrows from Bitcoin's transaction format but is designed to be validated without looking up data from the UHS, which we term *transaction-local validation*.

Our design choices let us exploit payment application semantics to create a streamlined commit protocol for distributed transactions. In particular, our transaction format guarantees that Hamilton knows the read and write sets for every valid transaction *before* its execution; similarly, our cryptographically-generated UHS identifiers (hashes) are globally unique. These two properties guarantee that Hamilton does not need any reads or locks before commit time, and that valid transactions (those that do not try to spend the same funds twice) will never conflict.

Our evaluation shows that co-design achieves improved performance over more general, commercial databases. We measured Hamilton's throughput at 1.7M txns/sec, 26$\times$ that of PostgreSQL on the same workload, though with higher latency (§6). We also compare against Rolis [64], a replicated in-memory database, and show that Hamilton achieves close performance, even though it stores data on disk for whole system crash recovery.

The UHS design, in combination with our transaction format, also affords us substantial flexibility. We believe that the abstractions our system provides and the assumptions it makes are compatible with most ideas underlying certain types of programmability and cryptographic privacy-preserving designs [10, 52, 69, 71]. In addition, we can upgrade the scripting language or add a cryptographic privacy-preserving protocol (even supporting multiple concurrent designs), as long as they are compatible with 32-byte hash storage, without needing any changes to the backing UHS, making it possible to defer decisions on specific programmability features. However, our design choices have implications on what data users or intermediaries need to store in their wallets and what messages are required to confirm a payment (§3.7).

In summary, the contributions of this paper are the following:

- Hamilton, a flexible transaction processor design that supports a range of models for a CBDC and minimizes data storage in the core transaction processor while supporting self-custody or custody provided by intermediaries
- A transaction format and implementation for a UHS which together support modularity and extensibility
- An implementation and evaluation which shows good performance in comparison to centralized databases. Hamilton and the benchmarks are available at https://github.com/mit-dci/opencbdc-tx.

2 System model and security goals

This section describes the actors in Hamilton, their roles, and the security properties we want Hamilton to satisfy. In our description, we make the simplifying assumption that users directly custody their money without help of an intermediary. Hamilton supports intermediaries, and adding an intermediary would not change the core security properties of the transaction processor.

2.1 Actors

We distinguish three types of actors: the *transaction processor*, the *issuer*, and *users*. The transaction processor keeps track of *funds* which are owned by different users. Funds are a representation of money and as such refer to an amount of money (such as dollars) and a condition that must be satisfied to move this amount (say, to another user or users). The funds enter and exit the system through acts of the *issuer* who can *mint* and *redeem* funds to add and remove them from the transaction processor, respectively. Users can execute *transfer* operations (transactions or payments) that atomically change the ownership of funds, with the requirement that the total amount of funds stored in the transaction processor has not changed. A user does so by submitting their transaction to the transaction processor over the Internet, which the processor then validates and executes. Figure 2 shows the high-level system model and potential communication channels between users and the transaction processor. Users run *wallet* software (e.g. on mobile phones or specialized hardware in smart cards) to manage cryptographic keys, track funds, and facilitate transactions. An important piece of future work is preventing spam and denial of service attacks.

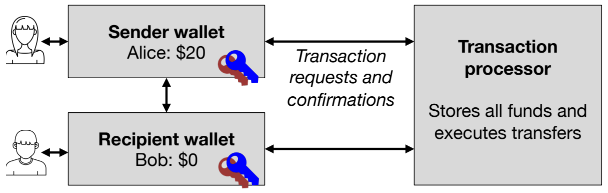
Sender wallet
Alice: $20
Recipient wallet
Bob: $0
Transaction requests and confirmations
Transaction processor
Stores all funds and executes transfers

Figure 2: Data flows between all participants in a transaction.

2.2 Threat model

Our goal is that each user's funds and the integrity of the monetary system are safe from interference of an external actor. We assume that the transaction processor is faithfully executing our design, that users' wallets are able to maintain secret keys, and that the users are able to use a secure channel to communicate with the transaction processor. Our design is a cryptographic system so we assume the security of standard cryptographic primitives such as hash functions and digital signatures.

We aim to protect against an adversary who can freely interact with the system as a regular user, and as such make no additional assumptions about an adversary's capabilities or behavior. For example, the adversary is free to create arbitrarily many identities and wallets, receive funds from other users, and engage in elaborate transaction patterns. Our designs are multi-server systems and the adversary is free to attempt concurrent attacks against all externally-exposed parts of the system.

2.3 Data representation

The two most common ways to represent funds are the account balance model and the UTXO model.

Account model. Traditional payment systems and several cryptocurrencies, like Ethereum [73] use an account model where the system stores unspent funds as balances associated with unique account identifiers. Users make payments by issuing requests to the transaction processor to move balance to another identifier (decrementing their balance and incrementing another identifier's balance).

UTXO model. Another choice is to track discrete pieces of outstanding funds without explicitly consolidating them in a single balance. For example, Bitcoin maintains an append-only ledger of accounting entries (sometimes called "coins") each of which records a value and conditions to spend the funds. Furthermore, each entry is either marked as "spent" or "unspent". Users make payments by issuing transactions that mark some entries (*inputs*) as spent, and appends new unspent entries (*outputs*) to the ledger. In Bitcoin these are called *UTXOs* or Unspent Transaction Outputs. Importantly, UTXOs are never modified and must be spent in their entirety. Therefore, Alice wishing to use a $10.00 UTXO to send $4.99 to Bob will create a transaction with two outputs: a $4.99 output meant for Bob and a $5.01 *change output* meant for herself.

We derive Hamilton's data model from the UTXO model for two reasons: First, because it offers greater transaction execution parallelism; inputs can only be spent once, and so in the common case there should never be conflicting transactions, unlike concurrent transactions against a popular account. We leverage this in our data storage design (§3.4). Second, UTXOs are the leading choice for privacy designs [10, 16, 26, 45, 48, 69, 71] (including for those deployed on top of account-based currencies [61, 72]). UTXOs can be less intuitive to the user, but note that the transaction processor's internal data representation is distinct from the user interface; wallets can still present a user account balance on top of the UTXO data representation.

2.4 System operations

Logically, Hamilton maintains a record of all unspent funds in existence and in order to spend funds, they must be present in the set of unspent funds. Our system supports three kinds of operations: Mint, Redeem, and Transfer, all of which are atomic and serialized.

Minting and redeeming. The Mint operation creates new unspent funds and adds these to the set of unspent funds, whereas the Redeem operation removes unspent funds from the system, making them unspendable. These operations also have semantics outside Hamilton: minting would normally correspond to funds in other forms of central bank money being set aside for use in Hamilton, whereas redeeming would make them available again.

Value transfers. The Transfer operation both consumes unspent funds and creates new unspent funds. This transaction is specified by a list of funds to be spent (inputs), a list of new funds to be created (outputs), and a list of witnesses (i.e., digital signatures) authorizing spends of each input. A successful Transfer completely consumes its inputs; these are removed from the system and cannot be used again, whereas the new outputs are available to be used as inputs to other Transfer or Redeem operations.

No editing of unspent funds. The set of unspent funds can only be modified via the above three operations, and funds tracked in the system cannot be modified to change their ownership or value.

Payment discovery. In public blockchains users can search the publicly available history of transactions to see if they have received payment. Transaction history in Hamilton is not public, and the sender must give the recipient the information about newly created unspent funds so that the recipient can further spend them. To ensure users know a Transfer is complete and has been applied, the transaction processor is also responsible for responding to user queries about the existence of unspent funds.

