Proof-Carrying CRDTs Allow Succinct Non-Interactive Byzantine Update Validation

Nick Marx
nick.marx@student.kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Florian Jacob
florian.jacob@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Hannes Hartenstein
hannes.hartenstein@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

Conflict-free replicated data types (CRDTs) are distributed algorithms that enable concurrent queries and updates without coordination with other processes, but still provide eventual consistency. In crash fault environments, processes adhere to the protocol and only send valid CRDT updates. But in Byzantine environments, processes need to individually ensure the validity of all updates. As the validity of an update may depend on all previous updates, Byzantine-tolerant CRDT processes typically must keep a grow-only set of all applied updates. We apply the concept of proof-carrying data to CRDTs to enable succinct non-interactive validation of updates, i.e., processes neither need knowledge of all previous updates nor coordination. Such proof-carrying CRDTs allow update validation in constant time and space, even in Byzantine environments. In a case study, we implemented and evaluated the performance of two proof-carrying CRDTs: a straightforward increment-only counter, as well as an update history CRDT based on the Matrix group communication system. We conclude that proof-carrying data has achieved practical relevance for applications in Byzantine CRDTs.

CCS Concepts: • **Computing methodologies** → *Distributed algorithms*; • **Information systems** → *Data structures*; • **Security and privacy** → **Distributed systems security**; • **Software and its engineering** → **Access protection**.

Keywords: Conflict-Free Replicated Data Types, Byzantine Fault Tolerance, Proof-Carrying Data, Zero-Knowledge Proofs

ACM Reference Format:
Nick Marx, Florian Jacob, and Hannes Hartenstein. 2025. Proof-Carrying CRDTs Allow Succinct Non-Interactive Byzantine Update Validation. In *12th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3721473.3722142

PaPoC '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1558-7/2025/03
https://doi.org/10.1145/3721473.3722142

1 Introduction

While the validity of an update of a conflict-free replicated data type (CRDT, [22]) is a prerequisite to apply the update, the validity itself is an application-specific notion. For example, a counter update may be valid only if it is syntactically correct, increases the value by one, contains an identifier of the counter to which the update would be applied, and is signed by its creator. Update validity especially includes update authorization. If an update is deemed valid by one correct process, it must be deemed valid by all correct processes to ensure CRDT convergence. Thus, update validity can be defined by any deterministic, wait-free [14] algorithm that only depends on the update itself and the set of previous updates, but not on their reception order or process coordination. In crash fault environments, processes can assume that updates are valid. In Byzantine environments, processes must independently validate updates as the sender could be Byzantine, raising the *problem of Byzantine update validation.* A typical solution is that processes record *update history* as grow-only, partially-ordered set (c.f. Fig. 1), even if unnecessary for CRDT queries. Requiring the full history of an update for its validation, as shown in Fig. 2, may take a long time. However, a timely validation and application of the newest updates may be crucial, e.g. in chat systems. This raises the need for a *succinct, non-interactive solution* to the Byzantine update validation problem, i.e., using no additional communication and little additional data.

The Matrix group communication system [1] serves as practical motivation for the Byzantine update validation problem. Matrix revolves around a per-group Byzantine-tolerant CRDT for update history [17, 18]. To ensure authenticity and integrity, updates are signed and stored as hash-linked directed acyclic graph (HashDAG). All prior updates have to be replicated and validated before the newest, most relevant updates can be validated, which makes group joins slow and expensive, and prevents garbage collection.

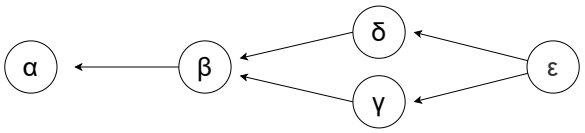


Figure 1. An update history stored as hash-linked DAG: Vertices are updates, edges are hash links to direct predecessors.

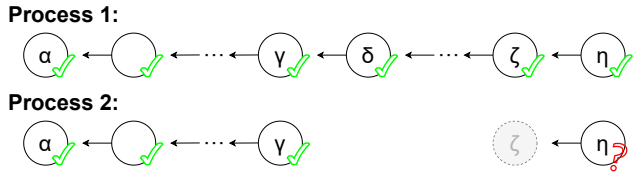


Figure 2. Two CRDT processes sharing a long history of consecutive updates. Process 1 sees and validated (green checkmark) all updates from α to η. Process 2 has only seen and validated updates α to γ. Update η is visible, but not its predecessor ζ. To validate η traditionally, all updates in its history need to be validated first. A PCD proof allows succinct validation of η, i.e., without full history.

