A High Performance Payment Processing System Designed for Central Bank Digital Currencies

James Lovejoy Cory Fields Madars Virza Tyler Frederick David Urness
Kevin Karwaski Anders Brownworth Neha Narula

1 Introduction

Central banks are increasingly investigating general-purpose central bank digital currency (CBDC), defined as a currency that is electronic, a liability of the central bank denoted in the national unit of account, broadly available, and used for retail and person-to-person payments [10, 11, 19, 20, 24, 29, 30, 45, 62, 81]. Figure 1 summarizes a figure explaining the different properties of a CBDC as compared to other forms of payment instruments [13]. Researchers have proposed that 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 a privacy-preserving digital payment method [4, 10, 20, 52, 84].

A CBDC's primary use case is to act as a payment instrument for individuals and businesses as part of a broader exchange of goods or services. For example, a user might pay for coffee in a cafe by sending digital currency to the cafe owner. However, beyond this core use case, the design of a CBDC can vary considerably based upon the public policy objectives and unique characteristics of various jurisdictions. Importantly, the feasibility, operating performance and impact of different CBDC design choices are inextricably linked to the technical design of the underlying transaction processor. To better inform policy discussions, central banks are recognizing the importance of technical experimentation in understanding the implications and tradeoffs of different CBDC models and design decisions on possible policy outcomes.

The Federal Reserve Bank of Boston (Boston Fed) and the Massachusetts Institute of Technology's Digital Currency Initiative (MIT DCI) are collaborating on a multi-year exploratory research project, known as Project Hamilton, to gain a hands-on understanding of a CBDC's technical challenges and opportunities.[1] This paper presents the first phase of Project Hamilton's research and describes the technical design of Hamilton, a research transaction processing system flexible enough to support experimentation with multiple CBDC models. Hamilton is the first contribution to OpenCBDC, a place for collaboration on technical research and development for CBDC.[2]

1.1 Goals

Project Hamilton's Phase 1 goal is to investigate the technical feasibility of a high throughput, low latency, and resilient transaction processor that provides flexibility for a range of eventual CBDC design choices. We intend to investigate more complex functionality in future phases. Note that our design is not a complete CBDC system; it is neither production-ready nor does it provide all the functionality needed for a working CBDC. In this technical paper, we do not assess a CBDC's policy, regulatory, and legal questions or whether or how it could be issued.

Performance. To support the scale of retail transactions in a large country such as the US, a CBDC transaction processor should be able to process, at minimum, tens of thousands of transactions per second in real-time and scale to account for the potential growth in payment volumes [54]. These figures well exceed the transaction volumes interbank settlement systems are designed to process [7, 47, 48, 74]. We set the following initial set of performance targets to guide our design:

Speed. To capture the benefits of faster or real-time payments [6], we set a target of 99% of transactions completing within 5 seconds. Completion includes a transaction being validated, executed, and confirmed back to users. This is comparable to card payment methods and existing interbank instant payment systems.

Throughput and scalability. To support settlement finality and CBDC models which don't require intermediaries to aggregate transactions, Hamilton must be able to handle peak projected transaction volumes produced by hundreds of millions of users. We chose 100,000 transactions per second as a minimum target based on existing cash and card volumes and expected growth rates.

Resiliency. To maintain trust in the digital currency, a CBDC must guarantee the ongoing existence and usability of funds. In this phase of research, we focus on continuing to provide system access and preventing data loss even in the presence of multiple data center failures.

[1] This project is named in tribute to two Hamiltons: Margaret, an MIT computer scientist who led the software development for the Apollo Program's guidance system at NASA, and Alexander, who laid the foundation for a U.S. central bank.

[2] https://github.com/mit-dci/opencbdc-tx

Property	Cash	Bank deposits	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 [13].

We will address upgradeability and other measures of resilience in future phases of research.

Privacy and minimizing data retention. There is strong user demand for financial privacy since fine-grained transaction data can reveal sensitive user details [59], even if anonymized [50]. Respondents to a Eurosystem CBDC public consultation ranked privacy as the most important feature of a digital euro (46% of respondents) [45]. Any payment system's architecture is influenced by the design choices made around data privacy, access, and retention, and achieving robust privacy requires making explicit architectural choices at each layer of a system's design. In particular, if many parts of a system require access to sensitive data (either raw or derived), it can be challenging to retrofit such a system to provide data protection after the fact. Though exploring the implications of cryptographic designs for strong privacy will be a part of our Phase 2 research, during Phase 1 we focused solely on design options that limit data access and retention in the central transaction processor, to support future research and design optionality. Note that the safest way to secure data is not to collect it in the first place. We designed Hamilton's transaction processor to retain very little data about transactions.

Intermediary and custody flexibility. One of the most important questions in CBDC design is that of the role of the central bank and other intermediaries.[3] These roles will likely vary by jurisdiction, due to policymaker decisions and consumer preferences.

Currently, members of the public 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. These institutions 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 power to spend their own funds without the need for an account provider or custodian to generate transactions [22].

[3]We use the term "intermediaries" to include financial institutions, custodians, payment service providers, and other third parties who perform payment-related functions and services. Other entities which do not perform payment-related functions, such as Internet service providers, are not included in this definition.

The Bank for International Settlements (BIS) simplifies intermediary choices to three possibilities—the "direct" model, in which the central bank issues CBDC to users directly, "two-tier", in which the central bank issues CBDC to intermediaries who then manage relationships with users, and a hybrid of the two [8].

We do not directly address intermediary roles in Phase 1. However, we foresee much more complexity of choice in the roles for intermediaries in a CBDC, along dimensions like authorization, custody, and viewing transactions.

Importantly, our work shows the design space for intermediaries is much broader than previously assumed.

Design choices not addressed in Phase 1. Fees, compliance and fraud controls, and several other design considerations were not addressed in Phase 1 and are left to future work.

1.2 System design

Our system processes payments from users who address and sign transactions using their public/private key pairs stored in their digital *wallets*, as is the case in many cryptocurrencies.

User 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). A transaction indicates the unspent funds being used and the new unspent funds being generated (i.e., the new data record indicating who now has ownership over the money). We refer to these as transaction *inputs* and *outputs*, respectively, consistent with many cryptocurrency systems [14, 32, 67]. Hamilton validates the transaction is correct and executes it by deleting the inputs and creating the outputs. We implement two architectures for high throughput, low latency, and fault-tolerant transaction processing. The first, the *atomizer* architecture, uses an ordering server to create a linear history of all transactions. The second, the *two-phase commit* (2PC) architecture, executes non-conflicting transactions (transactions which do not spend or receive the same funds) in parallel and does not create a single, ordered history of transactions.

1.3 Technical challenges and contributions

We had to solve 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 can support a range of intermediary roles including one where users custody their own funds. We explore a design which minimizes storing personally identifying user data and information about transaction addresses and 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. In payment applications, all transactions require strong consistency; it is vital that payments execute correctly even in the presence of unforeseen events or computer crashes. Given our performance and resiliency requirements, we must store data on multiple computers. This requires correctly coordinating data updates across computers for most transactions, since we cannot rely on payments having data locality, which is often exploited by traditional database systems for partitioning to make workloads predominantly single-partition transactions. We decided to support atomic transactions, meaning a payment is guaranteed to execute in an all-or-nothing fashion. Atomicity provides better semantics for payments and guarantees to users, and is helpful for programmability in the future, but increases the cost of achieving these requirements. It remains to be seen if it will be required for a CBDC.

Hamilton addresses these challenges using three key ideas:

The first is to decouple transaction validation from fund existence checks; only a *validating layer* needs to see the details of a transaction. Beyond the validating layer, Hamilton stores funds as opaque 32 byte hashes inside an *Unspent funds Hash Set*, or UHS [49] (§3.2). This hides details about the funds (like amounts and addresses) from the UHS storage, reduces storage requirements, and creates opportunities to improve performance.

Our second key idea is the UHS-designed transaction format (§3.3), 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 heavily from Bitcoin's transaction format but is designed to be validated without looking up data from the UHS, which we term *transaction-local validation*.

The UHS design, in combination with our transaction format, affords us substantial flexibility. First, 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 [14, 66, 83, 85], which, along with auditability, we intend to explore in Phase 2. Second, 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. Third, if needed, it is always possible to store more data at other layers outside the transaction processor, for example in user wallets or an intermediary such as a custodian. 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 (§3.4).

Our third key idea is a system design and protocol for efficiently committing atomic payment transactions that leverage the UHS to achieve high performance, strong consistency, and geographically-replicated fault tolerance in a 100% read/write, non-partitionable workload. We implemented two high-performance architectures with different properties (§4). In both architectures, the UHS is partitioned across servers to support higher throughput and an expanding UHS; executing a single transaction often involves multiple servers. Each architecture uses a different technique to coordinate the consistent application of a transaction across servers. In the atomizer architecture, we use a replicated server to order all updates, which are then applied to the state of the rest of the system; one can think of this as an attempt at a high-performance blockchain.

In the 2PC architecture, we exploit payment transaction semantics and our transaction format to limit the locking required to achieve atomic transactions and serializability [15]. Transactions using different funds do not conflict and can execute in parallel; once a valid transaction's funds are confirmed to be unspent, the transaction can always proceed, and we can batch many transactions together to amortize two-phase commit overhead. Because of these choices, we can use a simpler version of two-phase commit without rollback.

Our evaluation demonstrates 1.7M transactions per second in the 2PC architecture with less than one second 99% tail latency, under 0.5 seconds 50% latency, and adding more resources could increase throughput further without negatively affecting latency. The atomizer design peaks at 170K transactions per second with under two seconds 99% tail latency and 0.7s 50% latency. We reduced the functionality in the atomizer state machine to simply ordering and deduplicating the inputs for a small set of transactions; even so, we were limited in throughput because the atomizer could not be sharded across multiple servers. This implies that a design which

requires strongly ordering valid transactions to prevent double spends will be throughput-limited.

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
- Two architectures to implement Hamilton: the atomizer architecture which provides a globally ordered history of transactions but is limited in throughput, and the 2PC architecture that scales peak throughput almost linearly with resources but does not provide a globally ordered list of transactions.
- An evaluation of the performance of the two architectures with different types of transaction workloads. Hamilton and the software to evaluate its performance are implemented in OpenCBDC-tx.

Our architectures are for research purposes and, accordingly, have limitations that need to be addressed in future work. These experimental designs are not ready for real-world use and do not provide system-wide auditability, protection against internally compromised machines, complete privacy guarantees, or resilience to denial of service attacks.

The rest of this paper discusses the system model and security goals for Hamilton (§2), explains the transaction format and UHS (§3), describes the design of the two architectures (§4) and their implementation (§5), evaluates Hamilton's performance on a variety of transaction workloads (§6), and puts Hamilton in context with related work (§7). We discuss broader learnings, limitations to our design, and future work in (§8).

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 the assistance of an intermediary. We note that 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*. At a high level they operate and interact as follows. The transaction processor

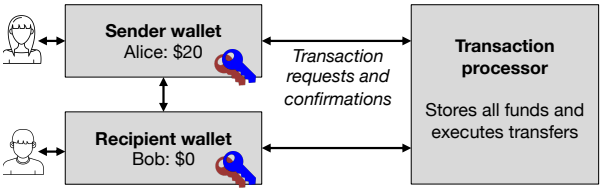


Figure 2: Data flows between all participants in a transaction.

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. We leave offline transactions and transfers without Internet connectivity to future research. Figure 2 shows the high-level system model and potential communication channels between users and the transaction processor.

Users run *wallet* software to manage cryptographic keys, track funds, and facilitate transactions. Wallets could run on a mobile phone or specialized hardware in smart cards. We do not discuss how users obtain wallets and get system access; this could be done using a PKI and access control, or, the system could be open to all users. An important piece of future work is preventing spam and denial of service attacks, which we briefly discuss in §8.

2.2 Threat model

Our goal is to design a system where each user's funds and the integrity of the monetary system are safe from interference of an external actor. For the purposes of this paper 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 channels 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. Some of 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: prior work

To design a transaction processor we have to make a choice about how the users' funds are represented in the system. The two most common ways are the account balance model and the UTXO model, which we now summarize.[4]

Tracking of balances. The simplest way to implement a payment system is using balances. The system can store unspent funds as balances associated with unique identifiers, and a user can make a payment by issuing a request to the transaction processor to transfer balance to another identifier. Traditional payment systems choose this approach and manage authorization by storing identifiers under *user accounts*, usually accessed via a username and password. Traditional payment systems could use public key cryptography and digital signatures instead of passwords for authorization, but this is not widely used in practice outside of cryptocurrency.[5] Several cryptocurrencies, like Ethereum [87], choose this data representation.

Tracking of discrete funds. Another way to implement a payment system is to track outstanding funds without explicitly consolidating them into balances. Here a system maintains an append-only ledger of accounting entries (sometimes called "coins") each of which records a value (i.e., amount of dollars) and conditions to spend the funds. Furthermore, each entry is either marked as "spent" or "unspent". To transfer funds, a user creates and authorizes a transaction which: (a) marks some entries (called *inputs*) as spent, each with a *witness* that satisfies the conditions to spend the entry; and (b) appends new (unspent) entries (called *outputs*) to the ledger. A valid transaction must preserve balance: the sum of a transaction's input values must equal the sum of its output values.[6]

[4]There are other, less common, representation models, such as, David Chaum's original eCash design [33] and the ECB prototype [46] using fixed value bills that atomically change ownership.

[5]Public-private key pairs have significant advantages over usernames and passwords. Private keys are harder to guess or crack with brute force and can be reused without the same risks as passwords (a common security problem). Furthermore, private keys do not need to be seen or stored by the central transaction processor; signatures made with a private key only authorize a single transaction instead of providing permanent access to a user's money. They also allow for interoperability with other public-private key systems and for novel privacy options.

[6]In cryptocurrencies with fees, the requirement is that the sum of the transaction's input values must be greater than the sum of its output values, with the difference going to the block miner as fees.

Tracking of unspent entries is central to this model so, following Bitcoin, these have a special name: *UTXOs* (Unspent Transaction Outputs). Importantly, UTXOs are never modified and must be spent in their entirety. Therefore, Alice who wants to use her $20.00 UTXO to send $4.99 to Bob will create a transaction with two outputs: one $4.99 output meant for Bob and one $15.01 *change output* meant for Alice herself. In contrast to physical banknotes or coins the UTXO values are not restricted to a fixed set of denominations. Note that it is not required to make change in a system that tracks balances since the default is that the remaining balance stays under the same identifier.

2.4 Data representation in Hamilton

Both of these designs have benefits and drawbacks, but we chose to build Hamilton in the UTXO model. The choice of UTXOs is compatible with privacy extensions in the future. Notably, most scalable privacy designs [14, 21, 33, 60, 63, 83, 85], including those deployed on top of account-based systems [76, 86], use a UTXO-style data representation internally. In contrast, privacy designs in the account model [25, 69] require locking all of the accounts in the anonymity set. The UTXO model also offers greater transaction execution parallelism. However, UTXOs can be less intuitive to the user than account balances. Although UTXOs can support arbitrary programmability, it is much easier to implement general programmability in an account-balance design. Account balances are also more fungible, which is an important property for money. It might be useful to consider an account balance data model which minimizes the amount of data stored in the transaction processor in the future.

We emphasize that the transaction processor's internal data representation is distinct from the interface presented to the user. In particular, both of these choices support an account balance *user interface abstraction* (i.e., tallying the total balance of user's holdings, showing their transaction history, etc), even though only one has an account balance internal *data representation*.