2.5 Security properties

In brief, the system must faithfully execute transactions, ensuring that each was authorized by the owner of the input funds, and safeguard that transactions do not disturb the overall balance of funds (outside of minting and redemption). Hamilton's transaction processor ensures this by satisfying the following four security properties.

Authorization. Hamilton only accepts and executes Mint and Redeem operations authorized by the issuer, i.e., only the issuer can mint and redeem funds. We use digital signature authorization for these. Similarly, we require that each Transfer transaction is signed by owners of all inputs the transaction attempts to spend.

Authenticity. The set of unspent funds tracked in Hamilton only contains *authentic* funds, as we now define. Define unspent funds created by authorized Mint operations to be *authentic*. Moreover, define unspent funds created by Transfer operations to also be authentic if and only if all inputs consumed by the transaction were authentic and the transaction preserves balance. Note that the recursive authenticity property depends on both the contents of the transaction itself, as well as the set of unspent funds when Transfer is applied.

Durability. Mint, Redeem, and Transfer are the only operations in Hamilton that change the set of unspent funds.

As a consequence of the three integrity properties defined above the set of unspent funds always remains authentic and transactions in Hamilton cannot be reverted. We further require that the transaction processor makes the following availability guarantee and always makes progress:

Availability. The transaction processor will always accept an authorized transaction spending authentic funds.

3 Transaction design and processing

This section first reviews Bitcoin's UTXO model in more detail and explains the challenges associated with using this data model in our setting. It then describes the UTXO hash set (UHS), a different idea that we choose as a basis for Hamilton's data model and the motivation behind our choice. Finally, it introduces Hamilton's transaction format, describes how to securely create and process transactions in this model, and discusses implications on future functionality.

3.1 Bitcoin's UTXO model

Bitcoin uses the UTXO model, where each output utxo has a value and an encumbrance: The value v is an integer multiple of the smallest subdivision of Bitcoin and an encumbrance is a *script*, an executable program which evaluates the conditions for a valid spend. An encumbrance expresses a predicate P taking two arguments: a transaction tx seeking to spend this utxo, and a witness wit. A transaction is a list of references to input UTXOs to be consumed together with a list of corresponding witnesses, and lists of values and encumbrances for new UTXOs to be created. Each encumbrance predicate returns true if and only if its corresponding witness signifies that this spending transaction should be authorized.

A common encumbrance is that of digital signature authorization, known as pay-to-pubkey (P2PK). Here the predicate P hard-codes a public key pk and $P(\mathsf{tx},\mathsf{wit})$ checks that wit consists of a valid signature under the public key pk where the message is the serialized spending transaction tx. To spend such a utxo, the user creates a transaction tx having the utxo as an input and signs tx with the corresponding secret key sk.

3.2 UTXO model challenges

Adopting Bitcoin's design in our setting comes with a number of challenges. First, Bitcoin's UTXO model requires maintaining a copy of the entire UTXO set in full detail. This has unwanted implications for privacy as Bitcoin node operators are both privy to values and encumbrances (i.e., users' public keys), as well as for data storage, especially when using complex or post-quantum secure encumbrances, which might be large. Second, in Bitcoin, transactions only refer to their inputs by specifying a hash txid of a prior transaction together with a particular output index idx of that transaction. Thus, to validate a transaction a node has to look up the input encumbrances and values in its local UTXO set, and only trivial validation checks can be done statelessly. This is reasonable when the UTXO set is small and nodes store it locally, but becomes more challenging when it must be partitioned across many machines to achieve higher performance, while still being accessed consistently.

3.3 UTXO hash set

A key observation in the design of Hamilton is that we can divide transaction validation checks into two parts – *transaction-local validation*, which does not require access to shared state, and *existence validation*, which does. We can then scale these two tasks independently. This is useful because they have different scalability profiles, with transaction-local validation requiring mostly compute resources (i.e., verifying digital signatures used in spend authorization) and existence validation requiring mostly persistent storage I/O.

By doing this, we can go even further and observe that after transaction-local validation, instead of processing and storing the entire UTXO, we can operate on cryptographic *commitments* to the UTXOs. In Hamilton we replace the UTXO set with a *UTXO hash set* (UHS), extending an idea first proposed as a Bitcoin storage and scalability improvement [41]. That is, our transaction processor stores unspent funds as a set of opaque 32-byte cryptographic hashes of UTXOs, not UTXOs themselves. We refer to hashes of UTXOs as UHS IDs or simply hashes. Instead of looking up the transaction input data (which we do not have), we ask the (untrusted) user to provide full input UTXOs in a transaction. However, a malicious user might lie and claim to have more funds to spend than they actually do. To catch this, we reduce the problem of checking UTXO correctness to *UHS commitment existence*—Do the funds the user is claiming they can spend actually exist? As we'll see in §4, this affords us the opportunity to piggyback existence validation with actual execution inside the distributed transaction commit protocol.

UTXOs must be stored in the user wallets and are supplied as part of transactions. We also note that while in Bitcoin the UTXO set is derived by processing the Bitcoin blockchain and keeping the set of unspent UTXOs, Hamilton's backend is a transactional database that maintains the UHS without operating a ledger.

Using a UHS has a number of benefits. First, as described above, a UHS-based transaction format lets us decouple transaction validation from funds existence checks and affords us opportunities for performance improvement in the backend. Second, it lowers storage requirements, as the transaction processor only stores a 32-byte hash per UTXO, independent of a UTXO's size. Third, it increases flexibility, as the UHS abstraction makes no assumptions about what hashes represent: it is easy to adapt a high-performance system maintaining a UHS, like Hamilton, to a different transaction formats or scripting languages without needing to change the core execution engine. Fourth, it improves privacy as the transaction processor does not store balances or account information.

However, a UHS design also presents some challenges, stemming from its data minimization. The UHS, as described above, does not contain enough information to audit the total amount of unspent funds (the "full" UTXOs only reside in user wallets). However, UHS hashes could be augmented to store a value alongside hashes (making supply auditing trivial, at some privacy cost), or by converting UHS IDs to homomorphic commitments that can be maintained and tallied using additional cryptographic techniques [54, 60]. The sender also has to provide the recipient with the UHS ID preimage to further spend their funds, as described in §3.7. Finally, decoupling transaction-local validation and access to shared state means that future transaction programmability is restricted to only referencing transaction-local data.

3.4 Transaction format

To build Hamilton we designed a new, extensible transaction format in the UHS model. As we will see later, Hamilton's transactions can be split into transaction-local validation and existence checks.

Unspent funds. We represent unspent funds as triples

$\mathsf{utxo} := (v, P, \mathsf{sn})$. Here v and P are value and encumbrance. Currently we only support encumbrances of public keys, and thus represent predicate P by the 32-byte public key pk itself. Our model supports future encumbrances, such as requiring a subset of signatures from multiple public keys.

The third component, sn is a globally unique *serial number*. The serial number, enables us to reference and distinguish funds that share the same encumbrance and value (e.g., Alice having received same \$5.00 value in two different transactions encumbered with the same public key $\mathsf{pk}_{\mathsf{Alice}}$); it also implies that UHS is a set, not a multiset. Our transaction format will ensure that serial numbers do not repeat across time: a serial number associated with a spent UTXO cannot "reappear" as a serial number for a new unspent UTXO. Global uniqueness of serial numbers is not a mere technicality: they express the intent of singling out a particular UTXO and prevent *replay attacks* (see §3.6 for discussion).