More than 100 million users take part in the public Matrix federation on more than 100 000 servers [15] — especially long-running or busy communication groups can benefit from succinct, non-interactive Byzantine update validation. HashDAGs can be formalized as delta-state CRDTs [18], and are prevalent for making operation-based CRDTs Byzantine-tolerant [21]. In Matrix, HashDAGs serve as Byzantine-tolerant communication layer for operation-based CRDTs on top, e.g., to store a chat room's title, membership, and chat messages.

Proof-carrying data (PCD) systems allow data exchanged between processes in a distributed algorithm to carry a cryptographic proof that *every* previous step in the computation satisfied some safety properties [12]. PCD systems were prohibitively expensive due to recursion of succinct non-interactive arguments of knowledge (SNARKs), until recent constructions based on polynomial commitment schemes [7] were proposed. These efficient PCD systems are already used in a blockchain for succinct, non-interactive block validation [8, 26]. Similarly to using PCD for the validation of blockchain blocks totally ordered by consensus, in this paper, we use PCD for validation of CRDT updates partially ordered by eventual consistency.

We start in Sec. 2 with our proof-carrying CRDT concept. In Sec. 3, we demonstrate how to augment an increment-only counter CRDT with PCD to enforce a Byzantine validity property that would otherwise require the full update history. In Sec. 4, we show a Byzantine-tolerant proof-carrying CRDT based on the Matrix communication CRDT. We analyze the asymptotic performance of our concept and empirically measure the performance of our implementations in Sec. 5. We point to related work in Sec. 6, and conclude in Sec. 7.

2 Concept: Proof-Carrying CRDTs

We present proof-carrying CRDTs as a succinct, non-interactive solution to the Byzantine update validation problem. We say that an *update history* is a partially ordered set of updates including their causal predecessors. An update is a *direct* predecessor of another update if no update is ordered in between. The proof-carrying CRDT concept is straightforward:

1. Define update validity as a function of update history.
2. On update creation, generate a cryptographic proof of update validity that accompanies the update.
3. On update reception, verify the carried proof.

PCD systems enable proofs of validity that neither require knowledge of the full history nor coordination with other processes, and *Byzantine update creators cannot forge correct-looking proofs for invalid updates*. Therefore, verifying a PCD proof is wait-free and tolerant against Byzantine faults, matching Byzantine-tolerant CRDTs. Like Byzantine-tolerant CRDTs, PCD is limited to equivocation-tolerant validity notions, i.e., update validity must not depend on concurrent updates. Due to the recursive nature of PCD, creating a validity proof for an update only requires its direct predecessors with their proofs. Since the veracity of a PCD proof is deterministic, correct replicas make the same validity decision on a given update. Thereby, proof-carrying CRDTs decouple update creation and validation from knowledge of the update's full history.

2.1 Byzantine-tolerant CRDTs

Conflict-free replicated data types (CRDTs) are a class of asynchronous, wait-free replication algorithms [2, 29]. Processes query and update their local state independently and without coordination. A distributed algorithm is wait-free if its processes execute computations without depending on others, even under partition [14]. This minimizes latency and maximizes fault tolerance [20]. Despite concurrent updates, CRDTs guarantee *eventual consistency*, which consists of *eventual visibility* (an update visible for one correct process is eventually visible for all correct processes) as liveness property and *convergence* (two correct processes seeing the same update set are in the same state) as safety property [3].

Due to their suitability for Byzantine environments via equivocation tolerance [16, 18], we focus on state-based and delta-state approaches to CRDT replication [2]. The state space of state-based CRDTs is formalized as a join-semilattice: any two valid states have a least upper bound that is also a valid state. Processes periodically gossip their own state as updates, and use the least upper bound to merge received states with local state [27]. It is critical to validate received state, so that the merged state is also valid. In delta-state CRDTs, updates may be reduced to deltas that still lead to the desired state. For example, a state-based grow-only set CRDT gossips the full set every time, but its delta-state variant also gossips sets that only contain new elements.