2.5 Unspent funds

Formally, we represent unspent funds as triples $\mathsf{utxo} := (v, P, \mathsf{sn})$. Here v is the amount of money and its role in representing unspent funds is clear. The other two elements are an *encumbrance* predicate P, and a *serial number* sn, which we now explain.

The encumbrance predicate P takes two arguments: a transaction tx (to be formally defined later) seeking to spend this utxo, and a witness wit. The predicate returns true if and only if the witness signifies that this spending transaction should be authorized. This is similar to Bitcoin, where each UTXO is encumbered with a *script*, an executable program which evaluates the conditions for a valid spend.

A common encumbrance is that of digital signature authorization. Here the predicate P hard-codes a public key pk and $P(\mathsf{tx}, \mathsf{wit})$ checks that wit consists of a valid signature where the message comprises the serialized spending transaction tx and the signature is under the public key pk. 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. In a system supporting only digital signature authorization, a predicate P can be represented by the public key pk itself.

In our system we permit users to reuse encumbrances, e.g., a user Alice could publish her public key $\mathsf{pk}_{\mathsf{Alice}}$ and receive multiple payments meant for it. Therefore, we need a way 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 $\mathsf{pk}_{\mathsf{Alice}}$).

We express this distinction between otherwise identical UTXOs through a globally unique *serial number* sn, the third component in a utxo. In our security definitions below we require 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 §2.8 for discussion).

Skipping ahead, our system assigns each UTXO a serial number by deterministically hashing all the corresponding transaction's inputs, as well as the output UTXO's encumbrance, value, and its index among all outputs. This in turn references previous serial numbers and recursively incorporates the entire transaction history.[7] The collision resistance of the hash function and the system property that valid inputs can only be spent once guarantees that all serial numbers are globally unique.

2.6 System operations

Logically, Hamilton maintains a record of all unspent funds in existence; consistent with other cryptocurrencies we call this record the *UTXO set*. In order to spend funds, they must be present in the UTXO set. Our system supports the following three kinds of operations: Mint, Redeem, and Transfer, all of which are atomic and are applied one at a time.

Minting and redeeming. The Mint operation creates new unspent funds and adds UTXOs to the UTXO set, whereas the Redeem operation removes unspent funds from the UTXO set, making them unspendable. When deployed these operations also have semantics outside Hamilton: namely, minting would normally correspond to currency in the outside world being set aside for use in Hamilton, whereas redeeming would make them available again. The issuer must choose unique serial numbers for newly minted UTXOs. It suffices to set these as uniformly random, or as result of monotonically increasing counter value (i.e., the issuer minting the i-th UTXO would set its serial number to i).

Value transfers. The Transfer operation both consumes UTXOs and creates new UTXOs; this is the only operation which both adds and removes from the UTXO set. The input to Transfer is a transaction tx comprised of: (a) a list of input UTXOs to be spent; (b) two lists of output values and encumbrances specifying output UTXOs to be created; and (c) a list of witnesses , one for each input. In a valid transaction, balances are preserved, and each input UTXO to be spent has its encumbrance predicate satisfied by the corresponding witness (e.g., a signature). When a transfer operation succeeds, the input UTXOs are completely consumed (removed from the UTXO set) and cannot be used again, and the outputs are available to be used an inputs to other Transfer or Redeem operations. Hamilton also computes and assigns unique serial numbers to the output UTXOs.

No editing of unspent funds. The above three operations are the only ways the UTXO set can be modified. In particular, the unspent funds tracked in Hamilton cannot be modified to change their ownership (encumbrance), value or serial number (see change output discussion in §2.3).

Payment discovery. Transaction history in Hamilton is not public. The sender must give the recipient the newly created UTXOs (or the information needed to reconstruct them) so that the recipient can further spend them. To ensure users know a Transfer is completed and has been applied, the transaction processor is also responsible for responding to queries from users about the existence of UTXOs.[8]

2.7 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). The transaction processor in Hamilton 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. Similarly, Hamilton only accepts and executes Transfer operations where encumbrances of each consumed UTXO

[7] This is similar how Bitcoin whitepaper [67] defined a coin to be a chain of digital signatures.

[8] This is unlike in public blockchains where users can search the publicly available history of transactions to see if they have received payment.

are satisfied (e.g., all three operations are covered by digital signature authorization).

Authenticity. The UTXO set of Hamilton only contains *authentic* funds, as we now define. Define UTXOs created by authorized Mint operations to be *authentic*. Moreover, define UTXOs 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 UTXO set when Transfer is applied.

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

Note that, as a consequence of the three integrity properties defined above the UTXO set 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. An authorized transaction spending authentic funds will always be accepted by the transaction processor.[9]

2.8 Discussion

We carefully designed our data representation (§2.5), system operations (§2.6) and security properties (§2.7) so that any system satisfying these maintains an authentic and authorized UTXO set, 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 member of the UTXO set either in the past or in the future. The issuer chooses uniformly random serial numbers for each Mint transaction output. In Hamilton, for each Transfer the output UTXO serial numbers are set by hashing all the corresponding transaction's inputs, as well as details pertinent to the particular output UTXO itself (see §2.5). Therefore each UTXO serial number recursively incorporates the entire transfer history up to the original Mint transactions that engendered system with these source funds. Under standard cryptographic assumptions, it is infeasible to create two distinct chains of transfers resulting in the same serial number, thus all serial numbers and all UTXOs are globally unique.

No double-spends. Transfer operations permanently mark UTXOs as spent. Therefore, as serial numbers are unique, no UTXO 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 hopes 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. Or, if Alice only has one $5.00 bill available right now, Bob can wait until she receives $5.00 as a change, resubmit the (old, already confirmed) transaction and take possession of Alice's newly received change.

Hamilton's transaction format prevents replay attacks as each transaction references globally unique input UTXOs, and each signature covers the entire transaction, including all its inputs and outputs. Thus, signatures are not valid for spending any other UTXO, 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.[10]

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 [38]. 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).

[9] This does not preclude potential access control outside the transaction processor.

[10] There are other ways to prevent replay and signature reuse attacks, for example, by incorporating a timestamp or an incrementing nonce, or enforcing unique encumbrances: each of them ensure that a signed transaction can effectuate at most one transfer, and that signatures cannot be repurposed. We made our choice to incorporate serial numbers derived from the transaction's history due to its simplicity and flexibility. For example, deterministic serial numbers do not require the sender to maintain state and allow for pre-signing transactions that can be kept online to be broadcast later. This does introduce challenges to programmability since a transaction cannot be signed until the user knows exactly what outputs it is spending; we could use other techniques from cryptocurrency systems to address this.

3 Transaction design

A payment system's transaction format determines the user experience when making a payment and has policy implications in a wide range of areas including the level of user privacy, whether interaction with financial institutions is required, and how minting is performed.

In the abstract design described in §2.6, the transaction processor has full visibility into transactions, including public keys, the transaction graph, and values, and stores the entire UTXO set. Storing the entire UTXO set is unfortunate because it requires the transaction processor to store encumbrances and values. This has an effect on storage and bandwidth requirements (Bitcoin's UTXO state is over 4 gigabytes and Ethereum's is almost a terabyte [80, 91]), and, as described in §1.1, this poses data retention and user privacy challenges. Instead, we explored a design which does not *require* storing encumbrances (which could identify users) and values in cleartext in the transaction processor. Depending on how the system is architected, we believe this design can be later extended to avoid even temporarily showing this data to the transaction processor. In Hamilton, the transaction processor stores unspent funds as a set of opaque 32-byte cryptographic hashes of UTXOs, not UTXOs themselves. The rest of this section explains the technical motivation behind this choice and how to securely create and process transactions in this model. We introduce the transaction format, steps in which a transaction is processed and applied to the state, and implications these choices have on future functionality.

3.1 Processing transactions in Hamilton

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

1. whether the funds exist to be spent;
2. whether the spender has provided authorization to spend the funds; and
3. does the transaction preserve balance of funds.

The first and third items provide authenticity. The second item is authorization. Applying a valid transaction to the UTXO set involves atomically removing the spent funds and creating the new funds under the control of the recipient(s); this in combination with the other checks provide durability.

Separating validation checks. An important part of the design of Hamilton is that these three validation checks can be divided in *transaction-local validation*, which does not require access to shared state, and *existence* validation which does. We can then scale these two pieces of work 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.

Performing local-validation. With this separation in mind, Hamilton has dedicated components, which we call *sentinels*, that receive transactions from users and perform transaction-local validation, which is stateless, and then forward the locally-validated transactions for further processing. This local validation (1) checks that the transaction is correctly formatted, (2) confirms that each input has a valid signature for the output it is spending, and (3) confirms that balance is preserved (i.e., the sum of the outputs equals the sum of the inputs).

Checking existence and executing a locally-validated transaction. Now, given a transaction that passes transaction-local validation, our system needs to atomically check for input existence and, if valid, update the UTXO set as follows. First, check if all transaction's input UTXOs exist in the UTXO set, and abort further processing if any of the input UTXO's are missing. Otherwise, continue and (a) remove the transaction's input UTXOs from the UTXO set, and (b) add the newly created output UTXOs to the UTXO set.

In our current design, sentinels ensure that locally-valid transactions with inputs in the UTXO set have globally unique outputs, therefore we do not need to explicitly check that none of the transaction's output UTXOs exist in the UTXO set once the sentinel has correctly derived serial numbers. However, we do so in the transaction executor in case we wish to support a different transaction format in the future which might not have this property (see §4.3.1).

3.2 UTXO hash set

We start by observing that executing a transaction that passes transaction-local validation does not require access to transaction's witness data, e.g., digital signatures. This is because neither input nor output UTXOs depend on the witness data and so the atomic update is independent of witnesses. Therefore, after sentinels have checked that a transaction passes transaction-local validation, the sentinels could strip witness data and only forward the transaction's inputs and outputs for processing.

Existing UTXO-based cryptocurrencies look up the contents of input UTXOs in a transaction-processor maintained UTXO set, to confirm the user has provided valid UTXOs (i.e., part of the current UTXO set) to spend. Our key insight was relying on the (untrusted) user to provide UTXO data by reducing the problem of checking UTXO correctness to *existence*—Do the funds the user is claiming they can spend actually exist?

By doing this, one can go further and observe that after transaction-local validation, instead of processing and storing the entire UTXO, the transaction processor can operate on cryptographic *commitments* to the UTXOs. In Hamilton we replace the UTXO set with a *UTXO hash set* (UHS), and instead of storing a set of entire UTXOs $\mathsf{utxo} = (v, P, \mathsf{sn})$, we store cryptographic commitments $h := \mathcal{H}(v, P, \mathsf{sn})$ to UTXOs, which we subsequently refer to as hashes, or *UHS IDs*. Here $\mathcal{H}$ is a cryptographic hash function, and in Hamilton we use SHA-256 to derive these hash commitments.

Converting a user's Transfer transaction into commitments to be applied to the UTXO hash set is a new step to transaction processing which we call *compaction*. When processing a Transfer transaction, Hamilton's sentinel computes the hashes for input UTXOs, deterministically derives serial numbers for output UTXOs, and computes hashes for output UTXOs. These two sets of hashes form a *compact transaction*. Sentinels forward compact transactions to the execution engine to be applied to the UHS, described in §3.3.

We note that replacing UTXOs by cryptographic commitments preserves security, and an attacker can not create a transaction that would be invalid in UTXO set model but succeed in the UHS model. 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 $\mathcal{H}$. Therefore, if a transaction format is secure in the UTXO model, then it must be in the UHS model. We explain security of our transaction format in §3.3.

The idea of storing unspent funds as commitments was first proposed as a Bitcoin storage and scalability improvement [49]. We now discuss the benefits and drawbacks of a UHS. It lowers storage requirements, increases flexibility, and improves privacy, but creates challenges for auditing, transaction flows, and programmability.

Storage. In the UHS model the transaction processor only stores a 32-byte hash per individual UTXO, independent of a UTXO's size. If transactions contain programmable features in the future that require a large amount of storage space in the transaction format, the storage requirement for the state remains the same. This state would be maintained by wallets and the user would need to provide the necessary commitment preimages alongside the transaction. It also keeps the data format uniform and for transaction formats that include user-supplied data, this hampers users from storing arbitrary data (such as copyrighted or illegal data [79]) in the transaction processor.

Flexibility. The UHS makes no assumptions about what hashes represent, and this data structure is not limited to UTXOs; it could be applied to a digital currency with balances or some other application that requires atomically swapping hashes. This means we can experiment with different transaction formats or scripting languages without needing to change the core execution engine.

Privacy. The transaction processor does not need to store balances or account information, though sentinels do need to see (but do not need to retain) parts of this information to validate a transaction. We anticipate being able to remove this requirement using cryptographic privacy-preserving designs which we will investigate in the next phase of work [14, 66, 85].

However, this design also presents challenges for certain kinds of auditability, transaction protocols, and programmability.

Auditability. The UHS does not contain enough information to audit the total amount of unspent funds. This type of auditing would probably be important in the context of a digital currency, but can be achieved either by logging data outside the UHS or, to continue preserving privacy, by storing values in homomorphic commitments that can be maintained and tallied using additional cryptographic techniques [69, 75].

Sender/recipient transaction protocols. The UHS design requires a recipient to learn the commitment to find out if they have received funds, and know the serial number, encumbrance, and value to further spend their funds; the transaction processor does not store enough information to help a user recover this if they lose it. This information could be stored elsewhere, or third parties could conceivably provide this service to users. Requiring a user to receive the serial number, encumbrance, and value to spend their funds has implications on our transaction protocol and the types of transactions supported, which we discuss in §3.4.

Programmability. Decoupling transaction-local validation and access to shared state means that future transaction programmability is restricted to only transaction-local state. The UHS requires the person constructing the transaction to be able to specify the start and end states for the modifications to the spender's funds and the recipient's funds. If there are concurrent transactions debiting or crediting an account balance this might be challenging. This is easier in the UTXO model since we do not need to support concurrent access to UTXOs. It would be challenging to implement a complex smart contracting language (such as Solidity [44]) using this abstraction.

We will consider auditing, alternative data models and advanced transaction semantics in the next phase of work.

3.3 Transaction format and execution

Recall that we represent unspent funds as UTXO triples $\mathsf{utxo} = (v, P, \mathsf{sn})$, comprised of a value v, encumbrance

P, and serial number sn (§2.5). We now describe the concrete choices for v, P, and sn, Hamilton's transaction format, and transaction execution in detail.

Values. We represent values v as 64-bit unsigned integers specifying multiples of the smallest subdivision of money, i.e., multiples of $0.01.

Encumbrances. Currently we only support encumbrances of public keys, indicating that the authorization needed to spend this output is a signature on some specific data by the corresponding private key.[11] Thus, an encumbrance P is a 32-byte public key. Our model supports future encumbrances, such as requiring a subset of signatures from multiple public keys.

Serial numbers. It is important that UHS hashes (or, equivalently, UTXO set entries) do not collide, yet at the same time it is possible to spend the same amount v to the same encumbrance P multiple times. This property is both about completeness—the ability to put multiple outputs with same encumbrance and value in the UHS—and about security—to prevent replay and signature reuse attacks (§2.8). This is the role of the globally unique serial number sn.

We make UTXOs (i.e., UHS hashes) 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 Mint or Transfer transaction that created this UTXO. This hash covers all input UTXOs, output encumbrances and values (as well as a unique nonce for each Mint transaction which has no inputs). The second component, idx, is the particular output index, i.e., first, second, etc, output of the transaction.[12] Since inputs can only be spent once and they are all unique, this ensures that valid transactions create unique serial numbers sn and unique output hashes: txid's are different for distinct transactions and the idx values distinguish multiple outputs of the same transaction.