UTXO hash set. Instead of storing a set of entire UTXOs $\mathsf{utxo} = (v, P, \mathsf{sn})$, we store cryptographic commitments $h := \mathscr{H}(v, P, \mathsf{sn})$ to UTXOs. In Hamilton we set $\mathscr{H}$ to be SHA-256 to derive these hash commitments.

Mint **transactions.** New funds enter the system by system operator creating new utxo's and adding their hashes to the UHS. The issuer must choose unique serial numbers for newly minted UTXOs. It suffices to set these as uniformly random nonces.

Transfer **transactions.** A k-input, l-output Transfer transaction seeks to fully consume k UTXOs currently present in the system, and create l new UTXOs specified by encumbrances and values. Such a transaction $\mathsf{tx}_{\mathsf{Transfer}} = (\vec{\mathsf{utxo}}_{\mathsf{inp}}, \vec{v}_{\mathsf{out}}, \vec{P}_{\mathsf{out}}; \vec{\mathsf{wit}})$ is comprised of (a) a size-k list $\vec{\mathsf{utxo}}_{\mathsf{inp}}$ of input UTXOs to be spent; (b) two size-l lists $\vec{v}_{\mathsf{out}}$ and $\vec{P}_{\mathsf{out}}$ of output values and encumbrances specifying output UTXOs to be created; and (c) a size-k list of witnesses $\vec{\mathsf{wit}}$, one for each input.

Such $\mathsf{tx}_{\mathsf{Transfer}}$ creates l UTXOs with value/encumbrance pairs $(v_{\mathsf{out},i}, P_{\mathsf{out},i})$. We make UTXOs unique by deriving the serial numbers sn as pairs $\mathsf{sn} := (\mathsf{txid}, \mathsf{idx})$ as follows. The first component, txid is the unique transaction identifier, the cryptographic hash of the transaction that created this UTXO. This hash covers all input UTXOs, output encumbrances and values: $\mathsf{txid}(\mathsf{tx}_{\mathsf{Transfer}}) := \mathscr{H}((\vec{\mathsf{utxo}}_{\mathsf{inp}}, \vec{v}_{\mathsf{out}}, \vec{P}_{\mathsf{out}}))$. The second component, idx, is the particular output index, i.e., first, second, etc, output of the transaction.

Note that our transaction format includes input UTXOs in the Transfer itself. In contrast, a Bitcoin transaction does not: it references UTXOs via txid and idx instead, and requires UTXO look-ups in transaction processing.

Transaction creation. To create a Transfer transaction, users digitally sign txid with private keys corresponding to inputs they are spending. Each of these signatures serves as the witness authorizing the transaction to spend the given input. Witnesses are not included in the transaction identifier so signing can be deferred by the sender to after the transaction has been shared with the recipient. This is useful to support future smart contract functionality where unsigned transactions could be shared between parties to be signed and broadcast later under certain conditions. Recall that encumbrances are applied to individual outputs rather than whole transactions, and transactions can have multiple inputs, which means that funds can be spent atomically from multiple public keys in a single transaction. Once a transaction is finalized, the users will deterministically derive serial numbers of each of the output UTXOs from the transaction contents. Users store this outpoint information in their wallets.

3.5 Transaction execution

Processing a Transfer transaction involves confirming that it is valid and then applying it to the state. Validation involves checking the following:

1. *Syntactical correctness.* Check that the transaction has at least one input and output, and that the transaction supplies exactly one witness per input.
2. *Balance.* Check that transaction's input values tally up to exactly the same value as outputs to be created.
3. *Authorization.* Check that each input UTXO is accompanied by a valid signature, relative to the input's public key, on a message comprised of the transaction's identifier txid.
4. *Authenticity.* Check that transaction's input hashes exist in the UHS.

To apply a valid transaction to the UHS we atomically remove the spent input hashes and create the new output hashes under the control of the recipient(s); this in combination with the other checks provides durability.

Performing local-validation. Hamilton has dedicated components, which we call *sentinels*, that receive transactions from users and perform transaction-local validation. This local validation performs the above syntactical correctness, balance, and authorization checks.

Compaction. We further observe that while the transaction-local validation does not reference any data from the state and only uses transaction-local data, the UHS, in turn, does not reference a transaction's contents and only operates on the hash values. Thus, once locally validated, a transaction is *compacted*. First, the sentinel derives the output UTXO serial numbers; together with output encumbrances and values they fully specify output UTXOs to be created. Next, the sentinel hashes the input and output UTXOs and obtains two lists of hashes which we call a *compact transaction*. Finally, sentinels forward compact transactions to the execution engine, which maintains the UHS, for existence checks and to update the UHS state.

Checking existence, execution and the `swap` abstraction. Now, given a compact transaction, our system does the following: First, check if all input hashes exist in the UHS; if so (a) remove the input hashes from the UHS and (b) add the output hashes of newly created UTXOs to the UHS. We call this UHS primitive `swap`. Processing Hamilton transactions at scale reduces to the challenge of implementing a fast, scalable, and durable backend for executing `swap`, which we describe in §4. Such a backend abstraction maintains a set of hashes, and exposes `swap` as the only operation. The inputs to `swap` are two lists of hashes: one for existence checks and removal (called input hashes), and one for insertion (called output hashes). To execute a `swap`, the system atomically checks that all input hashes are present. If an input hash is missing, `swap` aborts. Otherwise, it obtains an updated set of hashes by erasing all input hashes and inserting all output hashes. All other hashes in the set remain unchanged.

We note that separating transaction-local validation and execution means that `swap` supports multiple transaction formats concurrently without affecting UHS performance.

3.6 Transaction format security

Using our transaction format Hamilton maintains an authentic, authorized and durable set of unspent funds (§2.5), eliminates the possibility of double spends, and also achieves additional security goals related to its use. In particular, transactions in Hamilton are not *replayable* and digital signature authorizations are not *reusable*.

These properties are a consequence of the fact that each UTXO created by a Mint or Transfer transaction is unique and guaranteed to not equal any other UTXO in the past or in the future, as we now explain. In Hamilton, each Transfer's UHS IDs are derived by hashing all the corresponding transaction's inputs, as well as details pertinent to the particular output itself (see §3.4). In particular, sn references previous unique serial numbers and recursively incorporates the entire transaction history up to distinct (due to presence of the uniformly random nonce) Mint's. Collision resistance of $\mathscr{H}$ guarantees that these serial numbers are unique. Because UHS hashes commit to the same UTXO data which must be provided in the transaction, an attacker can not fit a different UTXO preimage into the same UHS hash without violating the collision-resistance of $\mathscr{H}$.

No double-spends. Transfer operations permanently delete input hashes from the UHS. Therefore, as serial numbers are unique, no UHS ID can be spent more than once or recreated after having been spent.

No replay attacks. In a basic replay attack the victim has signed a single transaction to authorize a single value transfer. The attacker, however, submits this transaction twice in the hope of effectuating two value transfers. For example, Alice, who has two unspent \$5.00 "bills", might give Bob a transaction that spends one of her \$5.00 bills to pay for ice cream, which Bob then submits twice to take possession of both.

Hamilton's transaction format prevents replay attacks as each transaction references globally unique input hashes, and each signature covers the entire transaction, including all its inputs and outputs. Thus, signatures are not valid for spending any other UHS ID, including those created in the future, and it is not possible to copy a Hamilton transaction and apply it multiple times to spend additional funds.