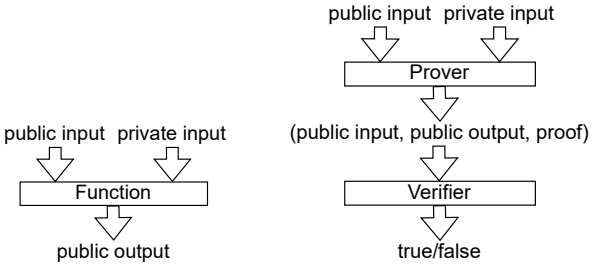
2.2 Proof Systems and Proof-Carrying Data Systems

Probabilistic proof systems are cryptographic primitives in which a prover provides correctness guarantees to a verifier on a computation performed by the prover [30, p. 6]. Fig. 3 illustrates the interaction between computation, prover and verifier. Proof systems expect adversarial, i.e., Byzantine,

provers [12]. Probabilistic proof systems based on polynomial commitment schemes need the computation in form of a boolean circuit or arithmetic circuit, and then encode the execution of the circuit into a polynomial [25]. While these circuits can express any finite-input deterministic computation, the computation is usually expressed in a high-level language and compiled into circuit form by the proof system implementation [30, Chapter 6].

Proof-carrying data (PCD) systems enable a set of processes to execute a distributed algorithm and verify that other processes correctly carried out every step of the computation [7, 12]: Data exchanged between processes carries a recursive, aggregated proof that the data's source computation and all preceding computations were correct [12, Section 1.2]. Correctness is represented as a predicate, i.e., a set of safety properties, that must hold at every process on every message [12, Section 1.2]. Recursive proof composition is a property of proof systems where proofs can attest the correctness of other instances of themselves [9, p. 1], which allows to split up a computation into incrementally-verifiable parts. Proof aggregation allows to verify multiple previous proofs in a new proof [30, p. 285].

In this paper, we use a computationally-sound, succinct, non-interactive PCD system developed as part of a blockchain project [26]. A proof system is called sound if even a Byzantine prover can find incorrect proofs only with negligible probability [24]. If soundness only holds for computationally bounded provers, the proofs are also called arguments [30, p. 24]. In non-interactive proof systems, the proof is a single message of the prover to the verifier [30, p. 7]. A proof system is succinct if the proof size grows slower than linearly with the size of the computation [30, p. 108]. A proof system is called zero-knowledge if a proof reveals nothing but its own validity [30, p. 6f]. While PCD can be used in ways that have the zero-knowledge property, it is not a necessity [7].



(a) A function that describes a step in the computation of a distributed algorithm, to be executed by the prover. The function has a public and a private input, as well as a public output.

(b) The prover executes the function from (a), and shares public input, public output, and proof of validity with the verifier, but not the private input. The verifier only returns true if the function from (a) returns the public output given public and private input.

Figure 3. Given functions that describe steps of a distributed algorithm as in Fig. 3a, a proof system implementation generates functions for the prover and the verifier to generate and validate a proof of the function, as in Fig. 3b.

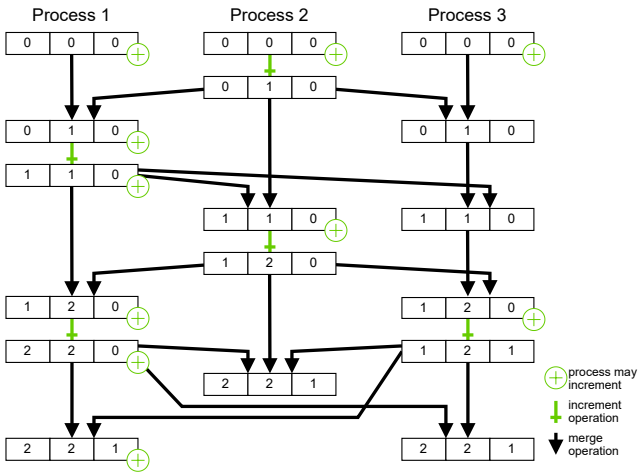


Figure 4. Exemplary state evolution of an increment-only counter with the additional restriction that a process may only increment the counter if the current value is divisible by the identifier of the process (divis-validity).

3 Case Study: Proof-Carrying Counters

We now study an artificial but illustrative counter CRDT to exemplify the principles of proof-carrying CRDTs, preparing for the more practical but complex CRDT in Sec. 4.