This design matches Bitcoin where previous outputs being spent are referenced via an outpoint, the transaction identifier/output index pair. A Bitcoin outpoint uniquely identifies a previous output and is never reused for a different output once spent, therefore we use serial number and outpoint interchangeably when describing Hamilton transactions.

A notable difference is that Bitcoin transactions only contain outpoints but not the outputs themselves, so validating nodes must look up output information (like the amount) in a local database in order to validate a transaction. As we operate in the UHS model, our transaction processor does not store this output information (it stores a cryptographic commitment to it). Therefore, when spending a previous output in our system, the output's value and encumbrance are included in an input along with outpoint reference to that input.

```
outpoint:
    transaction_id: byte[32]
    index: uint
output:
    public_key: byte[32]
    value: uint
input:
    outpoint: outpoint
    output: output
witness:
    signature: byte[64]

transaction:
    inputs: input[]
    outputs: output[]
    witnesses: witness[]
```

Figure 3: Description of a Transfer transaction. When submitted to the execution engine, the transaction is byte serialized to remove labels and delimiters required for a human-readable format.

We are now ready to fully specify the transaction format, computation of transaction identifiers, and transaction validation.

Mint **transactions.** Unspent funds enter the system as outputs created by Mint transactions. A k-output Mint transaction $\mathsf{tx}_{\mathsf{Mint}}$ is a quadruple $(\vec{v}_{\mathsf{out}}, \vec{P}_{\mathsf{out}}, \mathsf{nonce}; \sigma)$, comprised of two size-$k$ lists of output values $\vec{v}_{\mathsf{out}}$ and $\vec{P}_{\mathsf{out}}$, as well as a unique nonce, and issuer's signature σ. Such a transaction creates k UTXOs with value/encumbrance pairs $(v_{\mathsf{out},1}, P_{\mathsf{out},1})$, …, $(v_{\mathsf{out},k}, P_{\mathsf{out},k})$. We define $\mathsf{txid}(\mathsf{tx}_{\mathsf{Mint}}) := \mathcal{H}((\vec{v}_{\mathsf{out}}, \vec{P}_{\mathsf{out}}, \mathsf{nonce}))$, where $\mathcal{H}$ is a cryptographic hash function.

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 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. (See Figure 3 for a machine-readable specification of a Transfer transaction.)

The transaction's inputs $\mathsf{utxo}_{\mathsf{inp},i} = (v_{\mathsf{inp},i}, P_{\mathsf{inp},i}, \mathsf{sn}_{\mathsf{inp},i})$ must have values that sum up exactly to the values for transaction's outputs: $\sum_{i=1}^{k} v_{\mathsf{inp},i} = \sum_{j=1}^{l} v_{\mathsf{out},j}$. Note that this is different from Bitcoin which requires the sum of the outputs to be less than or equal to the sum of the inputs, because the difference is used as transaction fees which go to the block miner. We do not require fees,

[11] In Bitcoin and other cryptocurrencies, such encumbrances are known as *Pay-to-Pubkey*, or *P2PK*, scripts.

[12] While in this exposition we use 1-based indexing, our software implementation uses 0-based indexing

but could consider them in a future phase of this work.

Similar to Mint transactions, such $\mathsf{tx}_{\mathsf{Transfer}}$ creates l UTXOs with value/encumbrance pairs $(v_{\mathsf{out},i}, P_{\mathsf{out},i})$, and we define $\mathsf{txid}(\mathsf{tx}_{\mathsf{Transfer}}) := \mathcal{H}((\vec{\mathsf{utxo}}_{\mathsf{inp}}, \vec{v}_{\mathsf{out}}, \vec{P}_{\mathsf{out}}))$. (See Figure 4 for an explicit description of this computation.[13])

The transaction format satisfies the properties specified in §2.7 of authorization, authenticity, and durability. It is not possible to create counterfeit money in the system as an outpoint is globally unique and unusable once spent, and transaction-local validation checks, described later, ensure preservation of balance.

Transaction creation. To create a Transfer transaction, users use their private keys to create a digital signature on the txid, which serves as the witness for authorizing the transaction, obtaining one signature per transaction input. Witnesses are not included in the transaction identifier so signing can be deferred by the sender to after the outpoint 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, meaning that funds can be spent atomically from multiple public keys in a single transaction.

Once a transaction is finalized, the users will deterministically derive outpoints (i.e., serial numbers) of each of the output UTXOs from the transaction contents. Users store this outpoint information in their wallets.

Transaction execution. As described in §3.1, transaction execution in Hamilton can be separated in two parts: (a) transaction-local validation, and (b) checking for UHS hash existence and execution of a locally-validated transaction.

The sentinel completes transaction-local validation of a Transfer transaction by performing the following three checks:

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.

[13] In our system we use SHA256 both for computing UHS hashes and transaction identifiers. To make vector serialization unambiguous we also explicitly hash k and l as part of tuple serialization.

See Figure 5 for Python pseudocode of a `validate` function specifying the transaction validation algorithm.

Once 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 it sends to the transaction processor, which maintains the UHS, for existence checks and execution. See Figure 6 for a pseudocode description of the transaction compaction algorithm.

The `swap` abstraction. Note that while the `validate` function 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 compacted hash values. Consequently, processing Hamilton transactions at scale reduces to the challenge of implementing a fast, scalable, and durable system for executing the following kind of UHS primitive, which we call `swap`. We describe two such systems in §4.

A UHS system maintains a set of hashes, and exposes a single operation called `swap`. 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 UHS by erasing all input hashes and inserting all output hashes. All other hashes in the UHS remain unchanged. Figure 7 describes contents of a compact transaction and how such a transaction is then processed by `swap`.

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

Security. Note that the transaction format itself guarantees that old and new hashes output by the `compact` function are unique, as the hashes commit to the entirety of pertinent transfer history up to the distinct (due to presence of a nonce) Mint's. Once `swap` has removed hashes from the UHS they can not be recreated (this would require duplicate outpoints) thus ensuring that outputs cannot be double-spent and transactions cannot be replayed: the subsequent spends would be rejected by `swap`'s existence checks as input hashes would not be present in the UHS. Similarly, since the swap abstraction provides atomic deletion and addition of inputs and outputs, the transaction is final once accepted and cannot be reversed. Finally, transaction IDs will never repeat for valid transactions as described above, so signatures cannot be reused once the transaction is settled as changing any aspect of the inputs or outputs of the transaction will change the transaction ID, resulting in an invalid signature.

```
def transaction_id(transaction):
    hash_args = [len(transaction['inputs'])]
    for inp in transaction['inputs']:
       hash_args += [inp['outpoint']['transaction_id'], inp['outpoint']['index'],
                     inp['output']['public_key'], inp['output']['value']]

    hash_args += [len(transaction['outputs'])]
    for out in transaction['outputs']:
       hash_args += [out['public_key'], out['value']]

    return serialize_and_hash(hash_args)
```

Figure 4: Calculation of transaction identifier.

```
def validate_local(transaction):
    if len(transaction['inputs']) < 1:
        return False
    if len(transaction['outputs']) < 1:
        return False
    if len(transaction['witnesses']) != len(transaction['inputs']):
        return False

    total_input_value = 0
    for inp in transaction['inputs']:
        total_input_value += inp['output']['value']

    total_output_value = 0
    for out in transaction['outputs']:
        total_output_value += out['value']

    if total_input_value != total_output_value:
        return False

    txid = transaction_id(transaction)

    for inp, wit in zip(transaction['inputs'], transaction['witnesses']):
        if not check_signature(inp['output']['public_key'], wit['signature'], txid):
            return False

    return True
```

Figure 5: Transaction validation algorithm.

```python
def input_hash(input):
    hash_args = [input['outpoint']['transaction_id'], input['outpoint']['index'],
                 input['output']['public_key'], input['output']['value']]
    return serialize_and_hash(hash_args)

def compact(transaction):
    txid = transaction_hash(transaction)
    input_hashes = []
    for inp in transaction['inputs']:
        h = input_hash(inp)
        input_hashes.append(h)

    output_hashes = []
    for i, out in enumerate(transaction['outputs']):
        inp = {
            'outpoint': {
                'transaction_id': txid,
                'index': i
            }
            'output': out
        }
        h = input_hash(inp)
        output_hashes.append(h)

    return (txid, input_hashes, output_hashes)
```

Figure 6: Calculation of UHS input hashes and transaction compaction algorithm.

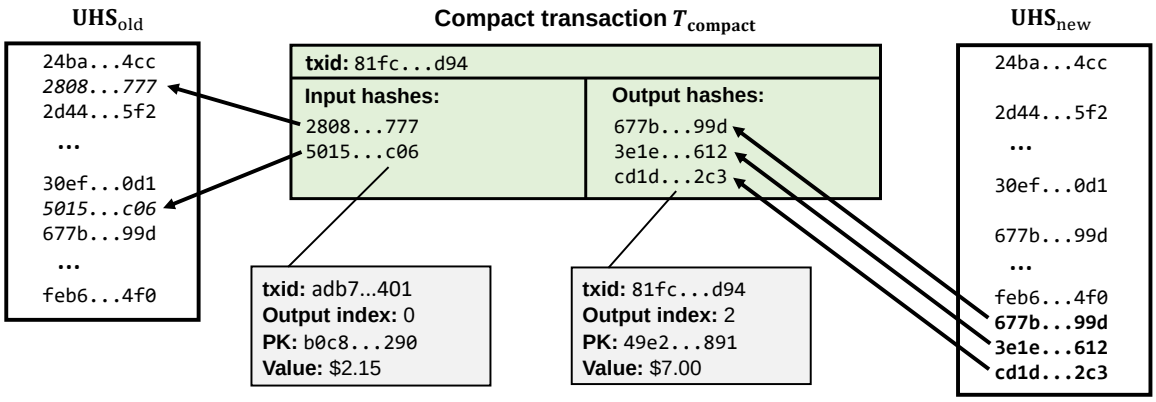


Figure 7: Processing of a compact transaction. As explained in §3.3, a transaction T is first validated, and after that T is compacted to obtain the corresponding compact transaction T_{compact}. The compact transaction T_{compact} consists of a transaction identifier (txid), input hashes (referring to previously committed transaction outputs), and output hashes (referring to outputs of the transaction T itself). To process T_{compact}, the `swap` function atomically does the following: it checks that all input hashes of T_{compact} are in the UHS, and if so it obtains an updated UHS by erasing T_{compact}'s input hashes (highlighted in italics) and adding T_{compact}'s output hashes (highlighted in bold). All other hashes in the UHS remain unchanged.

3.4 Transaction protocol

A transaction protocol is the series of user actions (or actions performed by wallets on the user's behalf) needed to create and submit a transaction to the transaction processor. This includes how the recipient shares their public key with the sender, who participates in constructing and authorizing the transaction, who submits the transaction to the transaction processor, how confirmation (or rejection) is communicated, and any other actions needed for a transaction to succeed. For example, a protocol may be: (1) the recipient shares their public key with the sender, (2) the sender constructs, signs, and submits a transaction to the transaction processor, and (3) both the sender and recipient query the transaction processor (possibly repeatedly) to find out if the transaction has completed successfully. Note that once constructed and shared, either the sender or recipient could submit the transaction.

Our choice of transaction data model and format directly impact potential transaction protocols. For example, transaction compaction for the UHS adds a new communication step requirement between sender and recipient. Note that the *recipient* does not need to authorize the transfer, beyond sharing a public key with the sender. This means that a sender could construct and submit a transaction without the recipient's knowledge (e.g., by reusing a public key), or without sending the recipient the constructed transaction. This would make the funds unspendable, and the recipient might not even know they exist. 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, other 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 further spend those funds. Even if the UHS were public, recipients would not be able to unilaterally detect payments as the output hashes are only generated during transaction construction.

This communication requirement means we cannot always safely execute certain transaction protocols, including non-interactive or "billboard" payments. We define non-interactive payments as transactions where the recipient does not need to engage with the sender at all at the time of transaction. For example, a charity may want to solicit donations in a train station by posting their public key as a QR code. If the sender did not communicate with the charity to *also* send the new outputs, the money would be rendered unspendable (it is controlled by the charity's public key, but the charity does not have enough information to construct a valid transaction to spend it). One way to address this would be to have the transaction processor store the outputs as well, so the recipient could query for them later, but this would require storing public keys and amounts, which would allow users to be tracked across transactions.

An alternative transaction format could compute the hash with only the public key and value, so the recipient could deterministically find out if they have received money without needing to know exactly how it was spent. This fixes the above problem but has downsides. The swap function would need to explicitly check for and reject duplicate transaction IDs to prevent transactions from being replayed. Unlike in the format described above, it would be trivial to recreate the same input set, and thus the transaction ID, if outputs with the same public key and value were created, allowing signatures to be reused. This would effectively force users to generate new public private key pairs for transactions of the same value because the swap function must reject transactions that repeat the same public key and value pair.

Learning transaction confirmation. There is no public ledger of transactions, so recipients must rely on the transaction processor to learn about the status of outstanding transactions. In our system they do this by querying the transaction processor directly, but we could also consider a design where the transaction processor signs confirmed transactions so the spender could relay confirmation directly to the recipient. In §4, we introduce a service that responds to user queries about whether a transaction was successful. This service stores transaction IDs and output hashes. As described above, recipients must receive either the transaction ID or output data about the transaction before they can confirm it has been successful; this can be shared at any point after transaction construction (including before submission).

Instead of requiring the user to poll for transaction confirmation, the processor could support *receipt callback endpoints*. Users would specify a *callback endpoint* in the transaction format and the transaction processor would push a notification to that endpoint when the transaction is complete. Users are already familiar with this payment protocol as it is commonplace for credit card payments over the Internet: an e-mail address or phone number is provided at transaction time, and a receipt is sent to that address upon completion. It may be possible for third-party intermediaries to emerge who do nothing but provide a finality inbox service to users, much like how e-mail providers hold messages until users grab them. Importantly, this callback would not affect the execution of the transaction itself, merely the finality notification, so these intermediaries would not need to take custody of user money.

Using a receipt callback endpoint has two primary drawbacks. First, it increases data storage requirements

within the UHS or within an alternative look-up service and, second, it requires high availability for the callback endpoint. To link a successful transaction from the UHS to an endpoint (e.g., an email address), the endpoint data would need to be included within the UHS. If not included, a separate service would need to scrape the endpoint data along with the transaction ID or output hashes from the validation set. This increases data retention and, accordingly, impacts privacy and likely performance. Furthermore, if the endpoint is unavailable or incorrectly specified by the user, the confirmation notification would fail, leaving polling the transaction processor as the only alternative. A central directory containing all public keys and notification endpoints would simplify the process, but creates a similar privacy risk by linking transactions to personally identifying data (e.g., email addresses).

Limitations on types of transactions supported. Hamilton only supports *push payments*—the sender must explicitly authorize and initiate each transaction. We do not yet support *pull payments*, where the sender can preauthorize the recipient to continuously charge money to the sender, like with a subscription service. It is not clear how to support this using a UHS because transactions, and transaction authorization, must reference the specific funds being spent.

3.5 Learnings

Constructing a payment system using a UHS showed how choices in transaction design and data storage can impact data retention requirements and transaction protocols (including potential use cases). Importantly, the flexibility of a system's transaction protocols will impact what use cases are possible and the user experience, which are critical for adoption. The amount of data the transaction processor retains and to whom it is visible dictates what out-of-band interactions between users are needed. During out-of-band communication, wallet communication protocols could fail or transaction data could be lost, creating edge cases where a sender no longer has the authority to spend funds and the receiver does not have the information required to reference them. We leave solving these tradeoffs and building fully functional user wallets to future work.