Transactions are non-malleable. In a system with malleable transactions, an attacker can change some details about the transaction (e.g., the witnesses used to satisfy input encumbrances or output UTXO serial numbers) without otherwise changing the input UTXOs or modifying output UTXO values or encumbrances. For example, if the transaction format included an auxiliary field not covered by the signatures but used in serial number computation, an attacker could change this field. This would change output UTXO serial numbers and make it unsafe to accept a chain of unconfirmed transactions, thus preventing certain higher level protocols like the Lightning Network. In 2014, the largest Bitcoin exchange Mt. Gox closed after claiming to be a victim of malleability attacks [32]. In our implementation, we require signatures to cover all fields of uniquely-encoded transaction and derive UTXO serial numbers from the same fields (plus, output indexes).

3.7 Transaction protocol

Our choice of transaction data model and format directly impacts potential transaction protocols. For example, transaction compaction for the UHS adds a new communication step requirement between sender and recipient. The recipient should not consider a payment "complete" until they have received both a confirmation from the transaction processor *and* the full preimage data for their new outputs. If the recipient does not receive these, the sender has essentially destroyed the funds.

In theory, cryptocurrencies in which the recipient's address is obfuscated also have this problem. In practice, because the entire blockchain is public and standard address formats are used, recipients can scan *every* transaction to detect if they have been paid and, if so, construct new transactions to spend those funds. Even if the UHS were public, recipients would still not be able to unilaterally detect payments. The hash preimage for a UHS ID depends on data from the transaction that creates the UTXO which is unavailable to the recipient. As there is no public ledger, recipients must rely on the transaction processor to learn about the status of outstanding transactions.

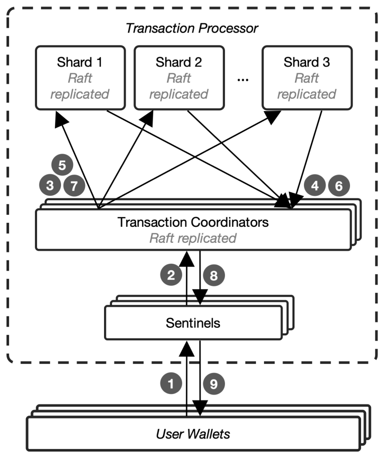

Figure 3: System diagram for the 2PC architecture and inter-component data flow

4 Processing transactions at scale

As described in §3, transaction processing can be split into transaction-local validation, existence validation, and execution, which creates opportunities for improving performance. To process more transactions, we *partition* the set of unspent funds across multiple computers. Transactions might reference unspent funds stored on different machines, requiring a coordination protocol to check existence of inputs and execute transactions atomically.

One way to achieve this is to first explicitly order all valid transactions and subsequently apply them to the partitioned state in the same order, if the inputs exist and have not already been spent. We investigated this architecture and found performance was limited by the ordering server (§4.3). However, we note that payment semantics do not *require* materializing a linear transaction history. Thus, we also built an architecture which executes transactions in parallel to achieve high performance, described next.

4.1 Applying transactions to the UHS

We use variants of two-phase commit and conservative two-phase locking[36, 44] to atomically apply transactions to the UHS, partitioned across *shards* which are each responsible for a subset of the UHS IDs which are unspent within the system.

Figure 3 shows a diagram of the components in the 2PC architecture and the data flow between components. As described in §3.5, a wallet submits a transaction to a sentinel (1), which validates everything *except* the existence of inputs. Upon success, sentinels convert transactions to compact transactions and send compact transactions to a *transaction coordinator* (2).

Each coordinator has a thread pool to execute transactions in parallel, and adds incoming compact transactions from sentinels to a queue. Once a thread from the pool becomes available, it drains the queue and creates a new distributed transaction (*dtxn*) containing the pending compact transactions (up to a maximum size). The thread then performs the 2PC protocol to commit the dtxn. If a thread is available, it will begin a new dtxn even if there is only one compact transaction waiting in the queue, it will not wait for the queue to grow. Due to application choices described in §3, we know the read/write sets ahead of time and can execute the dtxn entirely inside the 2PC protocol, without any extra roundtrips. This is the same technique introduced in Sinfonia [1].

There are three steps to commit a dtxn:

1. **Prepare (3).** The coordinator contacts each shard responsible for a UHS ID included in the dtxn and requests that it durably lock the input UHS IDs (a *prepare* request). (Note that by design of our transaction format (see §3.4), valid output UHS IDs are guaranteed to be unique across transactions, so reserving outputs is not necessary.) Each shard responds to the *prepare* request indicating which compact transactions in the dtxn had their IDs successfully locked, and which no longer exist, or were already locked by a different dtxn (4).
2. **Commit (5).** The coordinator uses the shards' responses to determine which compact transactions in the dtxn can be completed, and which cannot complete because some of the inputs are unavailable or already locked. The coordinator makes this decision durable and then contacts each shard again to indicate which compact transactions in the dtxn to complete and which to cancel (a *commit* request). Each shard then atomically unlocks the input UHS IDs belonging to a canceled transaction, deletes input UHS IDs and creates the output UHS IDs for successful transactions, and updates local dtxn state about the status of the dtxn. The shard then responds to the coordinator to indicate that the *commit* was successful (6).
3. **Discard (7).** The coordinator issues a *discard* to each shard informing them that the dtxn is now complete and it can forget the relevant dtxn state.

Once every shard participating in the dtxn has completed the *commit*, the coordinator informs each sentinel whether its transactions were successfully executed or rejected by the shards (8). The sentinels in turn forward these responses to the users who submitted the transactions (9).

It is possible that if two concurrent transactions by different transaction coordinators spend the same inputs, neither will succeed, because both will be canceled due to observing the other's lock conflicts. This means that at least one will need to be retried, which is left to the user's wallet. An adversary could try to continually conflict a user's transaction by spending the same input. However, this requires the adversary to have the authorization to spend the input in order to pass sentinel validation. Investigating methods to fairly resolve concurrency conflicts is left to future work.

Combining many compact transactions into a larger distributed transaction amortizes the costs of messaging and making the result of each phase of the protocol durable on each shard, whether by flushing to persistent storage or replicating as part of a distributed state machine. Because our application semantics are constrained, this is slightly different from traditional two-phase commit in that dtxns always complete successfully, and individual compact transactions are executed (or not) deterministically: if all of a compact transaction's input UHS IDs are locked and output UHS IDs are reserved, the compact transaction will succeed. The transaction coordinator always completes both phases of dtxns, even if some of the compact transactions within do not succeed. General 2PC designs need to support transaction coordinators that might make arbitrary decisions about whether to commit or abort transactions.

4.2 Fault Tolerance

Each transaction coordinator and shard is made fault tolerant using Raft [58], a distributed consensus algorithm. Sentinels only maintain state during the duration of the user wallet request to return transaction status to the user; if one fails, the user's wallet will need to retry its transaction or ask about its status with another sentinel.

Only the leader node in the transaction coordinator Raft cluster actively processes dtxns; followers simply replicate the inputs to each phase of the dtxn. This is a technique used in deterministic scalable database systems [68]. Before initiating each phase of the distributed transaction, the coordinator replicates the inputs to both the *prepare* and *commit* requests to each shard. Shards remember which phase each dtxn has last executed and the response to the coordinator. If the coordinator leader changes mid-dtxn, the new leader reads the list of active dtxns from the coordinator state machine and continues each dtxn from the start of its most recent phase. Shards that have already completed the requests will return the stored response to the new coordinator leader. To ensure proper completion of the *commit* across all shards, shards will remember the response for the *commit* until the coordinator has received responses from all shards in the dtxn and issued a *discard* to inform shards the dtxn is complete and can be forgotten. Note that these can be applied lazily and the transaction coordinators can inform the sentinels the transactions were successful before issuing *discard*.