State-based CRDTs for increment-only counters provide a `value` query and an `increment` update operation, whereby updates are intrinsically commutative [27]. The state is an array of integers. Assuming that each process has an identifier $(1, 2, 3, \ldots)$, each process has a dedicated array element which the process may increment. The counter `value` is the sum of all array elements. Merging two states means taking their pairwise maximum. We assume that processes know the public keys of each other, and digitally sign updates. To restrict the set of valid updates, we add a contrived but simple-to-validate application invariant: a process may only increment the counter if the current value is divisible by its identifier. Specifically, we recursively define **divis-validity** as follows: A counter update is valid if:

1. the update is validly signed by its denoted creator
2. the `value` of the state from merging the update's history is divisible by the creator's identifier
3. the update is equal to merging the update's history and increasing the entry of the creator by one
4. all updates in the update's history are divis-valid

The behavior of a CRDT with divis-validity is shown in Fig. 4. A Byzantine-tolerant update validation algorithm for

```
1  state S: Integer [], p: Proof // state and validity proof
2  query value(): Integer
3      return ∑S
4  mutate increment()
5      if value() mod id() = 0
6          S' ← S with S'[id()] ← S[id()] + 1
7          p ← genProof('inc', S, p, S', sign(S')) // prove
               divis-valid increment
8          S ← S'
9  on receive(S': Integer [], p': Proof)
10     if vfyProof(S', p') // verify divis-validity
11         p ← genProof('mrg', S, p, S', p')
               // prove divis-valid merge
12         S ← pairwise max of S, S'
13 periodically
14     gossip( S, p)
```

Algorithm 1: A state-based proof-carrying counter CRDT that ensures divis-validity. Our implementation is provided in [23, src/crdt/counter.ts].

divis-validity without proof-carrying data is not straightforward: Processes must fully replicate the update history among all processes, and wait for visibility of an update's history for validation. As the standard algorithm for a state-based counter CRDT only replicates and stores a most recent update, divis-validity as a seemingly minor application invariant introduces a comparatively major overhead.

As shown in Alg. 1, by making all updates carry a proof of divis-validity, we can substantially simplify the update validation algorithm: `genProof('inc', S, p, S', sign(S'))` creates a proof that signed state S' is a valid increment based on state S and proof p; `genProof('mrg', S, p, S', p')` creates a proof that the pairwise maximum of local state S, proven valid by p, and received state S', proven valid by p', is a valid state. Before an update is applied, its proof is verified using `vfyProof()`, which ensures that the update's full history is recursively divis-valid. The proof-carrying counter CRDT thereby provides succinct non-interactive Byzantine update validation, as it only needs to replicate the set of current state updates instead of the full update history.

4 Case Study: Proof-Carrying Update Sets

We now study a more complex, practice-oriented example of a proof-carrying CRDT, based on the underlying CRDT of the Matrix decentralized group communication system. In Matrix, the history of a communication group is a Byzantine-tolerant delta-state CRDT for a grow-only, partially-ordered set of updates [18]. An update either describes a message in the communication group, like a chat message, or a change in the associated data of the group, like chat group topic or membership information. In Matrix, group data like whether the sender is a member of the group also affects the validity of updates. Like in Fig. 1, the partial order among updates describes their causal history, as declared by each update's creator process. The partial order includes a unique minimal update, called the genesis update, that describes the group's creation, initial state, and identifier of the group. Update history CRDTs provide an append method to add new updates causally after a subset of predecessor updates, as well as queries like the set of maximal updates, the update for a given identifier, or causal order of two given updates.

To ensure authenticity and integrity of the update history in face of Byzantine faults, the update history CRDT algorithm employed in Matrix rests on digital signatures and recursive history hashing [18, 19, 21]. Updates are identified by their hash value using a collision-resistant hash function. Every update contains a set of identifiers of the update's direct predecessors. This establishes the binding between hash identifier, update, and, recursively, its causal history.

For this study, we recursively define a simplified **update-validity** as follows. To avoid needing update history for partially ordering two updates, ordering by update depth measured as maximal path length to the genesis update is consistent with causality [18, 28]. An update is valid if:

1. the genesis update is part of the update's history
2. the update creator knows preimage updates of the update's linked predecessor hashes
3. update depth = depth of its deepest predecessor + 1.
4. all updates in the update's history are update-valid

Processes send and receive delta-states, i.e., sets of updates. To validate an update, a Byzantine-tolerant update validation algorithm without proof-carrying data needs to wait until the update's full history has been received and validated. Thereby, the algorithm cannot deal with "gaps" without trust in other processes. The newest updates have to be validated last, which is detrimental to performance especially if a process joins the group or has been offline for a long time.