4 Processing transactions at scale

To illustrate how architecture design choices for the transaction processor affect the broader properties of a CBDC, we designed and implemented two architectures. This required exploring the tradeoffs in user-facing wallet software, the payment processor's back-end software, and the communication layers between them. Importantly, these transaction processing systems would require significant further development for real-world CBDC usage. We present them as examples to illustrate key ideas and facilitate discussion.

There are many other potential architectures to explore for fast transaction processing. We made several early design choices that ultimately defined other properties of the system. Examples of these early design choices include defining how users learn about execution results, or whether those results are globally *linearizable*, meaning that a time-based ordered list of transaction history logically exists and can be materialized [58].

In this section we describe the two architectures we implemented and evaluated for processing transactions at scale. Both would require solving significant additional challenges before they would be ready for use in a production-quality system.

4.1 Consistency

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 investigate this type of architecture in §4.2. However, our correctness requirements do not *require* materializing a linear transaction history. In §4.3, we describe an architecture which uses a variant of two-phase commit [51] to achieve atomicity and serializability without actually materializing a linear order.

Our invariants suggest that we could further relax consistency requirements so transactions would not need to execute atomically. That is, the new funds could be created lazily and a user might observe that their spent funds are not available for some time before the transferred funds are available to spend. (Note that delayed execution is quite common in today's payment systems where settlement might even take days.) In addition, we might not require that a total order of all transactions exists (even an implicit one). Relaxing one or both of these guarantees might improve performance. We leave these explorations to future work.

As described in §3.1, in both of our implemented designs a sentinel receives a transaction from a user, performs transaction-local validation, condenses the transaction into a compact transaction, and sends it to the execution engine to enact the transfer and update the UHS.

4.2 Atomizer design

This design takes a two-stage pipelined approach: users submit transactions to sentinels and then subscribe to a *watchtower* to learn transaction status. *Shards*, each of which stores some portion of the set of unspent outputs (the UHS), receive compact transactions from the sentinels. Shards check to see if the inputs to a transaction exist, and then send this information to an ordering server we call an *atomizer*, which produces a linear ordering of transactions in *blocks* of state updates to the UHS. These blocks are made durable on the *archiver*. Finally, each block is broadcast and applied atomically in order (by block height) to each shard in parallel. Each shard keeps track of its current block height. The watchtower also digests blocks and keeps state on transaction status for users.

Figure 8 shows a diagram of the components in the atomizer architecture and the data flow between components. The order of messages during normal transaction execution are described below:

1. User wallet submits a valid transaction to the sentinel for execution by the system.
2. Sentinel validates the transaction and responds to the user that the transaction is valid and is now pending execution.
3. Sentinel converts the transaction to a compact transaction and forwards it to the shards.
4. Shards check the input UHS IDs are unspent and forward the compact transaction to the atomizer. The shards attach their current block height and the list of input indexes the shard is attesting are unspent to the notification.
5. Atomizer collects notifications from shards and appends the compact transaction to its current block once a full set of attestations for all transaction input UHS IDs have been received. Once the *make block* timer has expired, the atomizer seals the current block and broadcasts it to listeners. Shards update their current block height and their set of unspent UHS IDs by deleting UHS IDs spent by transactions in the block and creating newly created UHS IDs. The watchtower updates its cache of UHS IDs to indicate which have been spent and created recently.
6. User wallet queries the watchtower to determine whether their transaction has been successfully executed.
7. Watchtower responds to the user wallet to confirm the transaction has succeeded.

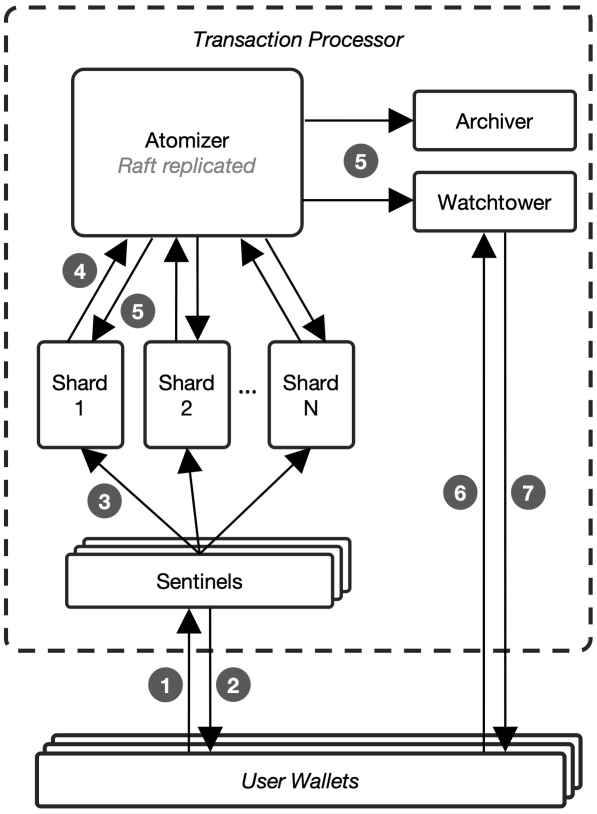


Figure 8: System diagram for the atomizer architecture and inter-component data flow

4.2.1 Validating transactions

The sentinel is responsible for validating all transaction rules except the existence of inputs. This includes checking that the transaction is correctly formatted, that it preserves funds, and that any necessary signatures are present and valid. If a transaction does not meet these criteria, the sentinel will return an error to the user without forwarding the transaction for further processing. We could extend the transaction format and sentinel validation to support more complex encumbrances in the future.

Assuming the sentinel validates the transaction successfully, it converts the transaction to a compact transaction and broadcasts this to the shards. As described in §3, a compact transaction is the minimal data necessary to validate that all transaction inputs are present in the UHS, and update the UHS by deleting spent inputs and inserting new outputs. Each shard is responsible for a range of UHS IDs. Relevant shards (responsible for a UHS ID range covering input UHS IDs in a transaction) will check if the inputs exist. If they do, they will form an *attestation* for the atomizer, which contains the compact transaction, a list of the input UHS ID indexes the shard is attesting are unspent, and the block height for which the attestations are valid. This means an attestation is a confirmation by the shard that the input exists *as of a specific block height*. Note that the shard does not remove inputs or change state in any way at this point, and might attest to the same input across multiple transactions. This could conceivably result in a double spend but is prevented by the atomizer, as described below.

4.2.2 Ordering transactions

The atomizer collects, processes, and applies attestations from shards. The atomizer stores attestations by block height and transaction ID, and when a transaction has a complete set of attestations at the latest block height (we will relax this requirement later), the atomizer considers it for a block. In our implementation, the atomizer produces blocks on a specific schedule (in §6, every 250ms), but could also produce blocks when a certain number of complete transactions are ready to be included in a block. The atomizer creates a block of complete compact transactions, based on the order in which a complete set of input attestations were received for the transaction. Importantly, the atomizer does not include transactions containing inputs already referenced by another transaction in the block, even if both transactions have a full set of input attestations. The atomizer assigns the block the next sequential block height, makes the block durable, and then broadcasts the block to the shards, watchtower, and archiver. A transaction is considered finalized (meaning its effects will eventually be visible to users) once the block is made durable, as described in §4.2.4.

4.2.3 Updating state

As shards receive blocks, they atomically apply the blocks to their local state; they remove any inputs that were spent in the block and insert new UHS IDs created in the block into their local data stores. (Each shard does this for its own UHS ID range.) Once it has completely processed a block, the shard updates its block height to be used in future attestations.

Watchtowers receive blocks from the atomizer and maintain a time-limited cache of recently executed compact transactions. Users can query the watchtower by transaction ID to find out whether the system has successfully executed their transactions. Another service could provide longer-term, historical transaction status by reading the blocks from the archiver and maintaining an index, much like a cryptocurrency block explorer.

Correctness relies on the atomizer as an ordering server. In the design described above, an atomizer will not consider an attestation if it is not marked with the latest block height. A shard also will not update its block height until it has fully processed the previous block's updates, so at the time the shard produces an attestation the atomizer will accept, it must have processed the previous block, destroying any spent inputs. A shard might attest to the same input twice at one block height for different transactions, but the atomizer will deduplicate this and allow only one of the transactions into a block, whichever receives a full set of input attestations first.

The reliance on block height for attestations creates a synchronization loop problem between shards and the atomizer. A shard's attestations may no longer be valid after the atomizer updates its block height and before the shard processes new blocks. To allow the use of attestations that are still valid but not current (i.e., UHS IDs still not spent as of a certain block height), we introduce a spent transaction output (STXO) cache in the atomizer. If an attestation has a non-current block height, the atomizer checks in the STXO cache if the attestation's UHS ID has been spent in recent blocks. If not, the attestation is still valid and the transaction can proceed. The STXO cache depth determines the maximum usable attestation "age" (i.e., the difference between the block heights of the attestation and the atomizer). With each new block produced, the atomizer adds newly spent UHS IDs to its STXO cache and discards UHS IDs older than the cache's depth.

The STXO cache significantly improves performance because stale attestations can still be considered by the atomizer across block boundaries. Furthermore, the atomizer's STXO cache makes it possible for shards to process new compact transactions from sentinels in parallel with digesting a block. By taking a snapshot of its existing UHS partition before processing a block at

height h, the shard can issue attestations with the snapshot's block height of $h-1$. Once the shard has fully digested the block, the old snapshot can be discarded and attestations will reference the latest block height h.

4.2.4 Fault tolerance

The atomizer operates in a replicated state machine; in our implementation we use Raft [73]. We replicate inputs to the atomizer's functions to process transactions, make blocks, and prune blocks (shard attestations, complete transactions, and block heights). The replication process makes sure that blocks are replicated across atomizers; the lead atomizer (and at least half the replicas) will remember the block until an archiver has received it and notified the lead atomizer that the block is safe to prune. The lead atomizer will make sure this operation is replicated. Archivers are the long-term storage for historical data in the system to reduce the storage requirement for the atomizer. A block, and thus the transactions contained within it, is committed once the command to produce the block has been replicated by the atomizer state machine. At this point, a majority of the atomizer replicas have the state necessary to broadcast the block.

Interestingly, shards do not require consensus to stay up to date, since they apply blocks from the atomizer in block-height order. We can replicate shards by simply creating shards with overlapping UHS ID ranges. Each shard range copy can process blocks and provide correct attestations as of their current block height. Sentinels can send transactions to *any* of the copies of a shard range; if one fails, it can try another. Note that if a replica is out of date (has not yet processed the most recent block outside the atomizer's STXO cache) its attestations will be discarded at the atomizer; this would require the user or sentinel to retry the transaction. The atomizer must broadcast blocks to all shard replicas.

If there is a leadership change in the atomizer Raft cluster after a `MakeBlock` command has been replicated but before the resulting block has been broadcast to the archiver, the archiver will request the missing block from the new atomizer leader once a subsequent block is received and the discontinuity is recognized. Similarly, shards and watchtowers will request missing blocks from the archiver, allowing them to catch up after an atomizer leadership change. Since blocks are stored by the atomizer cluster until an archiver has backed them up, there is no risk of blocks being lost even if broadcasting them fails.

Sentinels do not need to retain state and thus do not need state recovery. New sentinels may be spawned at any time to support higher loads or drained as load decreases.

Shard state will be a consistent but possibly stale view of the overall UHS maintained by the system. Shard data loss is prevented by replicating each UHS hash on more than one shard. If a shard becomes unavailable, the service can still be maintained if the replication factor does not fall below one. A replacement shard can be created by either re-applying the blocks stored by the archiver or copying the required state from other replica shards. Additionally, if a shard falls more than one block behind, the shard can get the missed block(s) from the archiver and apply them to catch up. Blocks not yet in the archiver can be retrieved from the atomizer leader, making the atomizer the real-time source of consistency and synchronization between all system components.

The archiver is the historical record of state transitions of the overall UHS and can be used for recovery in the event of component failures or network degradation. If archive data is lost, the system can continue to operate as long as all shards remain synchronized with the atomizer. However, in this case, future shard reconstruction from archive data would be impossible. To alleviate this problem and speed up shard recovery from blocks in the archive, periodic snapshots of shard state at regular block heights could be taken. This would require fewer blocks to be processed while reconstructing shard state and would also remove the necessity of the archiver to store blocks prior to the most recent snapshot. In this way, the archiver could be recovered to full functionality.

4.2.5 Preventing double spends

Suppose an adversary tries to double-spend an output that was previously spent in a transaction confirmed a long time in the past (e.g., minutes). Typically, this will be caught and prevented at the shard layer. A shard copy that is responsible for the UHS ID of the input that references the previously spent output will check its UHS range and see that the UHS ID is not present, and thus not spendable. The shard will thus not forward an attestation to the atomizer for the offending input. Since the atomizer will never receive a full set of attestations for each input to the malicious compact transaction, the transaction will not be included in a block and therefore will not execute. The atomizer eventually discards these incomplete compact transactions.

Consider the case where two transactions (tx_A and tx_B) are submitted concurrently and double spend an output o; this means each transaction references o in an input. Assume that this double spend succeeds; this means that the UHS ID for this output is attested to twice at block heights h_1 and h_2, by shards s_1 and s_2. Assume the atomizer is at height h, and there is no STXO cache.

- Case 1: $h_1 = h_2$: shards s_1 and s_2 (these might be the same shard) will send attestations a_1 and a_2 with heights h_1 and h_2; $h_1 = h_2$ must be $\leq h$ (if the

atomizer is at height h, it could not have broadcast a previous block higher than h). The atomizer will receive attestations a_1 and a_2 in some order; assume it is a_1 first. If there is no new block created before the atomizer receives a_2, then later when making a block the atomizer will detect the duplicate attestation a_2 in tx_B and discard tx_B. If there is a new block created, then the atomizer will be at height $h + 1 > h$, which means $h + 1 > h_2$ and the atomizer will discard a_2 because its height is not current.

- Case 2: Assume $h_1 < h_2$. If $h_1 < h_2 < h$, then the atomizer will discard both attestations. If $h_1 < h_2 = h$ (h_2 cannot be greater than h for the reason above), then the atomizer will discard s_1's attestation when it is received, but accept s_2's because it is up-to-date.

Since both attestations will not be accepted by the atomizer, tx_A and tx_B cannot both succeed.

We can extend this argument to include the STXO cache by considering that if $h_1 < h_2 < h$ the atomizer will reject the attestation from shard s_1 if it is too old (h_1 has been phased out of the cache), and reject the attestation from s_2 if the attestation from s_1 at height h_1 is still in the cache.

4.2.6 Watchtower

The atomizer design uses a queryable *watchtower* to efficiently communicate a transaction's success to users. A transaction reaches finality (i.e., success) when the atomizer includes its compact version (containing input hashes, output hashes, and transaction ID) in a finished block. The simplest way to notify users would be for the system to broadcast completed blocks to all users, and require users to check each block for their transaction ID. This is analogous to how each node in the Bitcoin network stores the entire block history. Given this system's throughput requirement of 100,000 transactions per second, this high volume of transactions would create unreasonable bandwidth and processing demands for users. Similarly, broadly sharing the complete transaction history would undermine privacy (e.g., the transaction graph could be seen).

Instead, the system provides a watchtower which aggregates error messages from system components and blocks from the atomizer, and stores an index of recently confirmed transactions and errors to share with authorized clients upon their request. Users query a watchtower with a transaction ID and UHS IDs, and the watchtower returns the status of the UHS IDs corresponding to the given transaction ID within the system to indicate whether or not it was successful.