Similar to coordinators, in each shard cluster only the leader processes dtxns and responds to sentinels. Although followers do not handle RPCs, they maintain the same UHS as the shard leader, so they are prepared to take over processing RPCs if the leader fails without a specific recovery procedure beyond that provided by Raft. Once a dtxn has entered the *prepare* phase and has been replicated by the coordinator cluster, the dtxn will always run to completion. If a shard leader fails mid-transaction, the coordinator leader will retry requests until a new shard leader processes and responds to the request.

4.3 Comparison to blockchain architectures

Many have suggested using blockchain technology to design a central bank digital currency. We found that using a blockchain-based system in its entirety was not a good match for our requirements. First, there was no requirement to distribute governance amongst a set of distrusting participants. The transaction processing platform is controlled and governed by a central administrator. Blockchains use relatively new distributed consensus protocols which are designed to operate in a permissionless, adversarial environment. This introduces software and operational complexity as well as new cryptographic assumptions. A CBDC should rely on the simplest, most well-understood, well-tested protocols to achieve its goals.

Second, we anticipated the complexity of a blockchain architecture would limit performance. To evaluate this we implemented a streamlined permissioned-blockchain-inspired design, the *atomizer*. Instead of using transaction coordinators and two-phase commit, the atomizer orders compact transactions into blocks through a replicated state machine. To reduce load on this ordering server, the design outsources storage of the UHS to shards, which hold the partitioned UHS ID state as in the 2PC design. However, a shard's UHS ID state is only correct up to a specific block height. Sentinels send compact transactions to shards, and shards then pass attestations for input IDs that exist to the atomizer, which collects complete compact transactions into blocks. The atomizer broadcasts confirmed blocks so shards can update their state, deleting spent UHS IDs and creating new ones. Shards do not require consensus or even primary/backup for correctness, they are merely replicated. For the specifics of the atomizer design see a related technical report [51]. As might be anticipated, we found the atomizer architecture's throughput is limited by the resource constraints (network bandwidth and CPU) of a single server, the atomizer leader, and cannot benefit beyond a limited point from additional shard resources (§6).

OmniLedger [49] is a sharded blockchain design which, like Hamilton, operates in the UTXO model, but uses a client-driven commit protocol to commit cross-shard transactions without going through a single server. Based on reported results, OmniLedger can achieve much higher performance than our atomizer prototype (with a replication factor of four per shard and 1% adversarial power, approximately 400K txns/sec). However, in Hamilton we did not want to rely on clients, which might fail or disappear, to complete transactions. Hamilton instead uses replicated transaction coordinators, so incomplete transactions will never result in stuck or frozen funds.

5 Implementation

We implemented Hamilton in C++17, released at [https://github.com/mit-dci/opencbdc-tx], and tested on Linux and macOS, though it should be portable to any UNIX-like system. The primary dependence on a UNIX-compatible API is our use of UNIX sockets for network communication. Clients communicate via a custom serialization protocol, via single, short-lived TCP connections.

We use LevelDB [43], NuRaft [35], libsecp256k1 [13] and vendored components from Bitcoin Core [12]. We use BIP-340 compatible Schnorr signatures [76] as our digital signature scheme. We also use the cryptography components of Bitcoin Core to provide optimized implementations of SHA256 [56], used as our cryptographic hash function, SipHash [4] used for hashmaps and bech32 [74, 75] used for error-correcting public key encoding.

6 Evaluation

Our evaluation answers the following questions:

- How does Hamilton's performance compare to other database and blockchain-based designs?
- How does Hamilton perform with multiple regional failures?
- How well does Hamilton tolerate different transactional workloads, whether with larger transactions or double spends?

Setup. Unless otherwise specified, for benchmarking we deployed on Amazon Web Services (AWS) using EC2 virtual servers running Ubuntu 20.04 (c5n.2xlarge instances with 8 vCPUs and 21GB RAM). We ran the system components across three regions within the United States: Virginia, Ohio and Oregon. Round-trip time between Virginia and Ohio was $\approx$ 12ms, Virginia to Oregon $\approx$ 62ms and Ohio to Oregon $\approx$ 51ms. Unless otherwise stated, there were 1B outputs, 8 logical shards, and shard and coordinator clusters were replicated by a factor of three. In particular, each shard and each coordinator has a Raft node in each of these three regions. Non-replicated components such as sentinels and load generators were distributed between regions to simulate load from across the United States. Load generators (c5n.large instances with 2 vCPUs and 5.25GB RAM) were simulated wallets that produced valid, signed transactions with two inputs and two outputs unless otherwise stated. We limited dtxns to a maximum size of 2000 compact transactions, and each coordinator had a thread pool containing 75 threads.

We do not consider data from a benchmark for a configuration if any Raft cluster was unable to reliably replicate

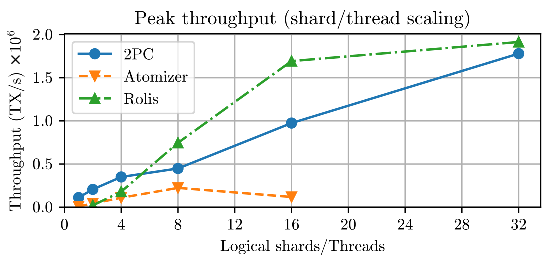

Figure 4: Compares the peak throughput of 2PC, the atomizer, and Rolis, when varying logical shard count (2PC and atomizer) or the number of threads (Rolis).

data between *all* regions during the experiment. Take, for example, a three node cluster: if one follower is reliably lagging behind due to data replication issues, though the leader and other follower still form a quorum, this configuration can't tolerate an additional failure of either node without potentially suffering a delay and throughput reduction. We took this to imply the system was oversaturated.

6.1 Comparison

Figure 4 shows the peak throughput for Hamilton and the atomizer when varying the number of load generators for different shard counts. Our 2PC architecture scales linearly as the number of logical shards increases, up to 1.7M txns/sec with less than one second 99% tail latency and under 500ms 50% latency, though we expect peak throughput would continue to increase with more shards and this would not negatively affect latency. Additionally, if a lower tail latency is desired for a particular transaction throughput, increasing the number of shards can decrease tail latency for the same offered load. This makes sense because, in the worst case, each transaction requires the participation of a subset of shards equal to the number of inputs and outputs in the transaction. Since transactions in the test load have an upper bounded number of inputs and outputs, increasing the number of shards results in each transaction requiring the participation of a smaller proportion of the total shards in the system.

The atomizer achieves 170K txns/sec with under two seconds 99% tail latency and 700ms 50% latency, the bottleneck being network bandwidth limitations between the replicas in different regions. In other experiments we found if bandwidth constraints are relaxed, computation in the lead atomizer replica to manage Raft replication and execute the state machine becomes the bottleneck.

We compare Hamilton's performance to three centralized databases: PostgreSQL, Rolis [64], and CockroachDB. In all cases the workload was the `swap` function with 2-in/2-out compact transactions; we did not run sentinels or do signature checking, which improved latency for these measurements.

PostgreSQL. We chose to compare against PostgreSQL since it is a widely used, fully-featured commercial database. We ran PostgreSQL on c5a.8xlarge EC2 instances, using benchmarking recommendations [50] and implementing `swap` as a stored function. We were able to obtain a max throughput on PostgreSQL v13.8 of 63.4K txns/sec and an average throughput over a 45 second run of 61K txns/sec, with average and 99% latency (as measured on the client) of 10ms, using hundreds of load generating clients. The database had 38.4M UHS IDs. Our experiments indicated that PostgreSQL is limited by throughput to the write-ahead log. Note that this was a single-machine benchmark with no replication.

Rolis. Rolis is a very high-performance in-memory research database. To evaluate Rolis we implemented the `swap` function in the benchmarking experiment software that accompanies Rolis. We used three replicas in the regions specified above and, as Rolis uses significant amount of RAM, used c5n.18xlarge instances (72 vCPUs, 192 GB RAM), largest among the c5n family of instances that we used to evaluate Hamilton. We did our experiments using Ubuntu 18.04.

We used a database containing 200M UHS IDs, unlike the 1B used for benchmarking Hamilton as 250M and larger data base sizes reliably ran out of memory. Figure 4 shows the performance as we increase the number of threads; the thread count there includes the additional thread [64, §6.1] that Rolis uses to advance the watermark and perform leader election tasks. To maximize Rolis's performance, we (a) implemented load generators inside each Rolis thread (so unlike when evaluating Hamilton, PostgreSQL, or CockroachDB, the load generators were not networked) and (b) used sequential UHS IDs. The latter avoids calling a hash function to generate UHS IDs in the critical path of the load generator, which halved the throughput when we tried it, but would be parallelizable in a different load generator implementation.

Rolis achieves a max throughput of 1.91M txns/sec on our workload. When replicating the YCSB++ benchmark on the same EC2 instances, the max throughput using 32 threads was 10.2M txns/sec, comparable to 10.3M reported in the Rolis paper. Our keys are 32 bytes instead of 8 bytes, and unlike YCSB++, which is 50% read-only, our workload is 100% read/modify/write, thus it requires more logic per transaction and more bandwidth for replication.

The almost-linear scalability when using 1, 2, 4, 8, and 16 threads did not continue when thread count increased from 16 to 31. We attribute this to hyper-threading: Rolis's implementation only supports a single socket (as it uses `rdtscp` counters for time-stamping), whereas our 72 vCPU instance had 16 cores (32 hyper-threads) per socket. This made our 31 thread benchmarks use hyper-threading and we would expect Rolis to perform better with an increased number of dedicated cores.

When comparing performance, it is important to note that Hamilton durably commits every transaction to disk, whereas Rolis runs wholly in-memory persisting nothing to disk. As our setting calls for ability to cold-start a system after potentially correlated failures, we see the excellent Rolis in-memory performance as establishing an upper, rather than a lower, bound for a backend.

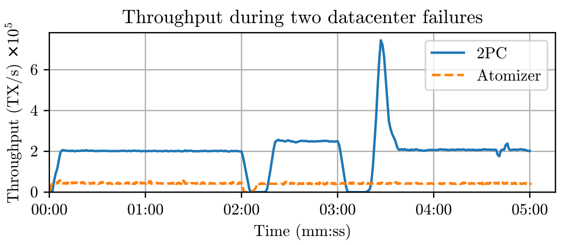

Figure 5: Throughput over time where the number of supported failures for both architectures was 2 and 2 whole data center failures were triggered at 120s and 180s. 5 sample moving average.

CockroachDB. We ran limited experiments against CockroachDB v22.1.9, a feature-complete scalable distributed database. CockroachDB automatically manages data partition assignments. We ran with a replication factor of three in the regions specified above, using c5d.4xlarge instances. We had to implement the `swap` function through client queries, requiring two roundtrips to commit a transaction, as CockroachDB does not yet support stored procedures or functions. It achieved throughput of 3.4K, 5.7K, and 11K txns/sec partitioning data across 1, 2, and 4 logical shards, respectively. CockroachDB is slower because it implements many more features than Hamilton, whereas implementing `swap` as a primitive in our distributed backend reduces the number of roundtrips and transferred data required to commit.

In summary, Hamilton achieves higher throughput than PostgreSQL and CockroachDB by co-designing the application with the data model, and pushing the `swap` primitive into the commit protocol. It approaches the performance of Rolis, a very fast in-memory replicated database. Rolis might be a better choice for a backend if in-memory replication is sufficient for durability; in our application it was important to have data on disk to have a path to recovery from simultaneous crashes. Another reasonable choice is to use an existing commercial database if performance is not as much of a concern.

6.2 Fault Tolerance

We evaluate how our system handles up to two regional data center failures, and its scalability as the number of supported faults increases.

Figure 5 shows the overall system throughput over time when shards and coordinators have a replication factor of five (supporting up to two failures per cluster). To test continued system availability when up to two data centers fail

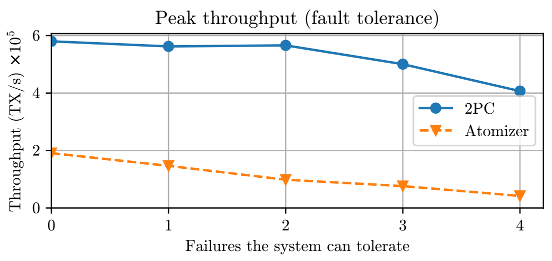

Figure 6: Peak throughput versus the number of tolerated failures per replicated service. Atomizer used 100M UHS IDs, 2PC used 1B.

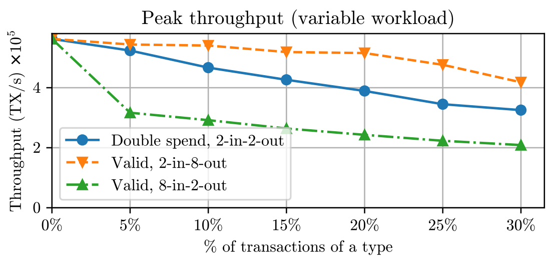

Figure 7: Peak throughput varying the proportion of valid, 2-in-2-out transactions.

completely, the Raft leaders for coordinators and shards were killed at 120 seconds into the test, and a subsequent set of nodes for each cluster were killed at 180 seconds into the test. The second group of nodes contained a mixture of leaders and followers depending on the result of the leader election from the previous failure. The plot shows that our system is successfully able to recover from the failure of two entire data centers with minimal downtime and no loss of system performance. For each failure, throughput was temporarily reduced for less than fifteen seconds, before automatically recovering to the baseline. There is no data loss and the system is not left in an inconsistent state as the replacement coordinators complete any distributed transactions that were in progress at the time of each failure.

Figure 6 compares how peak throughput is affected by the number of supported system failures between architectures by increasing the number of tolerated failures from 0 through 4. The plot shows that as replication factor increases, peak throughput for a given system configuration decreases. Since the performance of our Raft replicated services is limited by bandwidth constraints between the leader and follower nodes, more replicas require more leader bandwidth to provide the same throughput. In 2PC, we believe a higher replication factor can be supported without a loss in performance by increasing the number of shard and coordinator clusters. The atomizer architecture could not scale in this way, as the system throughput is limited by a single Raft service which provides the global order of transactions.

6.3 Workload Variability

We varied the proportion of transactions with a high number of inputs and outputs, and the proportion of double-spending transactions to see how Hamilton performs under different workloads from users.