In contrast to the fixed-size counter in Sec. 3, the state of the update history is grow-only, even with proofs of validity. Therefore, our proof-carrying CRDT does not replicate the full update history, but only the *maximal update set*, i.e., the subset of newest updates of the history, as shown in Alg. 2. This approach works because the subsets of maximal elements of causal histories are a join-semilattice as well [4]. Processes thereby eventually obtain a superset of the maximal updates present on at least one correct process.

In a similar way to full and light replicas of a blockchain [31, Section 1.3], the update history CRDT and the proof-carrying update set CRDT are interoperable: The maximal update set is a subset of the update history, and acts as common ground between all processes. Processes can freely decide how much history they store: It is possible for a process to eventually receive all updates known to at least one correct process via queries based on the partial order.

```
1  state g: Update // genesis update
2  state S: (Update, Proof){} ← ∅ // update history set
3  query updates(): Update {}
4      return { u | (u, p) ∈ S }
5  query maxUpdates(): Update {}
       // newest updates, i.e., without successors
6      return {u | (u, p) ∈ S ∧ ∄ (u',p') ∈ S: id(u) ∈ u'.pre}
7  function compare(a: Update, b: Update): Boolean
8      return a.depth < b.depth
9  mutate append(u: Update)
10     u.pre ← {id(u') | u' ∈ maxUpdates() }
           // direct predecessor hashes
11     u.depth ← max({u'.depth | u' ∈ maxUpdates() })+1
12     let p ← genProof(id(u), id(g), u, {(u', p') ∈ S | u' ∈
           maxUpdates() }) // prove update-validity
13     S ← S ∪ {(u, p)}
14     gossip( {(u, p)})
15 on receive(δ: (Update, Proof){})
16     for (u, p) ∈ δ
17         if vfyProof(id(u), id(g), u.depth, p)
             // verify update-validity
18           S ← S ∪ {(u, p)}
19 periodically
20     gossip( {(u, p)∈ S | u ∈ maxUpdates() })
```

Algorithm 2: A delta-state based proof-carrying CRDT for update sets.

Non-interactive proof systems inherently pose the limitation that algorithms need to be expressed as a fixed-size circuit with fixed-size inputs. The challenges of fixed-size input are the varying size of the update itself as well as the varying number of linked predecessors. We make use of (fixed-size) update *hashes*, and use the hash of the genesis update as group identifier. The identifier hash $\texttt{id}(U)$ of an update $U{=}(M, G, D, P)$ with message M, group identifier G, depth D, set of direct predecessors P, and number of direct predecessors $\#P$ is the hash of their concatenation, $\texttt{id}(U) := H(M\|G\|D\|\#P\|P)$. For the set of direct predecessors, we can either define a set size limit and make the circuit large enough to validate them all at once, or do recursion over the set as described below.

The PCD proof has to attest that the update-validity predicate $V(I, G, D)$ holds, i.e., that the prover knows a valid update with identifier I for the group with identifier G and update depth D. Update identifier I, group identifier G, and update depth D are considered public inputs (needed for verification), while message M and set of direct predecessors P are considered "private" inputs (not *needed* for verification). The update-validity predicate can be expressed as follows. Expression (1) holds if the prover knows a message M and direct predecessor set P for hash identifier I. If all direct predecessors have a smaller depth (Expr. (2)) and at least one direct predecessor has a depth of the update minus one (Expr. (3)), the update depth is the maximum of the predecessor depths plus one. Finally, Expr. (4) holds if the group identifier is the hash identifier of the genesis update.

$$
\begin{aligned}
V(I, G, D) := {} & \exists M, P\colon I = \texttt{id}([M, G, D, P]) && (1)\\
& \wedge (\forall p \in P\colon \exists d < D\colon V(p, G, d)) && (2)\\
& \wedge (\exists p \in P\colon V(p, G, D\text{-}1)) && (3)\\
& \wedge (\#P = 0 \Rightarrow D = 0 \wedge G = \texttt{id}([M, 0, 0, P])) && (4)
\end{aligned}
$$