By requiring a tuple of transaction ID and UHS IDs as the watchtower query payload from users, the watchtower reveals minimal information about transactions. A recipient of funds in a transaction will be able to query about the status of their own outputs, but they cannot learn the status of the transaction inputs or other outputs which the sender has not shared with them. Similarly, the sender of funds in a transaction can confirm that the system accepted the outputs they created, but they cannot learn about how the recipient spends those outputs, since the sender will not know the transaction ID for the transaction in which the recipient spends those outputs. For additional privacy, the watchtower could challenge the user to produce a signature for the UHS ID they are querying to ensure the user actually has the ability to spend the given output.

4.3 Two-phase commit design

In this architecture, shards use variants of two-phase commit and conservative two-phase locking [43] to atomically apply transactions to the UHS. There is no materialized order of transactions, though two-phase commit ensures serializability. There are two components: transaction coordinators and shards. Each logical shard is responsible for a subset of the UHS IDs which are unspent within the system, in the same fashion as in the atomizer architecture. Unlike in the atomizer design, there are no blocks, archivers, or atomizers; shards do not have any notion of block height; sentinels are responsible for communicating transaction status back to users synchronously; and we require a replication protocol for shard fault tolerance.

Figure 9 shows a diagram of the components in the 2PC architecture and the data flow between components. The order of messages during a single transaction's successful execution are described below:

1. User wallet submits a valid transaction to sentinel.
2. Sentinel converts the transaction to a compact transaction and forwards it to the coordinator.
3. Coordinator splits input and output UHS IDs to be relevant for each shard and issues a *prepare* with each UHS ID subset.
4. Each shard locks the relevant input IDs and reserves output IDs, records data about the transaction locally, and responds to coordinator indicating it was successful.
5. Coordinator issues a *commit* to each shard.
6. Each shard finalizes the transaction by atomically deleting the input IDs, creating the output IDs, and updating local transaction state about the status of the transaction. The shard then responds to coordinator to indicate that the *commit* was successful.

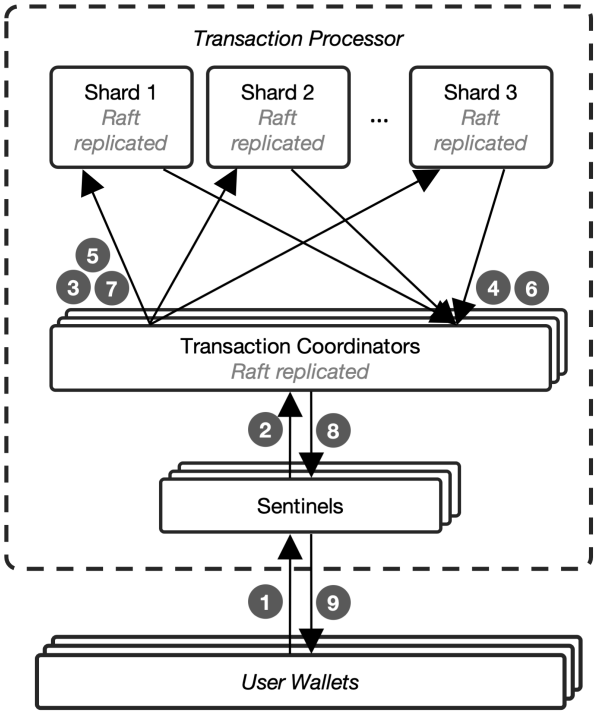


Figure 9: System diagram for the 2PC architecture and inter-component data flow

7. Coordinator issues a *discard* to each shard informing them that the transaction is now complete and it can forget the relevant transaction state.

8. Coordinator responds to sentinel indicating that the transaction was successfully executed.

9. Sentinel responds to user wallet, forwarding success response from coordinator.

4.3.1 Batching Transactions

Instead of processing one transaction at a time, a coordinator receives compact transactions from sentinels (the same as in §4.2) and adds them to a batch of many compact transactions, which represents a single distributed transaction, or a *dtxn*. After a delay, or when a batch has reached a size threshold, a coordinator initiates the protocol to try to commit the transaction batch. Many coordinators could create and execute dtxns in parallel. There are two phases to commit a dtxn:

1. **Lock.** The coordinator contacts each shard responsible for a UHS ID included in the batch and requests that it durably lock the input UHS IDs and reserve the output UHS IDs. (Note that in the transaction format described in §3.3 output UHS IDs are guaranteed to be unique across transactions by the nature of our transaction format, so this reservation is not strictly necessary. It is possible in other transaction format designs UHS IDs will not be guaranteed to be unique, so we do not assume this and reserve outputs.) Each shard responds to the request indicating which transactions in the batch had their IDs successfully locked or reserved, and which no longer exist, or were already locked/reserved by a different dtxn.

2. **Apply.** The coordinator uses the shards' responses to determine which compact transactions in the batch 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 transactions in the dtxn batch to complete and which to cancel. Each shard then atomically unlocks the input UHS IDs belonging to a canceled transaction, and deletes input UHS IDs and creates the output UHS IDs for successful transactions.

Once every shard participating in the batch has completed the second phase, the coordinator informs each sentinel whether its transactions were successfully executed or rejected by the shards. The sentinels in turn forward these responses to the users who submitted the transactions.

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 same input. Investigating methods to fairly resolve concurrency conflicts is left to future work.

Batching many user payments into larger distributed transactions amortizes the cost of 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.

4.3.2 Fault Tolerance

Each transaction coordinator and shard is made fault tolerant via a replicated state machine. Our implementation uses Raft. Sentinels maintain state during the duration of the user wallet request to return transaction status to the user. If a sentinel fails before a client request has been forwarded to a coordinator, the user's wallet will need to retry its transaction 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. Before initiating each phase of the distributed transaction, the coordinator replicates the inputs to both the lock and apply commands 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 phase will return the stored response to the new coordinator leader. To ensure proper completion of the apply phase across all shards, shards will remember the response for the apply phase until the coordinator has received responses from all shards in the dtxn and issued a "discard" message to inform shards the dtxn is complete and can be forgotten. Note that discards can be applied lazily and the transaction coordinators can inform the sentinels the transactions were successful before issuing the discard.

Similar to coordinators, only the leader in a given shard cluster 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 the Raft protocol. Once a dtxn has entered the lock 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.

If a user's wallet loses connection to its sentinel while waiting for a response to its transaction, that response will be lost and the wallet will have to query the shards to discover whether their transaction has succeeded, or if it will need to be retried.

4.3.3 Preventing double spends

Assume there is a double spend of output o by tx_A and tx_B (as described in §4.2.5). The UHS ID u for output o is handled by one shard cluster, at most. In order for tx_A to succeed, the transaction coordinator handling the compacted tx_A ($c(tx_A)$) must submit a dtxn containing $c(tx_A)$ which locks u. If $c(tx_A)$ succeeds, then the transaction coordinator will eventually call apply and the shard will destroy u. For $c(tx_B)$ to succeed, another transaction coordinator must also lock u, but it cannot do so without contacting the same shard and seeing either that u has already been locked by the transaction coordinator executing $c(tx_A)$ or that u no longer exists.

4.3.4 Comparison to atomizer design

There are two primary differences between the 2PC and atomizer architecture. First, the 2PC architecture does not materialize an immediately available total ordering of transactions, which the atomizer architecture does through a sequence of blocks. Although it might be possible to generate a partial ordering of transactions post-execution using a technique such as Lamport timestamps [64], this is left to future work. Ultimately, however, in two-phase commit, unrelated transactions could execute in any order while maintaining serializability and correctness from double-spends. This difference may have negative implications for future auditability but positive implications on the privacy of the system from post-execution transaction flow analysis. What's more, relaxing the requirement for a total ordering removes the primary bottleneck in the atomizer architecture (the atomizer cluster itself). As shown in §6, this means the 2PC architecture can scale linearly in throughput by deploying additional shards and transaction coordinators, whereas the atomizer architecture is limited by the resource constraints (network bandwidth and CPU) of a single server, the atomizer leader.

Second, the atomizer uses asynchronous communication between components whereas the 2PC architecture uses typical synchronous remote procedure calls for inter-component communication. Using blocks to coordinate state between individual components makes the consistency and replication story for the atomizer simpler, but it also means that transactions can fail for transitory reasons related to inter-component message timing that are opaque to the end user. When the atomizer-based system is operating at or close to peak capacity, or during degraded network conditions, users may have to retry their transaction multiple times before successful execution if a shard or the atomizer is overloaded and cannot provide or validate attestations before they expire. Furthermore, since transaction status and error reporting is handled entirely by the watchtower, users will need to actively poll the watchtower at the time of the transaction to discover its result.

In contrast, 2PC uses a more complex availability and consistency strategy that relies on replicated state machines for shards and coordinators. This adds significantly more code complexity, increasing the attack surface for exploiting bugs. It also requires careful consideration for how to safely recover partially completed

dtxns. However, from an end-user perspective, much less complex software is required to successfully complete a transaction. Once a coordinator has replicated a user's transaction, it will always run to completion, and the user will receive a success or error response directly from the sentinel that received their transaction. Furthermore, as shown in §6.2.1, the lack of message timing complexity between internal system components and the lack of fixed inter-block delays results in reduced transaction tail latency for the same throughput. Users would only need to retry transactions in the rare case of simultaneous failure of several internal system components, and never once replicated by a coordinator. The 2PC system itself ensures successful completion of transactions rather than depending on the user to work within the best-effort semantics provided by the atomizer architecture.

4.4 Considering blockchain technology

Many have suggested using blockchain technology to design a central bank digital currency; blockchain technology has been used to refer to a wide range of technologies comprising distributed consensus protocols, hashing, digital signatures, zero-knowledge proofs, and distributed databases. Many of these technologies predate the first time the term was used in Bitcoin [68].

We found that using a blockchain-based system in its entirety was not a good match for our requirements. The first reason is due to performance. Byzantine fault tolerant consensus algorithms and other new blockchain consensus protocols generally provide lower performance than Raft, and any single state machine architecture will be limited by the resources of one server.[14] Our atomizer architecture is inspired, in part, by a permissioned blockchain design. Though we minimized the functionality in the atomizer to just deduplicating inputs, we were unable to achieve throughput greater than 170K transactions per second in a geo-replicated environment; the cause being network bandwidth limitations between replicas in other regions. If bandwidth constraints are relaxed, computation in the leader atomizer to manage Raft replication and execute the state machine becomes the bottleneck. Section 6 describes bottlenecks and the performance of the atomizer under different workloads.

Second, there was no requirement to distribute trust amongst a set of distrusting participants. The transaction processing platform is, by its nature, controlled and governed by a central administrator, the central bank. Blockchains use relatively new distributed consensus protocols which operate in a very different adversarial environment. This introduces software and operational complexity. A CBDC has different adversarial assumptions and should rely on the simplest, most well-understood, well-tested protocols to achieve its goals.

Note that it might be beneficial to distribute read-only copies of the data to other actors for auditing purposes. This can be done in many architectures and must be carefully balanced with data privacy and performance considerations. Given a workload target of 100K transactions per second and a minimum transaction size of 64 bytes, this would require transferring over 500GB of data per day, which is out of scope for most users. In the next phase of work, we intend to explore adding forms of cryptographic auditing that do not require replicating all transactions.

Reasons to consider blockchain technology. Central banks that wish to distribute trust and governance might still consider blockchain technology for their implementations, and it might make sense to use blockchain technology if CBDC designers decide that intermediaries should run nodes in the system that validate and execute transactions. The state-of-the-art in blockchain performance is improving, which might remove this concern as a factor in the future.

5 Implementation

We implemented the 2PC and atomizer architectures as a set of standalone applications in C++. We used C++17, the most recent C++ specification that was widely supported by mainline compilers at the time, supporting builds using both GNU GCC and LLVM Clang. The codebase[15] has been tested on Linux and macOS but should be portable to any UNIX-like system with relatively minimal changes. The primary dependence on a UNIX-compatible API is our use of UNIX sockets for network communication. Aside from that the codebase uses only standard C++ and some third-party libraries so may be portable to non-UNIX systems in the future.

Clients communicate with the sentinel and watchtower components via a custom serialization protocol, via single, short-lived TCP connections. Our watchtower implementation accepts polling client status requests. We anticipate that users will have different status confirmation needs regarding latency, client overhead, interoperability, and range of historical data availability. Future implementations could allow clients to make archival queries for historical transaction information, reduce latency by adopting a more sophisticated publish/subscribe design where clients could subscribe to asynchronous updates for pending transactions, or accept queries via alternate protocols or more standard serialization formats.

We used four third-party libraries for the core codebase: LevelDB [56], NuRaft [41], libsecp256k1 [18] and vendored components from Bitcoin Core [17]. We use LevelDB for internal shard storage and atomic write

[14] Layer 2 designs can provide higher throughput, but add timing complexity and have different security assumptions.

[15] Published at https://github.com/mit-dci/opencbdc-tx

transactions as well as a persistent implementation of a Raft log. We use NuRaft to provide the Raft replicated state machine abstraction used for fault tolerance in the atomizer, 2PC shards, and 2PC coordinators.

From Bitcoin Core, we use libsecp256k1 for BIP-340 compatible Schnorr signatures [90] which we use as our public-key signature scheme for transactions. We also use the cryptography components of Bitcoin Core to provide optimized implementations of SHA256 [71], used as the cryptographic hash function in the codebase, SipHash [5] used for hashmaps and bech32 [88, 89] used for error-correcting public key encoding. Unit and integration tests also require the GoogleTest [55] framework, but the framework is not required to build and run the main codebase.

6 Evaluation

In this section, we evaluate and compare the atomizer and 2PC architectures against our original project requirements of high throughput and low latency, ability to tolerate the failure of multiple data center regions, and performance under a variety of workloads. We also describe our benchmarking environment.

6.1 Setup

For benchmarking and testing we deployed the codebase in Amazon Web Services (AWS) using EC2 virtual servers. All servers run Ubuntu 20.04. Atomizers, shards, coordinators, watchtowers, and 2PC sentinels used c5n.2xlarge instances (8 vCPUs, 21GB RAM) whereas load generators and atomizer sentinels used c5n.large instances (2 vCPUs, 5.25GB RAM). Both instance types are virtualized so the underlying hardware is being shared by other virtual machines operated by other AWS customers. Each EC2 instance has a network interface card (NIC) that provides up to 25 Gbps to Amazon's internal network and used elastic block store (EBS) volumes for persistent storage rather than local disks.

We ran the system components in three geographical regions, Virginia (us-east-1), Ohio (us-east-2), and Oregon (us-west-2), with VPC peering connections between each region utilizing Amazon's private network rather than the public Internet. Unless otherwise stated, all 2PC shards, coordinators, and atomizers were replicated by a factor of three (one node in each region, tolerating one failure per Raft cluster), and atomizer shards replicated by a factor of two (tolerating one failure per UHS ID range). Replicated components were equally distributed between regions to simulate conditions where the entire system is geo-replicated and tolerant to the loss of an entire region. Similarly, non-replicated components such as sentinels, load generators, and watchtowers were equally distributed between regions to simulate offered load from across the United States. We modified the default TCP window size [42] settings to increase the maximum window size to account for the high bandwidth, high latency links between servers in different regions. Without this change we were unable to maximize a single TCP connection over long distances, hurting performance at the edge of the transaction throughput envelope.