Figure 7 shows how the proportion of double-spending transactions, or those with a large number of inputs and outputs affects peak throughput. In this test, the proportion of invalid or non-2-in-2-out transactions in the workload was varied from 0% through 30%. The load generators sent double-spending transactions by storing previously confirmed transactions and re-issuing them at a later time. We only plot the throughput of valid transactions because double-spending transactions never complete.

As the proportion of large or double-spending transactions increases, the peak throughput decreases. This behavior is similar to increasing the overall number of 2-in-2-out transactions. The system is limited by the overall number of UHS IDs being processed, regardless of how they are grouped into transactions. Shards and coordinators replicate all transactions, so double-spends exert the same load as valid transactions. Thus, increasing the number of shards and coordinators could absorb an increased proportion of large or double-spending transactions while executing the same number of valid transactions. Transactions with a large number of inputs most negatively affect peak throughput because the sentinels have to validate more signatures per transaction. This could be solved by increasing the number of sentinels per load generator.

7 Related Work

Central banks are experimenting with or launching CBDCs. Some [22, 23, 34, 65] use DLT [57, 63], but according to their reports do not achieve as high performance as Hamilton. Other CBDC work uses a parallelized architecture; China's e-CNY is currently in public trials [24, 46, 62] and is a scalable system based on the UTXO model, but does not support self-custody. The Eurosystem has tested a CBDC design based on tracking groups of bills using a set of parallelized blockchains [38]. While it achieves linear scalability, transactions involving multiple bills require external coordination. The Regulated Liability Network [29] presents a design which claims to achieve 1M transactions per second with multiple coordinated blockchains. However, they do not discuss deployment across multiple geographic regions which is vital for resiliency or provide latency measurements.

Chaumian eCash [26] and designs based on it [18, 20, 21] operate with a central trusted intermediary, but either require maintaining an ever-growing list of all spent coins for double spend prevention, or require users to manage expiring coins, which has significant policy implications. The Swiss National Bank [27] expands upon the Chaumian ecash model using this technique.

Several central banks already support real-time gross settlement (RTGS) and fast payment systems [5]. These systems are designed to settle transactions between only eligible financial institutions. In practice, these systems do not handle a volume of traffic representative of a national retail payment system, do not provide programmability, nor provide access to the general public [6, 39, 40, 59].

Contrary to decentralized cryptocurrencies [53, 73] or stablecoins [11, 25, 33, 67], Hamilton is designed to be administered directly by the central bank or a related entity, and transacts in central bank liabilities. However, Hamilton borrows ideas from cryptocurrency designs; it uses the UTXO transaction model, but only stores 32-byte hashes [41]. Users cannot verify transaction execution themselves since they cannot access the ledger or state of the system. Techniques like authenticated datastructures [66] or cryptographic proofs of transaction inclusion [55] might be able to help with this.

Newer consensus algorithms [30] achieve higher throughput for agreement on ordering, and Hamilton could benefit from these as a replacement for Raft. However, faster consensus does not address the scalability bottleneck of state machine execution and validation in non-sharded blockchain-based architectures. Our 2PC architecture outperforms both our straw-man sharded blockchain as well as existing sharded blockchains which aim to operate in a decentralized setting, and thus cannot rely on a trusted coordinator to drive the cross-shard commit protocol to completion [2, 31, 49, 77]. However, some of these blockchains provide more features than Hamilton, like general smart contracts.

Via careful choices in application transaction design and format, Hamilton is able to avoid the need for reads or any other transaction execution before commit time, and can apply good ideas in traditional distributed transaction commit protocols [1, 28, 68] in a simplified backend that does not need to handle general transactions.

8 Conclusion

This work presents a high-performance, resilient transaction processor for CBDCs. We support a range of potential policy choices and can minimize data stored in the transaction processor while supporting a variety of custodial models. Our experiments show that a blockchain-based design for CBDC has seriously scalability limitations, but by validating transactions in parallel we can achieve millions of transactions per second.

9 Acknowledgements

The authors express gratitude to Robert Bench and Jim Cunha for their leadership and direction in this work. We are also grateful to Tyler Frederick and David Urness who were instrumental in the early stages of this research. In addition, we thank Chris Berube, Alexander Chernyakhovsky, Nikhil George, Gert-Jaap Glasbergen, Alistair Hughes, Ben Kincaid, Weihai Shen, Bernard Snowden, Sam Stuewe, and our shepherd and reviewers for their helpful contributions, feedback, and comments. Cory Fields, Neha Narula, and Madars Virza were supported by the funders of the Digital Currency Initiative.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):159–174, 2007.

[2] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[3] R. Auer, J. Frost, M. Lee, A. Martin, and N. Narula. Why central bank digital currencies? Liberty Street Economics, 2021. https://libertystreeteconomics.newyorkfed.org/2021/12/why-central-bank-digital-currencies/.

[4] J. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. Cryptology ePrint Archive, Report 2012/351, 2012. https://eprint.iacr.org/2012/351.

[5] Bank For International Settlements. Fast payments - enhancing the speed and availability of retail payments. Committee on Payments and Market Infrastructures, 2016. https://www.bis.org/cpmi/publ/d154.pdf.

[6] Bank for International Settlements. BIS statistics explorer, 2019. https://stats.bis.org/statx/toc/CPMI.html.

[7] Bank of Canada et al. Central bank digital currencies: foundational principles and core features. BIS Working Group, 2020. https://www.bis.org/publ/othp33.pdf.

[8] Bank of England. Central bank digital currency: Opportunities, challenges and design, 2020. https://www.bankofengland.co.uk/-/media/boe/files/paper/2020/central-bank-digital-currency-opportunities-challenges-and-design.pdf.

[9] M. L. Bech and R. Garratt. Central bank digital currencies. BIS Quarterly Review, September 2017.

[10] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014.

[11] Binance. Binance USD. https://www.binance.com/en/busd.

[12] Bitcoin Core Developers. Bitcoin Core. https://github.com/bitcoin/bitcoin.

[13] Bitcoin Core Developers. libsecp256k1. https://github.com/bitcoin-core/secp256k1.

[14] C. Boar and A. Wehrli. Ready, steady, go? – results of the third BIS survey on central bank digital currency. *BIS Papers No 114*, 2021. https://www.bis.org/publ/bppdf/bispap114.htm.

[15] Board of Governors of the Federal Reserve System. Money and payments: The U.S. dollar in the age of digital transformation, January 2022.

[16] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. Zexe: Enabling decentralized private computation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, S&P '20, 2020. ePrint: https://eprint.iacr.org/2018/962.

[17] L. Brainard. Update on digital currencies, stablecoins, and the challenges ahead, 2019. https://www.federalreserve.gov/newsevents/speech/brainard20191218a.htm.

[18] S. Brands. Untraceable off-line cash in wallet with observers. In *Annual international cryptology conference*, pages 302–318. Springer, 1993.

[19] N. Brewster and S. Bishop. Getting out the message. http://www.centralbank.org.bb/_economic-insightbb/getting-out-the-message.

[20] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 302–321. Springer, 2005.

[21] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Balancing accountability and privacy using e-cash. In *International Conference on Security and Cryptography for Networks*, pages 141–155. Springer, 2006.

[22] Central Bank of Nigeria. Design paper for the eNaira. https://enaira.gov.ng/download/eNaira_Design_Paper.pdf.

[23] Central Bank of The Bahamas. Sand dollar. https://www.sanddollar.bs.