As we have achieved fixed-size public inputs using hashes, the remaining challenge is to validate arbitrarily large direct predecessor sets as private input with a fixed-size circuit. Instead of calculating the hash identifier using all direct predecessors at once (Expr. (1)) and validating all direct predecessors at once (Expr. (2)), we recursively hash and validate one direct predecessor at a time. To evaluate the hash function on the arbitrarily long input, we utilize that the proof system optimized hash function Poseidon [13] is based on the 'recursion-friendly' Merkle-Damgård construction, which uses a compress function to incrementally build the hash, enclosed by an initialization and finalization function:

$$\texttt{H}(A\|B) = \texttt{final}(\texttt{cpress}(\texttt{cpress}(\texttt{init}(), A), B)),$$

where `cpress()` denotes the compression function. As we cannot validate all direct predecessors at once in a fixed-size circuit, we formulate a recursive validity predicate that needs only one predecessor per recursion step, with base case V_B and recursion case V_R. Let S denote the current state of the hash function, N the number of remaining predecessors, and F a boolean flag whether at least one predecessor so far had a depth of one less than the update's depth:

$$
\begin{aligned}
V_I(S, G, D, N, F) &:= V_B(S, G, D, N, F) \vee V_R(S, G, D, N, F)\\
V_B(S, G, D, N, F) &:= \exists M\colon S = \texttt{cpress}(\texttt{init}(), M\|G\|D\|N)\\
&\quad \wedge N \geq 0 \wedge (N = 0 \Rightarrow G = \texttt{H}(M) \wedge D = 0)\\
&\quad \wedge F = (N = 0)\\
V_R(S, G, D, N, F) &:= \exists S', p, d, F'\colon V_I(S', G, D, N{+}1, F')\\
&\quad \wedge F = (F' \vee D = d - 1) \wedge N \geq 0\\
&\quad \wedge S = \texttt{cpress}(S', p) \wedge d < D \wedge V(p, G, d)\\
V_f(I, G, D) &:= \exists S\colon I = \texttt{final}(S) \wedge V_I(S, G, D, 0, \text{True})
\end{aligned}
$$

We now have achieved a recursive validity predicate V_f that only depends on fixed-size public input and on one fixed-size part of the private input per recursion step, i.e., one direct predecessor per recursion step. Thereby, the validation predicate translates to a recursive but fixed-size circuit.

Please note that the presented proof-carrying update set is inspired by the underlying CRDT of Matrix, however, it does not yet include validation of the sophisticated access control system of Matrix [17]. We consider the inclusion of these further access control policies as out of scope of this paper, but important future work.

5 Evaluation

Asymptotic Performance. PCD proof size, generation, and verification times depend on circuit size as well as number of times the circuit is executed, i.e., the number of intermediate proofs for internal recursion inside a proof. Thereby, generation and verification of update validity proofs depends on the number of direct predecessors, $O(\#P)$. However, the cost of one recursion step is independent of recursion depth. Thereby, while a proof of validity for an update implies the validity of its update history, proof cost is independent of the update history size $\#U$. In contrast, validating an update without a proof or prior knowledge of the update history needs replication and validation of all prior updates, $O(\#U)$, starting from the genesis update. In use cases where the most recent updates are most relevant, like chat applications, proof-carrying CRDTs show an asymptotic advantage: they can explore history starting from the most recent updates.

Empirical Performance. Proof generation and verification dominate execution time. To evaluate our proof-carrying CRDTs, we implemented them using the o1js non-interactive proof system [26]. Our implementation is freely available under MIT license [23]. Performance was measured on an Intel® Core™ i9-12900 with 12 cores and 62 GiB of RAM. Empirically, 16 threads resulted in optimal execution speed, and less than 3 GiB of RAM were occupied. We implemented the two approaches to proof-carrying update sets presented in Sec. 4. In the first variant, there is an upper limit to the number of direct predecessors for an update. As o1js only supports the recursive validation of two previous proofs, we were limited to a maximum of two direct predecessors. In the second variant, updates may have arbitrary many direct predecessors via recursion of intermediate proofs. Table 1 shows a performance comparison between the two implementation variants. The fixed direct predecessor variant is simpler and faster, while the arbitrary direct predecessor variant needs about 12 s per direct predecessor due to the additional intermediate proof. Proof verification took ≈ 0.6 s in all cases, regardless of the number of direct predecessors.