Measurement and generating load. Load generators are simulated wallets that manage their own set of unspent outputs, public and private key pairs, and pending transactions. Load generators create and sign transactions and wait for confirmations from the sentinel in the 2PC architecture, or query a watchtower in the atomizer architecture. We simulate both the sender and the receiver querying transaction status separately. Latency is measured in the load generator as the time taken between the sender broadcasting a transaction and receiving a confirmation. In the 2PC architecture, load generators also record transaction throughput and the values are aggregated to produce throughput values over time for the overall system. In the atomizer architecture, the archiver calculates the transaction throughput based on the number of transactions in each block and the time between blocks. Since sentinels are not replicated and can be scaled independently of the remainder of the system components, each load generator is paired with a sentinel with a one-to-one relationship so that static transaction checks, signature validation, and conversion to compact transactions are not a bottleneck for overall system throughput and latency. Load generators start with a fixed number of outputs minted in the system and send transactions as fast as they can, limited by the speed of their virtual machine and the number of outputs available to spend due to existing pending transactions. Unless otherwise stated, load generators produce transactions with two inputs and two outputs.

If, in an experiment, Raft clusters are unable to reliably replicate data between *all* online nodes in the cluster, we discard the data point. This is to not count high throughput or low latency numbers in which data is not fully replicated as expected; we intend to show where bandwidth constraints between regions or variations in virtual machine performance prevent reliable fault tolerance for a given workload.

For peak finding, we ran sweeps with increasing load over a number of system configurations. To select the peak throughput configuration where the system was not overloaded, we only considered results where the average tail latency was below 5 seconds, with the maximum below 15 seconds, and completed successfully at least 3 times, or for the majority of test runs. Once the peak configuration was identified, we acquired at least 3 data points to plot the throughput and latency results irrespective of the individual latency values. For scalability plots,

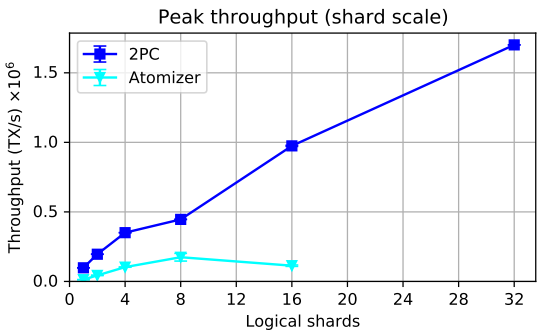


Figure 10: Peak throughput of the atomizer and 2PC architectures when varying logical shard count to be 1, 2, 4, . . . , 32.

we included all experiments that completed successfully regardless of the latency values.

6.2 Scalability

In this subsection, we consider two forms of system performance and scalability. The first relates to increased load from users in terms of transactions per second, and how varying the number of system components affects the maximum supported transaction throughput and tail latency. The second explores how increasing the size of the unspent transaction output set affects the performance metrics of both architectures. These two experiments compare how readily the architectures could support a high number of users.

6.2.1 Throughput and latency

Figure 10 compares the peak transaction throughput between the atomizer and 2PC as the number of logical shards increases. The atomizer architecture has a peak throughput of 170,000 transactions per second, beyond which adding additional shards fails to increase throughput, whereas the 2PC architecture scales linearly as the number of logical shards increases, up to 1.7 million transactions per second, though we expect peak throughput would continue to increase with more shards. The atomizer itself is the limiting factor for the overall system as all transaction notifications have to be routed via the atomizer Raft cluster. Adding more shards actually increases the network bandwidth and computation required of the atomizer leader as there are more block subscribers, as can be seen by the drop in performance between eight and sixteen shards. The leader is unable to serve both the followers in the atomizer Raft cluster and the subscribed shards and watchtowers with its available network bandwidth and compute resources. These constraints could be alleviated through an extra service purely responsible for distributing blocks to shards so that there are fewer subscribers to the leader atomizer node, or by using IP multicast. However, since the leader atomizer node has to ingest and replicate all transaction notifications from shards, there will always be a bottleneck at the atomizer cluster even if block distribution is offloaded to another service. This is the key drawback of the atomizer architecture and the cost of generating a total ordering of all transactions.

Diving deeper, Figure 11 shows the throughput and latency varying the number of clients for different shard counts for both architectures. Recall we only include data points where the system was not overloaded and the transaction data could be replicated reliably between all regions containing nodes. Often benchmarks with greater offered load succeeded when others with less load failed based on the above criteria, or there were large variations in latency between experiments using the same system configuration. We suspect that this is because of variation in the peak network bandwidth and compute available when running the benchmark in AWS, due to operating on shared hardware and network links. Since we were unable to control for these variations, many of the plots show large error bars, and may contain data points where an experiment was retried multiple times to obtain at least three results. It is possible that the variability is actually due to our system design or its implementation, but a controlled testing environment would be required to evaluate this hypothesis.

Here we see 2PC does not have a drop off in performance, supporting a greater offered load by increasing the number of shards. 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. Crucially, the 2PC architecture has no experimentally demonstrated bottleneck and can support more throughput without trading off tail latency by scaling the number of shard clusters. 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. By contrast, the atomizer architecture has a clear peak throughput plateau with 8 shards, where increasing to 16 nodes results in a drop in peak throughput.

6.2.2 Database Size

Figure 12 compares how the transaction throughput and tail latency for both architectures change as the number of unspent outputs increases, with the number of shards fixed at 8. The plot shows that the atomizer architecture can handle up to 100 million outputs with minimal effect on transaction throughput and latency. At one billion outputs, throughput suffers slightly for the same of-

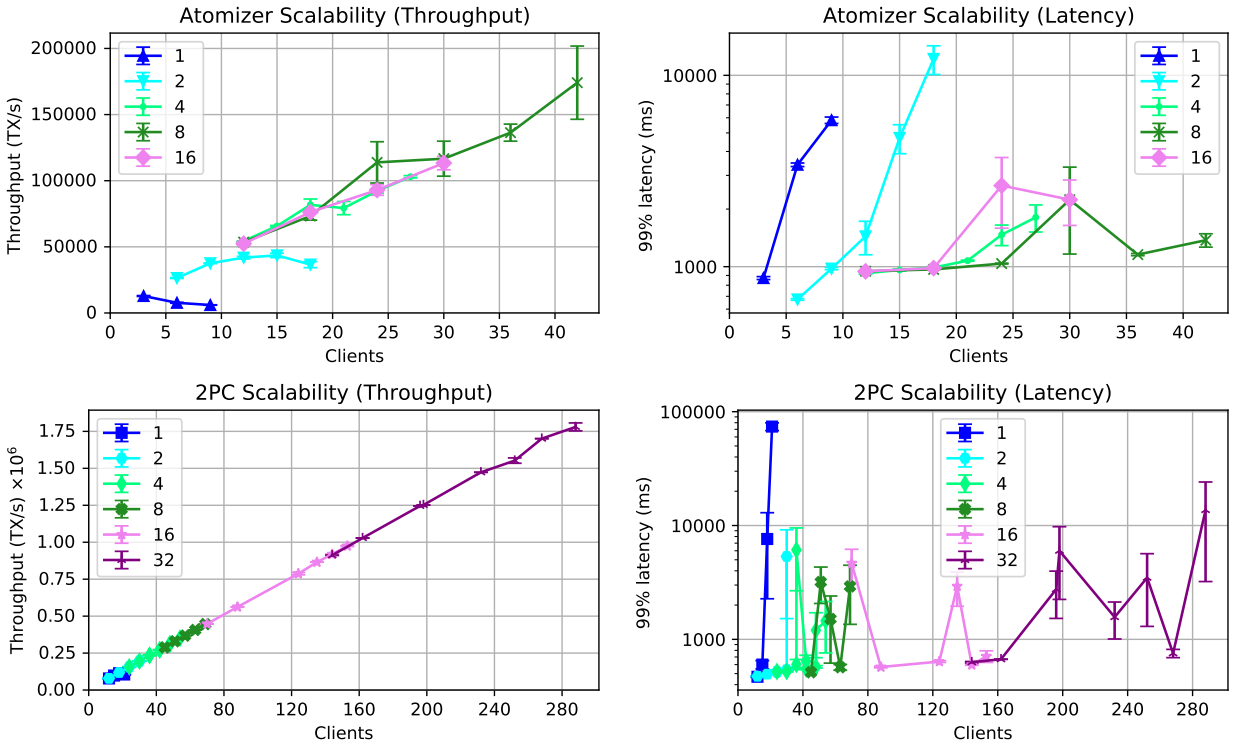


Figure 11: Atomizer and 2PC peak (considering clients) average throughput and 99% latency at peak average throughput with varying logical shard count and clients. In 2PC there are the same number of coordinators as shards.

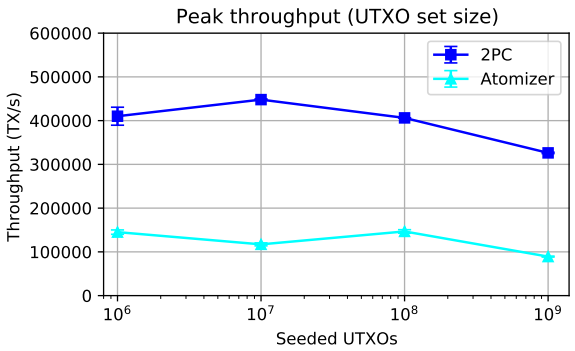
fered load. Recall that the shards must store the UHS on disk, meaning that as the size of the database grows, each lookup of a UHS ID and update of the UHS when a block is processed takes longer. Thus, peak throughput might have to be limited to support a larger number of outputs as the default atomizer architecture cannot easily accommodate more shards due to network bandwidth constraints on the leader atomizer node. Conversely, while the peak throughput decreases with a larger UHS in the 2PC architecture, it is able to scale by increasing the number of shards and thus maintain performance. Unlike the atomizer architecture, which is limited by the atomizer leader, the 2PC architecture is only limited by the performance of the shards themselves, the number of which can be increased to spread load between a greater number of shards.

The atomizer architecture may be able to accommodate a larger UHS if shards did not use an on-disk database like in 2PC. Note, however, that an in-memory only shard would not survive a crash or power failure and would need to be rebuilt completely from the archiver which may be challenging in a long-running system. The 2PC shard's state is still persisted to disk but through a sequentially written Raft log and snapshots. This method of persistence is more performant than the random reads and writes to disk needed by the atomizer architecture's shard. The 2PC shard's state machine is entirely in-memory leading to much better performance with a large number of outputs. Because of this, replicating the shards in the atomizer architecture using a Raft cluster might lead to better raw throughput for a given number of shards.

6.3 Fault Tolerance

In this subsection, we consider how the system responds to failures, such as random hardware failures, natural disasters, and network partitions. We evaluate how both architectures handle up to two regional data center failures, and the scalability of each architecture as the number of supported failures increases.

Figure 13 shows the transaction throughput over time for the atomizer architecture when two simulated data center failures occur and shards have a replication factor of three and the atomizer a replication factor of five (supporting up to two failures per cluster). At both 120 and 180 seconds into the test, an atomizer node and shard replica for each logical shard is killed to simulate two failures of entire data centers. The plot shows that the system can recover successfully and automatically restore the availability of the system in a matter of seconds. The failures cause a drop in throughput to zero for several seconds as the atomizer Raft cluster performs a leader election to select a new leader. Interestingly, we only see a dip in performance when the atomizer leader



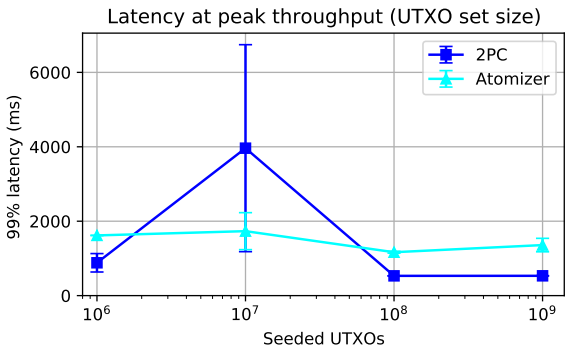


Figure 12: Comparison of 2PC and atomizer with different UHS sizes.

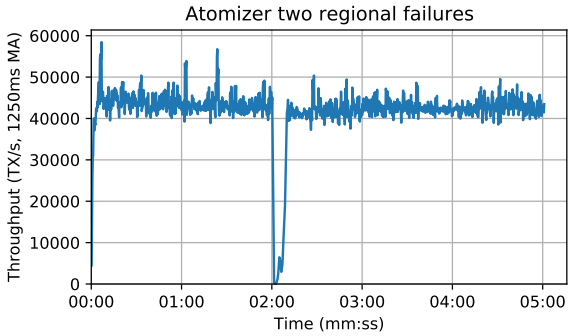


Figure 13: Atomizer architecture throughput over time with atomizer replication factor five, shard replication factor three and two whole-data center failures at 120s (leader) and 180s (follower) (see discussion for why the follower failure does not cause throughput drop). 5 sample moving average (1 sample per block).

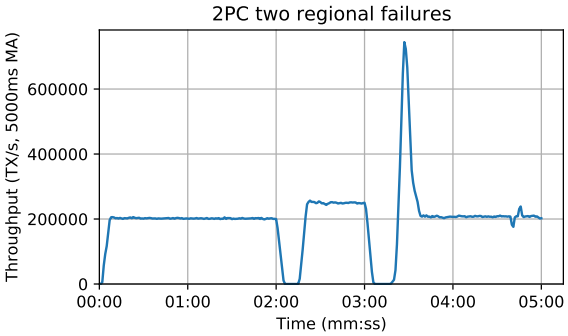


Figure 14: 2PC architecture throughput over time with replication factor 5 and 2 whole data center failures at 120s and 180s. 5 sample moving average (1 second per sample).

is killed, and the Raft cluster needs to elect a new leader, which is what happens at 120 seconds. At 180 seconds, in addition to shards, a follower Raft atomizer node is killed, which does not impact performance. Shards in the atomizer architecture do not use Raft consensus, so any sentinels previously using the failed shard simply connect to a different online shard covering the same range of UHS IDs. After the atomizer leader election has completed, the shards connect to the new leader and continue processing transactions. There is no loss of data or inconsistency in the unspent output set as a result of the failures. Load generators simply retry any transactions that were dropped by failed shards or the previous atomizer cluster leader.

The plot in Figure 14 shows the overall system throughput of the 2PC architecture over time when shards and coordinators have a replication factor of five (supporting up to two failures per cluster). To simulate continued system uptime and recovery when up to two data centers fail 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 (which comprised some leaders and some followers). The plot shows that the 2PC architecture is successfully able to handle and recover from the failure of two entire data centers with minimal loss of 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. As in the atomizer architecture, there is no loss of data from each failure and the system is not left in an inconsistent state as the replacement coordinators continue any distributed transactions that were in progress at the time of each failure.

Figure 15 compares the change in transaction throughput and tail latency between architectures as the number of supported system failures increases from zero through four. For 2PC this shows how the system performs when the replication factor of shards and coordinators increases from one through nine, the number of clusters remains fixed at eight and the offered load is increased until peak throughput is achieved. The plot shows that 2PC is tolerant to increased replication factor, showing only a modest decrease in peak throughput.

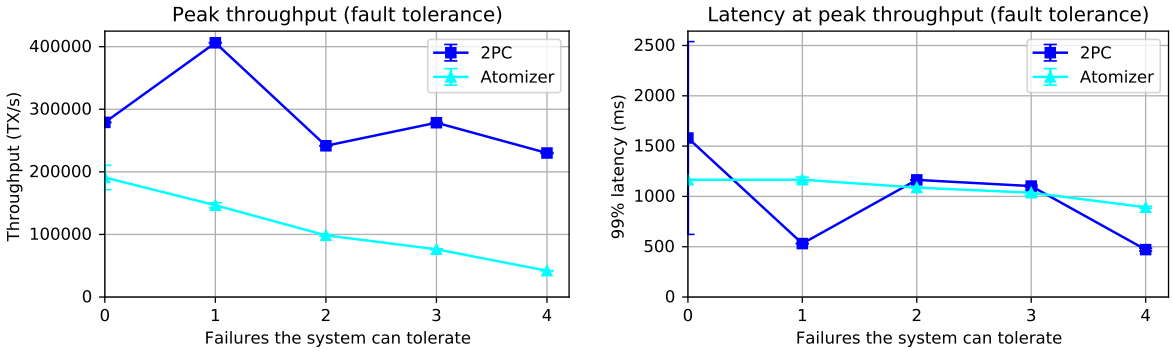


Figure 15: Throughput and 99% latency for different choices of number of faults tolerated, f. In the atomizer architecture this means $2f + 1$ atomizer replicas and $f + 1$ shard replicas. In 2PC, it means $2f + 1$ transaction coordinator replicas and shard replicas.

This suggests that, if desired, the 2PC architecture may be able to support a high number of simultaneous failures.

For the atomizer architecture, the shard replication factor is increased from one through five and the atomizer cluster from one through nine, showing peak throughput decrease. Since the atomizer is the bottleneck in the system, increasing the replication factor of the atomizer cluster results in increased bandwidth requirements on the leader atomizer node causing a decrease in peak throughput. Increasing the replication factor of shards also results in more bandwidth utilization on the leader atomizer. As explained previously for the shard scaling plot in Figure 10, the leader must broadcast the latest blocks to a larger number of subscribers as each shard replica receives all blocks. The atomizer architecture is therefore less tolerant to increased redundancy than 2PC due to bandwidth constraints on the leader atomizer node.

6.4 Workload Variability

This subsection compares how both architectures perform under varying transaction workloads from users. We vary the proportion of transactions with a high number of input and outputs, and the proportion of double-spending transactions. We are unsure how the transaction workload will look in practice, however, for Bitcoin we found that over 75% of transactions consist of one input and two outputs, or vice versa.

6.4.1 Transaction Size

Figures 16 and 17 compare how the proportion of transactions sent with a high number of inputs and outputs, respectively, affect the throughput and latency between architectures. In this test, the proportion of transaction load sent to the system with eight rather than two inputs/outputs was increased from 0% through 30%. The benchmarks were conducted using a database containing 1 billion UHS IDs.

As the number of inputs per transaction increases, the peak throughput drops in the atomizer architecture. This is because inputs must be checked by the shards to ensure they are unspent and aggregated within the atomizer to ensure all outputs have been attested to by shards. Since more shards on average are required to attest to a transaction with a larger number of inputs, more data must be replicated by the atomizer cluster. There is also a higher probability that it will take multiple blocks before all required attestations have been accumulated for a transaction in the atomizer.

Conversely, the increase in output count in the atomizer architecture exhibits only a minor loss in throughput and increase in latency because outputs are not the limiting factor for the atomizer to process transactions. Our transaction format guarantees unique output UHS IDs if the transaction is valid, so as an optimization the atomizer and shards are not required to check them. Therefore, additional outputs only increase the size of blocks and transaction notifications, and thus the network bandwidth requirement between atomizers and shards.

For the 2PC architecture, as the proportion of large transactions (inputs or outputs) increases, the peak throughput decreases as the system becomes overloaded. This is similar to increasing the number of clients offering two-input, two-output transactions. Ultimately the system is limited by the overall number of UHS IDs being processed, regardless of how they are grouped into transactions. Tail latency is largely unaffected by the transaction size as latency is dominated by Raft replication delays rather than the lookup time in the state machine for each UHS ID. As a result, in production environment it may be necessary to over-provision the number of shard clusters to absorb workloads with a high proportion of large transactions, or discourage large transaction via other means.

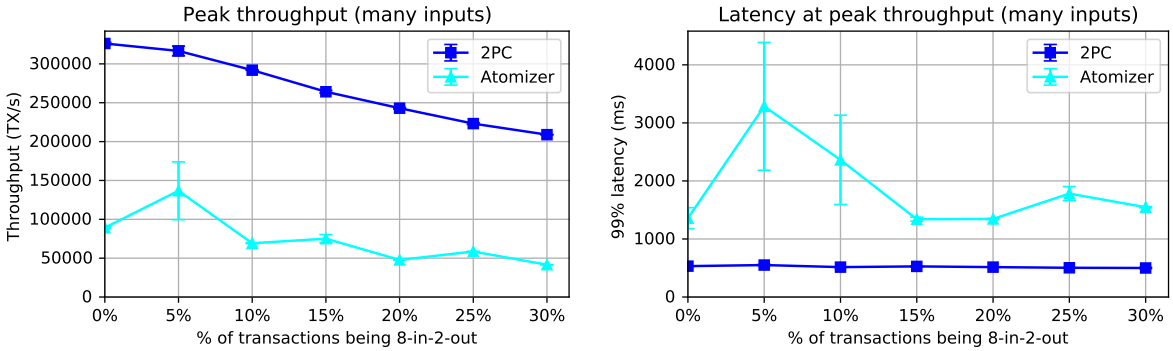


Figure 16: Throughput and 99% latency varying the proportion of transactions with eight inputs and two outputs.

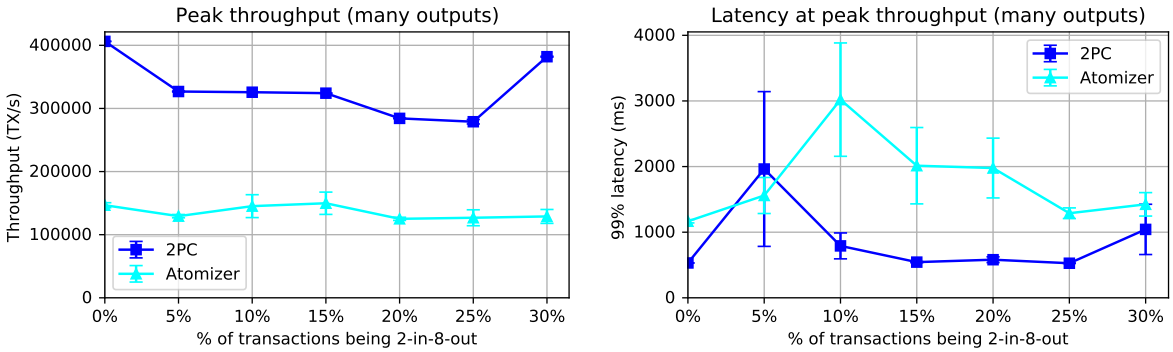


Figure 17: Throughput and 99% tail latency varying the proportion of transactions with two inputs and eight outputs.

6.4.2 Double Spends

Figure 18 compares how the transaction throughput and latency of valid transactions changes between architectures as the proportion of double-spending transactions sent from the load generators is varied between 0% and 30%. The load generators send double-spending transactions by storing previously confirmed transactions and re-issuing them at a later time. This ensures the inputs to the transaction are either not present in any shard's UHS or are present in the atomizer's spent transaction output cache. Only the throughput and latency of valid transactions are included in the plot. After sending a double-spend transaction, there is an artificial delay within load generators to simulate the additional time it takes to generate new valid transactions.

Double-spends do not greatly affect the throughput and latency of valid transactions in the atomizer architecture. This is because most double-spends are trivially caught at the shard layer so that additional load is not put on the atomizer cluster. Double-spends negatively affect the peak throughput of valid transactions in 2PC because each transaction, valid or not, has to be replicated as part of a distributed transaction batch. This requires shards to replicate all transactions as part of the lock phase, and the coordinators have to replicate the status of all transactions, so double-spends cause the same load as valid transactions. Absorbing an increased proportion of double-spending transactions in 2PC while maintaining the same load of valid transactions could be achieved by increasing the number of shards and coordinators. It may be more difficult to scale the atomizer architecture to absorb more double-spends by adding shards because of the increased load additional shards put on the atomizer cluster as shown in Figure 10.

7 Related Work

Central banks around the world are in a wide variety of stages with regard to CBDCs. Some are in research and development phases while others are running pilots and even launching products to the public. China's e-CNY is currently in public trials [30, 61, 77] and is a centralized system based on the UTXO model. e-CNY involves a two tier model and does not support end-user custody. On a smaller scale, the Central Bank of the Bahamas has launched a two tier CBDC, the Sand Dollar [29], which is built on the NZIA Cortex DLT platform. The Central Bank of Nigeria has launched eNaira [28], a two tiered system based on Bitt's DCMS platform. Some projects are in pilot phase such as the Eastern Caribbean Central Bank's DCash [40] system which is also based on Bitt's platform.

Other projects are in research and development phases such as the Riksbank's e-krona project, built on R3's Corda Enterprise Blockchain platform [78], which re-

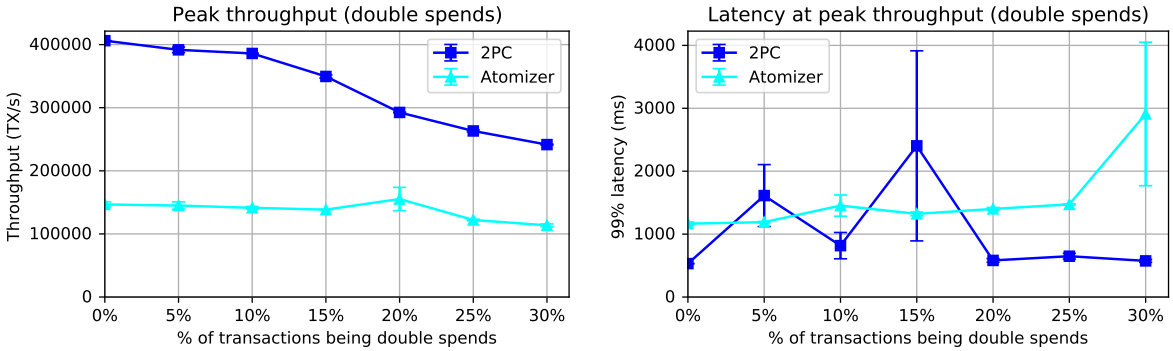


Figure 18: Throughput and 99% latency of valid transactions varying the proportion of transactions with double-spending inputs.

quires all transactions go through a single notary to enforce double-spend protection. This creates a similar scaling bottleneck to our atomizer architecture [81]. Several projects have achieved linear scalability with a parallelized architecture. Eesti Pank along with several other banks in the Eurosystem, have tested a CBDC design based on tracking groups of bills using a set of parallelized blockchains [46]. While it achieves linear scalability, transactions involving multiple bills require external coordination. No internal guarantee of atomicity for these transactions is provided.

Several central banks already support real-time gross settlement (RTGS) and fast payment systems [6]. These systems are designed to settle transactions between eligible financial institutions with low latency. In practice, these systems do not handle a volume of traffic representative of a national retail payment system nor do they provide direct access to the public [7, 47, 48, 74]. Allen et. al. identify these and other technical and legal issues related to CBDC design [2].

The Bank for International Settlements together with a group of seven central banks outlined [9] some of the tradeoffs between privacy, interoperability, resilience and other topics but do not propose a potential design. The Regulated Liability Network [36] from SETL and Amazon AWS presents a CBDC design which claims to achieve 1 million transactions per second utilizing multiple coordinated blockchains. However, the paper does not discuss deployment across multiple geographic regions which is vital for resiliency, and does not provide transaction latency figures.

Hamilton borrows ideas from both cryptocurrency and electronic cash designs. Hamilton uses the UTXO transaction model first used in Bitcoin and stores state as unspent coins [67]. Unlike Bitcoin, Hamilton operates in a model of centralized trust. Our transaction flow diverges from Bitcoin because the complete ledger is not publicly available to users, and the transaction processor only stores transaction hashes to reduce stored information [49]. Output data is blinded in the process of generating UHS IDs and the transaction processor does not store the output data itself. As a result, we introduce an interactive transaction protocol that relies on the sender of funds sharing the output data and identifier with the recipient. In Bitcoin, all parties can independently verify the success of a transaction by checking if it is included in a block which is not practical at scale as described in §4.4. We address this issue in the atomizer architecture with the addition of a watchtower where senders and receivers can verify transaction success. In the 2PC architecture, senders and receivers learn of success directly through the shards. Another design option might be for payers to send recipients cryptographic proofs of transaction inclusion, for example by using something like SkipChain [70], so recipients do not need a query service.

Hamilton's 2PC architecture uses a variant of two-phase commit [57] which does not need to support rollbacks. Like Google's Spanner [35], it uses a combination of two-phase commit with a replicated state machine (in this case, Raft [73]), but does not support general SQL. Narwhal/Tusk [37] is a consensus algorithm which commits to hashes of transaction sets using a DAG but does not present a full-featured state machine nor transaction system. It might be possible for Hamilton to use this instead of Raft for improved performance but it is not clear how a deterministic transaction execution state machine would be built that could take advantage of the increased consensus performance.

Chaumian eCash [33], and designs based on it [23, 26, 27], also 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. The Swiss National Bank's CBDC [34] project expands upon Chaum's model by proposing epoch windows in which coins must be spent. This addresses the issue of maintaining an evergrowing list of spent coins by pruning older entries, but imposes a

new requirement on users who might not be familiar with money that cannot be used across epochs, and has significant policy implications. Many of these schemes strive to achieve unlinkability, with mixed success against colluding attackers, while Hamilton does not. It is unclear what level of performance these schemes can achieve in practice, since few of them have been implemented.

Unlike most CBDC research efforts to date, the Hamilton project is open source. This allows results to be independently reproducible and fosters collaboration with external parties on continuing research. It also encourages global interoperability standards and provides a much lower barrier to adoption.

Contrary to other projects proposing backed stablecoin designs [16, 31, 39, 82], Hamilton is designed to be administered directly by the central bank or a related entity, and transacts in central bank liabilities.

8 Discussion

Phase 1 of Project Hamilton has identified several key results which challenge preexisting technical design assumptions, and highlight several open questions to be explored in future phases of the project. We discuss our learnings and opportunities for future research below.

8.1 Key Results

CBDC design choices are more granular than commonly assumed. Existing research often assumes that blockchain or distributed ledger technology is required to implement many of the desirable features for a CBDC, or makes broad suppositions about the capabilities of particular data models, such as so-called "token-based" and "account-based" models [3, 12, 65]. We found these limited categorizations lacking and insufficient to surface the complexity of choices in access, intermediation, institutional roles, and data retention in CBDC design [53]. Our research identified several key design choices that would need to be made. For example, the CBDC's trust and threat model, transaction format, and fault-tolerance and scaling strategy, the primary choices explored by this phase of research, present a range of potential options that affect user experience. Future research into auditability, tamper-resistance, spam prevention, programmability semantics, and privacy are among the most important design choices which have been left to future research.

CBDCs can adopt a wide variety of design characteristics depending on public policy objectives and system performance demands. Robust technical research and experimentation is required to inform policymakers as to the wide variety of technical capabilities and tradeoffs. Equally, clear public policy objectives and product design decisions are required to inform the appropriate technical design for the system. As a result, at this stage of CBDC research, it is important that policy and technical research are not conducted in isolation from each other.

Techniques from cryptography, distributed systems and blockchain technology can be combined to provide unique functionality and robust performance. By leveraging classical distributed computing algorithms, we implemented a highly scalable CBDC platform while supporting a Bitcoin-like transaction format. Without implementing a blockchain, our two-phase commit architecture supports both intermediation and self-custodying user wallets, and eliminates single points-of-failure to provide geographic fault tolerance. Our system also supports a range of potential privacy options by not requiring central storage of user balances or identities. The atomizer architecture uses a globally-ordered sequence of transactions grouped into batches, similar to a blockchain, which potentially provides better support for auditability in the future. However, generating the transaction sequence in the atomizer architecture limits its scalability potential compared to the two-phase commit architecture.