[24] Central Banking Newsdesk, 2020. https://www.centralbanking.com/fintech/cbdc/7529621/pboc-confirms-digital-currency-pilot.

[25] Centre Foundation. USD-C. https://www.centre.io/usdc.

[26] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of Crypto 82*, pages 199–203. Springer, 1983.

[27] D. Chaum, C. Grothoff, and T. Moser. How to issue a central bank digital currency. *arXiv preprint arXiv:2103.00254*, 2021.

[28] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[29] A. Culligan, N. Pennington, M. Delatine, P. Morel, E. M. Salinas, G. Vargas, N. Dusane, J. Iu, S. Sheikh, N. Kerigan, T. McLaughlin, P. D. Courcy, M. Low, and K. H. Park. The regulated liability network, 12 2021. https://setldevelopmentltd.box.com/shared/static/18mff2m990qabgzseiex3h7itq7qdnls.pdf.

[30] G. Danezis, E. K. Kogias, A. Sonnino, and A. Spiegelman. Narwal and Tusk: A DAG-based mempool and efficient BFT consensus, 2021. https://arxiv.org/pdf/2105.11827.pdf.

[31] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[32] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and MtGox. In *Proceedings of the 19th European Symposium on Research in Computer Security*, pages 313–326, 2014.

[33] Diem Foundation. Diem. https://www.diem.com/en-us/white-paper/.

[34] Eastern Caribbean Central Bank. ECCB digital EC currency pilot, 2021. https://www.eccb-centralbank.org/p/about-the-project.

[35] eBay. NuRaft. https://github.com/eBay/NuRaft.

[36] K. Eswaran, J. Gray, and L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, november 1976.

[37] European Central Bank. ECB publishes the results of the public consultation on a digital euro, 2021. https://www.ecb.europa.eu/press/pr/date/2021/html/ecb.pr210414~ca3013c852.en.html.

[38] European Central Bank. Work stream 3: A new solution – blockchain & eID, 2021. https://haldus.eestipank.ee/sites/default/files/2021-07/Work%20stream%203%20-%20A%20New%20Solution%20-%20Blockchain%20and%20eID_1.pdf.

[39] Eurosystem. TARGET Instant Payments Settlement user requirements, 2017. https://www.ecb.europa.eu/paym/target/tips/profuse/shared/pdf/tips_crdm_uhb_v1.0.0.pdf.

[40] Eurosystem. T2-T2S consolidation user requirements document for T2-RTGS component, 2018. https://www.ecb.europa.eu/paym/pdf/consultations/T2-T2S_Consolidation_User_Requirements_Document_T2_RTGS_v1.2_CLEAN.pdf.

[41] C. Fields. UHS: Full-node security without maintaining a full UTXO set. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-May/015967.html.

[42] R. Garratt, M. J. Lee, et al. Monetizing privacy with central bank digital currencies. Technical report, Federal Reserve Bank of New York, 2020.

[43] Google. LevelDB. https://github.com/google/leveldb.

[44] J. N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, pages 394–481. Springer, 1978.

[45] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specifiation, 2021. https://zips.z.cash/protocol/protocol.pdf.

[46] J. C. Jiang and K. Lucero. Background and implications of China's central bank digital currency: E-CNY. *Available at SSRN 3774479*, 2021.

[47] J. Kiff, J. Alwazir, S. Davidovic, A. Farias, A. Khan, T. Khiaonarong, M. Malaika, H. Monroe, N. Sugimoto, H. Tourpe, and P. Zhou. A survey of research on retail central bank digital currency, 2020. https://www.elibrary.imf.org/view/journals/001/2020/104/001.2020.issue-104-en.xml.

[48] koe, K. M. Alonso, and S. Noether. Zero to Monero: Second edition, 2020. https://www.getmonero.org/library/Zero-to-Monero-2-0-0.pdf.

[49] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[50] V. Kumar. Pgbench: Performance benchmark of postgresql 12 and edb advanced server 12, 2020. https://www.enterprisedb.com/blog/pgbench-performance-benchmark-postgresql-12-and-edb-advanced-server-12.

[51] J. Lovejoy, C. Fields, M. Virza, T. Frederick, D. Urness, K. Karwaski, A. Brownworth, and N. Narula. A high performance payment processing system designed for central bank digital currencies. *Cryptology ePrint Archive*, 2022.

[52] G. Maxwell. Confidential transactions – investigation. https://elementsproject.org/features/confidential-transactions/investigation.

[53] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 10 2008. https://bitcoin.org/bitcoin.pdf.

[54] N. Narula, W. Vasquez, and M. Virza. zkLedger: Privacy-preserving auditing for distributed ledgers. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, 2018. ePrint: https://eprint.iacr.org/2018/241.

[55] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford. CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security '17)*, pages 1271–1287, 2017.

[56] NIST. Secure Hash Standard, 2002. https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf.

[57] NZIA. Nzia cortex dlt. https://nzia.io.

[58] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 305–319, 2014.

[59] Pay.UK. Pay.UK 2020 annual self-assessment against the principles for financial market infrastructure, 2020. https://www.wearepay.uk/wp-content/uploads/Pay.UK-PFMI-Self-Assessment-Jun-20.pdf.

[60] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference*, CRYPTO '91, pages 129–140, 1992.

[61] A. Pertsev, R. Semenov, and R. Storm. Tornado cash privacy solution: Version 1.4, 2019. https://tornado.cash/Tornado.cash_whitepaper_v1.4.pdf.

[62] Y. Qian. Technical aspects of CBDC in a two-tiered system, 2018. https://www.itu.int/en/ITU-T/Workshops-and-Seminars/20180718/Documents/Yao%20Qian.pdf.

[63] R3. Corda. https://www.corda.net.

[64] W. Shen, A. Khanna, S. Angel, S. Sen, and S. Mu. Rolis: a software approach to efficiently replicating multi-core transactions. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 69–84, 2022.

[65] Sveriges Riksbank. E-krona pilot phase 1. *Sveriges Riksbank Report*, 2021. https://www.riksbank.se/globalassets/media/rapporter/e-krona/2021/e-krona-pilot-phase-1.pdf.

[66] R. Tamassia. Authenticated data structures. In *European symposium on algorithms*, pages 2–5. Springer, 2003.

[67] Tether Operations Ltd. Tether. https://tether.to/.

[68] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[69] UkoeHB. Mechanics of MobileCoin. https://github.com/UkoeHB/Mechanics-of-MobileCoin.

[70] A. Usher, E. Reshidi, F. Rivadeneyra, S. Hendry, et al. The positive case for a CBDC. *Bank of Canada Staff Discussion Paper*, 2021.

[71] N. van Saberhagen. CryptoNote v 2.0. https://web.archive.org/web/20201028121818/https://cryptonote.org/whitepaper.pdf.

[72] T. Walton-Pocock. Why hashes dominate in SNARKs: A primer by AZTEC, 2019. https://medium.com/aztec-protocol/why-hashes-dominate-in-snarks-b20a555f074c.

[73] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[74] P. Wuille. Bech32m format for v1+ witness addresses, 2020. https://github.com/bitcoin/bips/blob/master/bip-0350.mediawiki.

[75] P. Wuille and G. Maxwell. Base32 address format for native v0-16 witness outputs, 2017. https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki.

[76] P. Wuille, J. Nick, and T. Ruffing. Schnorr signatures for secp256k1, 2020. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki.

[77] M. Zamani, M. Movahedi, and M. Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.