Reality Check. For security and performance reasons, Matrix enforces an upper limit of 20 direct predecessors, and uses no more than 5 recent and 5 random predecessors from the maximal update set [1, 17]. With current proof systems, we see applications in rare operations not on the critical path, like garbage collection / snapshotting. Efficiency of proof systems has made incredible progress: Proof sizes were ≈30 KiB, which we deem manageable. Proof verification was surprisingly fast, but still took ≈900 MiB of RAM (including overhead of JavaScript runtime). However, proof generation is still too slow for Matrix servers to be used for every update.

Security Aspects. There are several possibilities for vulnerabilities in proof systems: in the circuit itself, the circuit compiler, the proof generator, the proof verifier, and in their integration [11]. The hash functions used in o1js are not properly padded, which allows for hash collisions for inputs of different lengths [6, V-O1J-VUL-036]. For hash inputs with variable length, we fixed this with a wrapper which provides reasonable padding similar to the original Poseidon proposal for variable input length hashing [13, Section 4.2], and provide caching of inputs which are smaller than the block size [23, src/utils/hash.ts]. Even with a carried proof, an update might prove as invalid, as the proof may not prove the right thing: It is crucial not to forget any constraints in the circuit [11, Section 5]. Strong typing and abstractions, as in o1js, help to avoid common errors [11, Section 8].

Table 1. Performance comparison between two proof-carrying update set CRDT variants (c.f. Sec. 2), average of 100 runs, on 12 CPU cores. Standard deviations ≤ 7 % of average.

PCD step:	# of supported direct predecessors:		
		$\{0, 1, 2\}$	$\mathbb{N}$
circuit compilation		3.55 s	2.81 s
proof generation	genesis update	5.41 s	5.26 s
	1 direct predecessor	8.82 s	17.09 s
	2 direct predecessors	11.89 s	28.90 s
	3 direct predecessors		40.69 s
	4 direct predecessors		52.53 s
proof verification	on 1 CPU core	2.22 s	2.23 s
	on 12 CPU cores	0.57 s	0.59 s

6 Related Work

Causal stability is an alternative to combat grow-only CRDT state size [5], which determines obsolete updates. An update is causally stable when a process knows for sure that it has received all concurrent updates. In crash-fault environments, this works by assuming that any process totally orders its updates, but in Byzantine environments, Byzantine processes can send concurrent updates at any time.

PCD proofs are used for validating blockchain blocks in constant time [8]. In a blockchain, blocks are arranged in an append-only total order. In CRDTs, however, concurrent updates lead to a partially-ordered update history.

Secure multi-party computation is similar to zero-knowledge proofs as it can preserve the privacy of inputs, but it is not possible for *others* to verify that a computation has been performed correctly [10], which is crucial for update validation.

7 Conclusion

We have shown that it is practically possible to prove the validity of CRDT updates in a succinct, non-interactive, and Byzantine fault tolerant way using proof-carrying data systems. We presented a proof of concept, freely available under MIT license [23]. The observed speed is good enough for high-level use cases off the critical path. Future work needs to address more elaborate validation functions as used in practice in Matrix, specifically including access control.

Acknowledgments

This work was funded by the Helmholtz Pilot Program Core Informatics. We like to thank Martin Kleppmann for his thoughts on the intricacies of proof system programming and his encouragement to deal with them.

References

[1] 2024. *Matrix Specification v1.13*. Technical Report. The Matrix.org Foundation CIC. https://spec.matrix.org/v1.13/

[2] Paulo Sérgio Almeida. 2024. Approaches to Conflict-free Replicated Data Types. *ACM Comput. Surv.* 57, 2, Article 51 (Nov. 2024), 36 pages. doi:10.1145/3695249

[3] Paulo Sérgio Almeida. 2024. A Framework for Consistency Models in Distributed Systems. doi:10.48550/arXiv.2411.16355 arXiv:2411.16355

[4] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. 2017. Composition in State-Based Replicated Data Types. *Bulletin of the European Association for Theoretical Computer Science* 123 (Oct. 2017), 21. https://run.unl.pt/handle/10362/93093

[5] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency (PaPEC '14)*. Association for Computing Machinery, New York, NY, USA, 1–2. doi:10.1145/2596631.2596632

[6] Benjamin Sepanski, Sorawee Porncharoenwase, Alp Bassa, and Daniel Dominguez. 2024. Audit Report: o1js. https://github.com/o1-labs/o1js/blob/a09c5167c4df64f879684e5af14c59cf7a6fce11/audits/VAR_o1js_240318_o1js_V3.pdf