Using a Byzantine fault tolerant (BFT) single state machine approach might cater for an unnecessarily strong threat model if the central bank directly operates the CBDC. Systems in which transaction validation and execution are distributed among multiple separate entities, such as in cryptocurrencies like Bitcoin and permissioned chains like the proposed Diem blockchain, can be implemented as replicated state machines using distributed consensus algorithms which provide Byzantine agreement or full Byzantine fault tolerance. This approach allows such systems to tolerate malicious nodes when multiple mutually untrusted parties participate in settling transactions and defining system rules. In a central bank operated CBDC, only the central bank settles transactions and defines the system rules so there is no requirement to expect malicious nodes under normal operation. If the CBDC is not operated directly by the central bank, and instead via multiple, distrusted third parties, a distributed BFT-based approach may be a better solution. We leave exploring this option to future work. Byzantine fault tolerant algorithms such as HotStuff [1] might still be useful to protect against bugs or compromised components as a drop-in replacement for Raft, the non-BFT consensus algorithm already used for this project.

Executing all transactions via a single-threaded state machine, whether generating a blockchain-like data structure or not, prevents horizontally scaling the maximum throughput of the system by adding more nodes. Our research was unable to partition the atomizer service, which must be scaled vertically using additional network bandwidth and processor speed for an increase

in maximum transaction throughput. Vertical scalability is more difficult to achieve than horizontal scalability because improvements in network bandwidth and processor speed occur over long timeframes and have increasingly plateaued in recent years. However, it may be impossible to avoid a limited capacity to scale for increased throughput if materializing a total ordering of all transactions proves to be the best method for implementing tamper-detection and programmability, important questions for future research. By contrast, depending on the workload, a traditional partitioned database implementation can scale horizontally to accommodate a greater maximum transaction throughput by adding more nodes to the system. In our specific data model, funds are uniformly distributed across partitions and transactions can require the participation of multiple partitions, but we expect most transactions will only reference a small number of unspent outputs relative to the total number of partitions. Other data models, such as accounts, may reduce the maximum number of partitions involved in a transaction and make the cross-partition workload more predictable.

It is challenging to implement a non-interactive payment protocol while maintaining user-to-user privacy. In public cryptocurrencies, transactions are visible to all parties making it easy for a user to independently discover whether they have received a payment under certain conditions. If transactions use standardized encumbrances, the recipient of a payment can identify funds they can spend by searching all transactions settled by the system for encumbrances they can satisfy. Public visibility of all transactions is unlikely to be a desirable feature for a CBDC due to user-to-user privacy concerns. Although some cryptocurrencies use cryptography to obfuscate or hide the transaction participants and values from observers, the volume of transactions settled by a CBDC may be too great for a user to check every transaction to determine whether they have received a payment. Since the transactions executed by the system are not broadcast to all users, the sender and recipient have to communicate with each other either directly or via a third party as part of the transaction protocol to provide a payment notification. Public cryptocurrencies allow for non-standard encumbrances which also require user-to-user communication to provide a payment notification, but this is uncommon in practice, and our system requires out-of-band notification for all transactions regardless of whether using a standard encumbrance. Third parties included in the transaction protocol would be useful if both the sender and receiver are not online at the same time, and could be the central bank itself or external service providers. Zero-knowledge proofs might make it possible to publish all transactions executed by the system without compromising user-to-user privacy, an interesting area of future research.

The central bank does not need to retain all transaction information to implement a secure CBDC system. We show that the central transaction processor only needs to store commitments to unspent funds, as opaque 32 byte hashes. This limits data retention by the central bank, which is appealing, but makes self-custody more operationally challenging for users, and the system harder to audit internally. Our data model only stores cryptographic commitments to unspent funds at the central bank and discards the underlying preimage of the commitment required to spend, in order to limit data retention at the central bank. In order to spend funds, the user must provide the preimage of the commitment with their transaction so it can be validated by the sentinels. Therefore, the sender of a payment must provide the recipient with the preimage required to spend the money before the transaction can be considered complete. The preimage must be retained by the user until they spend their funds, as it cannot be recovered if lost, and without it the sentinels cannot check whether the transaction is valid. The task of storing transaction data and communicating it between users could be conducted by a third party. However, the third party would have access to the transaction data of participating users. Zero-knowledge proofs have the potential to hide transaction data from sentinels, eliminate the need for direct communication between transacting parties, and enable internal system auditing.

8.2 Future Work

This paper demonstrates a high-performance, fault-tolerant CBDC implementation. However, we have not yet explored all design considerations for a practical CBDC deployment. Some ideas for future areas of research and implementation are presented below. We plan to investigate many of these research topics in future phases of Project Hamilton.

Privacy and auditability The UHS is a powerful data model enabling transaction validation to be fully decoupled from the database layer of the system. It also minimizes data retention in the core system, and opens the possibility of zero-knowledge sentinels which would hide transaction data and greatly increase user privacy from the central bank. However, only storing commitments to the underlying data makes the system difficult to audit for correctness of transaction execution, the total supply of money, and intrusion detection. Furthermore, it is unclear how to balance user privacy from the central bank and the desire of law-enforcement to access transaction data.

Programmability Our current transaction format and data model restricts programmability features to those

which can be implemented using transaction-local validation. Transactions are deterministic in that they must provide all state elements that will be mutated prior to transaction execution, and fully specify the state transition should the transaction complete. It is unclear whether these two restrictions affect the space of contracts that can be implemented. In either case, the UHS makes contract engineering more difficult than a model which supports non-deterministic transactions, so the performance of a system implementing such a model will need to be evaluated.

Interoperability To support further innovation on top of the CBDC, techniques for interacting with cryptocurrencies and existing payment solutions in the traditional financial sector will need to be researched. We are confident that our designs could support interoperability with cryptocurrencies via Layer-2 payment channel networks, though specific implementation details still need to be determined. It is unclear whether the CBDC will need to directly support formal standards used by payment platforms in the banking sector, or whether interoperable functionality could instead be delivered by third parties. Easier cross-border payments are often cited as an important policy goal for a CBDC, and our designs support such payments if users from multiple countries are able to directly use the CBDC. Techniques for cross-border payments between separate CBDCs will depend largely upon how CBDCs from other countries are designed.

Offline payments We have not yet explored the potential for payments using CBDC without an Internet connection. Our transaction format and data model requires interactive communication between the central bank and both transacting parties. One option is to operate a parallel system using trusted hardware requiring no connectivity with the central bank to conduct a transaction. Trusted hardware would be responsible for enforcing the authenticity of CBDC while outside central bank systems, and thus vulnerable to supply chain attacks or end-user tampering. Alternatively, radio, satellite or mesh networks could be exploited to retain connectivity with the central bank during an Internet outage.

Minting and redemption Our experiments assumed the entirety of the CBDC in circulation was already present in the system. In practice, CBDC will need to be minted or removed from circulation depending on the flow of money into and out of the system. We have yet to explore how best to implement changing the supply of CBDC while maintaining security against both insider attacks and external adversaries.

Productionization Our designs are fully fault-tolerant against multiple geographic data center failures, ensuring high-availability while preventing data loss. However, the implementation has not been hardened or tested for long-term, production-level readiness. Our evaluation focused on measuring peak performance for short periods of time under high load with a static number of system components. We did not evaluate system performance over extended periods of time, where supporting a large UHS may require a greatly increased number of shards. Scaling the number of shards and rebalancing UHS IDs between them may have performance implications need to be fully investigated. We also do not provide an implementation for various important production processes such as system health monitoring, shard rebalancing, and automated component scaling.

Denial of service attacks Our designs support self-custody of private keys, and we assume there are no fees per transaction in the base layer, making the system vulnerable to denial-of-service attacks. Adversaries could submit large volumes of invalid or valid transactions at no cost, consuming central bank resources and degrading system performance for legitimate users. Rate-limiting and spam-prevention techniques (aside from fees) could mitigate this risk. Options include network-level throttling, enforcing a cool-off period before money can be re-spent, charging nominal fees past a certain transaction volume threshold, or requiring a proof-of-work per transaction.

Quantum resistance If large-scale quantum computers are built, most cryptographic systems powering today's Internet, e-commerce, and finance could eventually be at risk. This stems from the fact that these systems rely on cryptographic primitives that are vulnerable to quantum adversaries. However, standards bodies, such as NIST [72], are developing a portfolio of cryptographic primitives resistant to classical, quantum, and hybrid attacks. This is a highly mature effort and is expected to yield final selections in the not-too-distant future. The cryptographic primitives used in Hamilton are either post-quantum with minimal modifications (e.g., hash functions, where post-quantum resistance can be obtained by a suitable increase in parameters), or can be replaced with a standardized post-quantum alternative, once one is available. Similarly, the extensions of Hamilton that we have identified, e.g., a privacy-enhanced version, use cryptographic primitives for which post-quantum alternatives are known. Transitioning to post-quantum systems will be an industry-wide effort. We are confident that Hamilton is well-prepared for such a transition and can remain a long-term secure system in a post-quantum world.

9 Conclusion

CBDCs are being considered widely by central banks and technical research is critical to understand what is feasible, identify interdependencies between technical

and policy choices, and discover novel approaches to achieving goals for a CBDC.

Our research presents a CBDC transaction processor design, implements two potential architectures to support transactions at scale, and high performance and resilience. We find that technical and policy choices are highly interdependent and that these choices are more granular and with more permutations than commonly discussed. Our work is limited to the transaction processor component of a CBDC and, as a research platform, is neither designed to launch a CBDC or address all potential requirements. Further research is needed in a wide range of technical areas and how these different technical options impact desired policy outcomes.

Through software design, development, and testing, Project Hamilton provides unique insight into technology relevant to implementing a CBDC. By designing a flexible research platform and issuing an open-source license for the software, the Project Hamilton team hopes to share its learnings with others and receive feedback and contributions to the code from other digital currency experts.

This open-source release concludes Phase 1 of Project Hamilton. The flexible core infrastructure developed in Phase 1 was designed to support future research and development with various potential designs and features. In Phase 2 of Project Hamilton, the Boston Fed and MIT DCI will continue their CBDC infrastructure research and explore different options and configurations in areas such as data privacy, programmability, and interoperability. The team will assess how these choices impact a platform's technical design and performance.

As the global CBDC discussion evolves and the Federal Reserve's research continues, Project Hamilton aims to continue providing valuable insights to policymakers and the general public through its experimentation with leading-edge technical research.

10 Acknowledgements

The authors express gratitude to Robert Bench, Jim Cunha, Ken Montgomery, and Eric Rosengren for their leadership and direction in this work. In addition, we thank Robleh Ali, Jonathan Allen, Spencer Connaughton, Thomas Cowan, Tadge Dryja, Rob Flynn, Kristin Forbes, Shira Frank, Nikhil George, Gert-Jaap Glasbergen, Ethan Heilman, Simon Johnson, Sean Neville, Ronald L. Rivest, Bernard Snowden, Michael Specter, Sam Stuewe, Robert Townsend, Reuben Youngblom, and staff at the Federal Reserve Board for their helpful contributions, feedback, and comments. We are also grateful to the funders of the Digital Currency Initiative for their ongoing support of the MIT researchers that participated in this work.

References

[1] I. Abraham, G. Gueta, and D. Malkhi. Hot-Stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069, 2018.

[2] S. Allen, S. Čapkun, I. Eyal, G. Fanti, B. A. Ford, J. Grimmelmann, A. Juels, K. Kostiainen, S. Meiklejohn, A. Miller, et al. Design choices for central bank digital currency: Policy and technical considerations. Technical report, National Bureau of Economic Research, 2020.

[3] R. Auer and R. Böhme. The technology of retail central bank digital currency. *BIS Quarterly Review*, March 2020.

[4] 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/.

[5] 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.

[6] 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.

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

[8] Bank for International Settlements. CBDCs: an opportunity for the monetary system. *BIS Annual Report Economic Report 2021*, pages 65–91, 6 2021.

[9] Bank for International Settlements et al. Central bank digital currencies: System design and interoperability, 9 2021. https://www.bis.org/publ/othp42_system_design.pdf.

[10] 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.

[11] 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.

[12] Bank of Thailand. Central bank digital currency: The future of payments for corporates, 2021. https://www.bot.or.th/English/FinancialMarkets/ProjectInthanon/Documents/20210308_CBDC.pdf.

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

[14] 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.

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-Wesley Reading, 1987.

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

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

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

[19] 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.

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

[21] 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.

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

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

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

[25] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh. Zether: Towards privacy in a smart contract world. In *Proceedings of the 24th International Conference on Financial Cryptography and Data Security*, FC '20, 2020. ePrint: https://eprint.iacr.org/2019/191.

[26] 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.

[27] 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.

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

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

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

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

[32] M. M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. P. Jones, and P. Wadler. The extended UTXO model. In *International Conference on Financial Cryptography and Data Security*, pages 525–539. Springer, 2020.

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

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

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

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

[42] ESnet. Linux tuning. https://fasterdata.es.net/host-tuning/linux/.

[43] 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.

[44] Ethereum Developers. Solidity, the smart contract programming language. https://github.com/ethereum/solidity.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] M. Fleder and D. Shah. I know what you bought at Chipotle for $9.81 by solving a linear inverse problem. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, volume 4, pages 1–17, 2020.

[51] B. I. Galler and L. Bos. A model of transaction blocking in databases, 1983. https://www.sciencedirect.com/science/article/pii/0166531683900123.

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

[53] R. Garratt, M. J. Lee, B. Malone, and A. Martin. Token- or Account-based? A digital currency can be both. Liberty Street Economics, 2020. https://libertystreeteconomics.newyorkfed.org/2020/08/token-or-account-based-a-digital-currency-can-be-both/.

[54] G. Gerdes, C. Greene, X. M. Liu, and E. Massaro. The 2019 Federal Reserve payments study, 2019.

[55] Google. GoogleTest. https://github.com/google/googletest.

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

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

[58] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[59] K. Hill. How Target figured out a teen girl was pregnant before her father did, 2012. https://www.forbes.com/sites/kashmirhill/2012/02/16/how-target-figured-out-a-teen-girl-was-pregnant-before-her-father-did/.

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

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

[62] 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.

[63] 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.

[64] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558565, jul 1978.

[65] T. Mancini-Griffoli, M. S. M. Peria, I. Agur, A. Ari, J. Kiff, A. Popescu, and C. Rochon. Casting light on central bank digital currency. *IMF Staff Discussion Note*, 8, 2018.

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

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

[68] A. Narayanan and J. Clark. Bitcoin's academic pedigree. *Communications of the ACM*, 60(12):36–45, 2017.

[69] 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.

[70] 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.

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

[72] NIST. Post-quantum cryptography, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography.

[73] 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.

[74] 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.

[75] 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.

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

[77] 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.

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

[79] K. Shirriff. Hidden surprises in the Bitcoin blockchain and how they are stored: Nelson Mandela, WikiLeaks, photos, and Python software. http://www.righto.com/2014/02/ascii-bernanke-wikileaks-photographs.html.

[80] Statoshi.info. Bitcoin unspent transaction output set. https://statoshi.info/d/000000009/unspent-transaction-output-set?orgId=1&refresh=10m.

[81] 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.

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

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

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

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

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

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

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

[89] 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.

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

[91] YCharts. Ethereum chain full sync data size. https://ycharts.com/indicators/ethereum_chain_full_sync_data_size.