[7] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. 2021. Halo Infinite: Proof-Carrying Data from Additive Polynomial Commitments. In *Advances in Cryptology – CRYPTO 2021*, Tal Malkin and Chris Peikert (Eds.). Vol. 12825. Springer International Publishing, Cham, 649–680. https://link.springer.com/10.1007/978-3-030-84242-0_23

[8] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. 2020. Mina: Decentralized Cryptocurrency at Scale. (March 2020). https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf

[9] Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. Recursive Proof Composition without a Trusted Setup. (2019). https://eprint.iacr.org/2019/1021

[10] Ran Canetti, Uri Feige, Oded Goldreich, and Moni Naor. 1996. Adaptively secure multi-party computation. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing (STOC '96)*. Association for Computing Machinery, New York, NY, USA, 639–648. doi:10.1145/237814.238015

[11] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. 2024. SoK: What don't we know? Understanding Security Vulnerabilities in SNARKs. https://arxiv.org/abs/2402.15293

[12] Alessandro Chiesa and Eran Tromer. 2010. Proof-Carrying Data and Hearsay Arguments from Signature Cards. (2010). https://conference.iiis.tsinghua.edu.cn/ICS2010/content/paper/Paper_25.pdf

[13] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2019. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. https://eprint.iacr.org/2019/458

[14] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (Jan. 1991), 124–149. doi:10.1145/114005.102808

[15] Matthew Hodgson. 2023. Matrix 2.0. https://archive.fosdem.org/2023/schedule/event/matrix20/

[16] Florian Jacob, Saskia Bayreuther, and Hannes Hartenstein. 2022. On CRDTs in Byzantine Environments. https://dl.gi.de/items/c04ab16e-a0c6-477b-8773-048966727979

[17] Florian Jacob, Carolin Beer, Norbert Henze, and Hannes Hartenstein. 2021. Analysis of the Matrix Event Graph Replicated Data Type. *IEEE Access* 9 (2021), 28317–28333. doi:10.1109/ACCESS.2021.3058576

[18] Florian Jacob and Hannes Hartenstein. 2024. Logical Clocks and Monotonicity for Byzantine-Tolerant Replicated Data Types. In *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Athens Greece, 37–43. doi:10.1145/3642976.3653034

[19] Brent Byunghoon Kang, R. Wilensky, and J. Kubiatowicz. 2003. The Hash History Approach for Reconciling Mutual Inconsistency. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* 670–677. doi:10.1109/ICDCS.2003.1203518

[20] Martin Kleppmann. 2015. A Critique of the CAP Theorem. https://arxiv.org/abs/1509.05393

[21] Martin Kleppmann. 2022. Making CRDTs Byzantine fault tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Rennes France, 8–15. doi:10.1145/3517209.3524042

[22] Marc Shapiro, Marek Zawirsk, Carlos Baquero, and Nuno Preguiça. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed*. Lecture Notes in Computer Science, Vol. 6976. Springer Berlin Heidelberg, Berlin, Heidelberg. https://link.springer.com/10.1007/978-3-642-24550-3

[23] Nick Marx. 2025. Proof-carrying CRDTs: Source Code. https://github.com/kit-dsn/proof-carrying-crdts/tree/papoc25

[24] Silvio Micali. 2000. Computationally Sound Proofs. *SIAM J. Comput.* 30, 4 (2000), 1253–1298. doi:10.1137/S0097539795284959

[25] Michael Walfish and Andrew J. Blumberg. 2015. Verifying computations without reexecuting them. *Commun. ACM* 58, 2 (Jan. 2015), 74–84. doi:10.1145/2641562

[26] O(1) Labs. 2024. o1-labs/o1js. https://github.com/o1-labs/o1js

[27] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. 2018. Conflict-free Replicated Data Types (CRDTs). doi:10.1007/978-3-319-63962-8_185-1

[28] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing* 7, 3 (March 1994), 149–174. doi:10.1007/BF02277859

[29] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. A comprehensive study of Convergent and Commutative Replicated Data Types. (Jan. 2011). https://inria.hal.science/inria-00555588/document

[30] Justin Thaler. 2023. Proofs, Arguments, and Zero-Knowledge. (July 2023). https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf

[31] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. 2018. *Blockchain technology overview*. Technical Report NIST IR 8202. National Institute of Standards and Technology, Gaithersburg, MD. NIST IR 8202 pages. https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf