Secure RDTs: Enforcing Access Control Policies for Offline Available JSON Data

THIERRY RENAUX, Vrije Universiteit Brussel, Belgium
SAM VAN DEN VONDER, Vrije Universiteit Brussel, Belgium
WOLFGANG DE MEUTER, Vrije Universiteit Brussel, Belgium

Replicated Data Types (RDTs) are a type of data structure that can be replicated over a network, where each replica can be kept (eventually) consistent with the other replicas. They are used in applications with intermittent network connectivity, since local (offline) edits can later be merged with the other replicas. Applications that want to use RDTs often have an inherent security component that restricts data access for certain clients. However, access control for RDTs is difficult to enforce for clients that are not running within a secure environment, e.g., web applications where the client-side software can be freely tampered with. In essence, an application cannot prevent a client from reading data which they are not supposed to read, and any malicious changes will also affect well-behaved clients. This paper proposes Secure RDTs (SRDTs), a data type that specifies role-based access control for offline-available JSON data. In brief, a trusted application server specifies a security policy based on roles with read and write privileges for certain fields of an SRDT. The server enforces read privileges by projecting the data and security policy to omit any non-readable fields for the user's given role, and it acts as an intermediary to enforce write privileges. The approach is presented as an operational semantics engineered in PLT Redex, which is validated by formal proofs and randomised testing in Redex to ensure that the formal specification is secure.

CCS Concepts: • **Software and its engineering** → **Semantics**; • **Security and privacy** → **Distributed systems security**; **Web application security**; • **Computer systems organization** → *Availability*; • **Computing methodologies** → *Distributed programming languages*.

Additional Key Words and Phrases: replicated data types, role-based access control, security, conflict-free replicated data types

ACM Reference Format:
Thierry Renaux, Sam Van den Vonder, and Wolfgang De Meuter. 2023. Secure RDTs: Enforcing Access Control Policies for Offline Available JSON Data. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 227 (October 2023), 27 pages. https://doi.org/10.1145/3622802

1 INTRODUCTION

Modern distributed applications often replicate data over the network. By storing a copy of the data locally, an application can guarantee low latency data access, and it becomes resilient to temporary network failures. Notable data abstractions called *Replicated Data Types* (RDTs) manage the complexity of data replication to ensure that each *replica* (i.e., local copy) can be brought back to a consistent state with the other replicas after a network failure. A popular variant of RDTs are *Conflict-free Replicated Data Types* (CRDTs) [Shapiro et al. 2011a,b], which are used in distributed databases and are offered by many libraries. Of particular importance for this paper is the recent

Authors' addresses: Thierry Renaux, Thierry.Renaux@vub.be, Vrije Universiteit Brussel, Brussels, Belgium; Sam Van den Vonder, Sam.Van.den.Vonder@vub.be, Vrije Universiteit Brussel, Brussels, Belgium; Wolfgang De Meuter, Wolfgang.De.Meuter@vub.be, Vrije Universiteit Brussel, Brussels, Belgium.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/10-ART227
https://doi.org/10.1145/3622802

work on a CRDT for the JSON data type,[1] which has practical implementations for JavaScript and Rust [Automerge Contributors 2023; Kleppmann and Beresford 2017, 2018].

Consider a modern web or mobile application built with popular Graphical User Interface (GUI) frameworks such as React or React Native [Madsen et al. 2020; Meta Platforms 2023a,b]. To reduce initial page load times and to encourage interactivity, users are initially served a page with limited static content and skeletons for the dynamic content. The skeletons are replaced dynamically, which in practice means that the client fetches JSON objects from an API and renders them locally via React components. Later on, the server will push more JSON objects to update the page dynamically (e.g., comments, chat messages, likes, shares, a data feed, . . .).

Instead of manually passing large JSON objects between a client and server, RDTs such as the aforementioned JSON CRDT are used as a programmer abstraction to more easily implement the application, and to make it usable offline "for free". In this case the client would receive a replica of the JSON data which, via the properties of RDTs, is automatically kept (eventually) consistent with other replicas of the same data whenever updates occur (e.g., by the server or other clients). This scenario is appealing for ease and correctness of the application's functionality, but raises security concerns because current approaches such as the JSON CRDT assume that all clients are trusted. For web applications this has the following security drawbacks:

(1) All clients receive a full copy of the data even if they are not allowed to read all parts of it. While those parts can be hidden behind the GUI, the clients *do* receive the data and can still access it (e.g., via a web browser's developer tools).
(2) It is a fundamental property of RDTs that any local changes to a replica will be merged with the replicas of other clients, such that when no other updates occur, all replicas will (eventually) converge to the same state [Shapiro et al. 2011a]. In other words, any changes made to any part of a local replica *will* be merged with the other clients. While the GUI can hide the ability to modify data, malicious clients can bypass it easily (e.g., via web browser extensions).

Hence, applications will have to enforce additional access control. For RDTs such as the JSON CRDT this currently requires ad-hoc coding which is: (1) very difficult to combine with an RDT's offline availability, (2) impossible to enforce on malicious clients, and (3) very likely to be insecure: "Broken Access Control" is the number 1 security problem identified in the 2021 OWASP[2] Top 10 [OWASP Foundation 2021].

The main contribution of this paper is the design of Secure RDTs (SRDTs), a data type which specifies Role-based Access Control (RBAC) [Sandhu et al. 1996] for offline-available JSON data. While the proposed approach applies to offline-available JSON data in general, it was designed with libraries such as Automerge [Automerge Contributors 2023; Kleppmann and Beresford 2018] and Yjs [Jahns and Yjs Contributors 2023; Nicolaescu et al. 2015] in mind, since they are popular, real-world implementations of offline-available JSON data. We assume a single *leader* per SRDT that specifies a security policy for said SRDT, and which acts as a central authority that is responsible for authentication and authorisation. The leader enforces read privileges by creating different projections of an SRDT depending on the security role of a client, and it enforces write privileges when clients modify a replica and try to push changes to the leader. As far as we know this is the first proposal to add RBAC to offline-available JSON data for the purpose of easier application development with code which is secure by design.

The paper is structured as follows. In Section 2 we more precisely state the problem with combining RBAC and offline available replicated data, and Section 3 explains our proposed approach. The formal semantics in PLT Redex is explained in Section 4. Sections 5 and 6 prove (formally and

[1] JavaScript Object Notation, a commonly used open standard for data interchange [ISO Central Secretary 2017].
[2] Open Web Application Security Project

through randomised testing respectively) that SRDTs are secure, and that they retain the properties of regular offline available replicas. Finally, Section 7 discusses the limitations and related work.

2 PROBLEM STATEMENT

We introduce the adversary model [Do et al. 2019], a running example, and state the problem.

2.1 Adversary Model

We assume an adversary model where some parts of an application run in a *trusted* environment, and other parts in an *untrusted* environment. The trusted environment consists of servers (e.g., owned by an organisation) which we assume to be secure in every way, i.e., trusted hardware and software, and not compromised by an attacker. Some of these servers will be responsible for user authentication and role assignment. The users that interact with the system using RDTs are potentially active adversaries that try to elevate their privileges, i.e., to read information from their replicas that they are not allowed to see, or to manipulate (write) information that they are not allowed to change. The threshold to become such an adversary is extremely low in web-based applications because every user can easily inspect and modify the client-side application via web browser extensions and a browser's standard code and network debugging tools.

2.2 Running Example

Citizen science is research conducted with participation from non-experts, often involving volunteers that collect data [Vohland et al. 2021]. In this context, consider a volunteer-driven organisation that is responsible for the upkeep of designated nature areas (e.g., Belgium's Natuurpunt [2023]). To maintain an overview of biodiversity, the organisation develops a web-based mobile application with which volunteers can report sightings of plant and animal species, and they invite the local youth organisations for a "spotting day" where the volunteers map a desired region (e.g., a forest). As is often the case for citizen science, the application spurs competition via gamification. In this case a biologist will award points to participants' sightings and provide live feedback.

JSON CRDT libraries such as Automerge [Automerge Contributors 2023; Kleppmann and Beresford 2018] and Yjs [Jahns and Yjs Contributors 2023; Nicolaescu et al. 2015] are an ideal implementation technology for the desired collaborative experience, and to ensure that the application remains usable offline (e.g., when users have bad cell phone reception in remote areas such as national parks). A proposal for the application's JSON data model is given in Figure 1a. The data contains 2 teams stored in the top-level fields `team1` and `team2`. As is typical for JSON, we will say that each field consists of a key (the field's name) and value. A team is represented by an object with a `name` and `sightings` field, where the latter is an object where each key is a Unix timestamp, and each value an object that contains the information of the associated sighting. A GUI is rendered based on this data, e.g., Figure 1b depicts a "card" GUI component that displays a sighting from the point of view of the biologist (e.g., as part of a live feed).

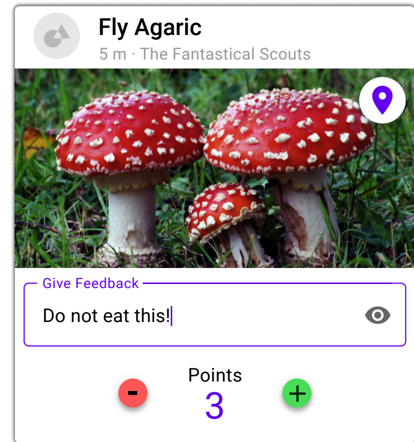
The application should enforce a straightforward security policy with the following rules:

(1) Users may add and modify sightings for their own team, which consist of a location, the species, and a photo.
(2) Users can see (but not modify) the points and feedback given on their sightings.
(3) Users can see the photo and points awarded to sightings of other teams.
(4) Biologists can see the sightings of all teams, can award points, and provide feedback.
(5) Anyone can see the name of a team.

Note that web-based applications are just one example of an application domain where offline available RDTs are used, and which have an inherent security component.

```
1  let data = {
2    team1: {
3      name: "The Fantastical Scouts",
4      sightings: {
5        1674813931967: {
6          location: { lat: 51.06038,
7                      lng: 4.67201 },
8          species: "Fly Agaric",
9          photo: "blob:...",
10         points: 3,
11         feedback: "Do not eat this!" }}},
12   team2: { ... },
13 }
```

(a) Data model of teams and sightings.



(b) Mockup of a sighting, biologist's view.

Fig. 1. Example data model and GUI mockup for a citizen science application.

2.3 Problems When Using RBAC for Offline Available Replicated JSON Data

By using an existing RDT such as Automerge or Yjs, one can already build an interactive application with the data model of Figure 1a. This makes the application easy to implement and usable when the user is temporarily offline. However, under the assumed adversary model, the application is vulnerable to the following problems.

Replicated Data Leaks Read privileges cannot be enforced because the entire data structure is replicated to every user. In the running example each volunteer would receive the data about their team and all other teams, including the locations of their sightings. While the GUI can implement the security policy by hiding information, it cannot *enforce* it since the data is stored on the local machine and can be extracted, e.g., via a web browser's JavaScript console. This clearly violates the principle of least privilege [Saltzer and Schroeder 1975], which is one of the most important principles to adhere to when securing an application. **The only way to prevent leaking sensitive data is to not send the data.**

Data Contagion Write privileges cannot be enforced locally, and any maliciously written data will be synchronised with well-behaved clients. E.g., volunteers are not allowed to modify the points and feedback of sightings. While the GUI can hide that functionality, it cannot be *enforced* since the user can bypass it (e.g., via the JavaScript console or web browser extensions), and due to the eventual consistency of RDTs all other replicas will eventually converge to the same, compromised application state. **Any change to a replica that does not conform to the security policy must not be merged with replicas of well-behaved clients.**

Lack of Offline Policy Enforcement One of the main benefits of RDTs is that an application remains usable without network connectivity, and when the network is restored the properties of RDTs guarantee that all replicas (eventually) converge to the same state. Without a network connection, a client cannot authorise read or write operations with the leader before executing them locally. Thus, if a replica's security hinges on its ability to reach the leader, then the RDT effectively becomes unusable without network connectivity, which is unacceptable. **Any security mechanism that is suitable for RDTs should preserve the offline availability of RDTs.**

3 APPROACH: SECURITY POLICY PROJECTION AND DATA PROJECTION

SRDTs overcome the problems listed in Section 2.3 by enforcing an application's security policy on both (well-behaved) clients and the leader. In brief:

Client-side Enforcement To support offline policy enforcement, each (well-behaved) client can check whether it is allowed to write to a field of their replicas. This prevents a well-behaved client from erroneously writing to a write-restricted field of a local replica (e.g., during a period of being offline). If write privileges are only checked when changes are (eventually) merged at the leader, then a well-behaved client could be forced to roll back its changes after the first disallowed write. This complicates the notion of eventual consistency in RDTs and invalidates the core purpose of offline available mutable state. Note that due to the adversary model it is impossible to *enforce* a security policy on the client-side. Hence, true enforcement must happen on the leader.

Leader-side Enforcement The leader prevents replicated data leaks by excluding the SRDT fields for which a client has no read privileges from the replica which is sent to said client. When a client authenticates with the leader to acquire an initial copy of the data, the *leader* will sanitise this data depending on the role of the client, as well as any future updates to the data. The leader also prevents data contagion by enforcing the security policy for all writes to fields, and *discarding* any writes from clients that are not authorised to do so. Hence, unauthorised changes will not be merged with the replicas from well-behaved clients.

To offer both types of enforcement, SRDTs require *projections* of the security policy and the SRDT data. Before showing a formal semantics in Section 4, we discuss our approach by specifying the concepts of security policies, security policy projections, and data projections in more detail.

3.1 Security Policy Specification Language

A leader securely replicates an SRDT based on a policy that consists of roles and *privileges*, which are machine-readable descriptions of operations that may be performed by a role. The technique to specify privileges is inspired by access control for XML documents [Fundulaki and Marx 2004], which has similar requirements. In essence, both JSON and XML are tree-structured documents where (parts of) the structure can be described as a path from the root of the document. We will call such a path a *path selector*. Then, a privilege grants read or write access to a set of SRDT fields (identified by a path selector) for a specified role, or a wildcard for all roles. In the S-expression syntax which we use throughout the remainder of this paper, a privilege is denoted as follows:

```
(ALLOW role READ/WRITE OF path-selector)
```

We first detail how to specify path selectors, followed by how to implement a full security policy.

3.1.1 *Specifying Object Paths With Path Selectors.*

Security rule (4) in the running example says that biologists can (among other things) see the sightings of all teams. This means that any biologist can access the values of the fields of the JSON object in Figure 1a named by the following paths, expressed as S-expression lists describing a path from the object's root:

```
(team1 sightings 1674813931967 location lat)
(team1 sightings 1674813931967 location lng)
(team1 sightings 1674813931967 species)
(team1 sightings 1674813931967 photo)
(team1 sightings 1674813931967 points)
(team1 sightings 1674813931967 feedback)
(team2 ... ) ...
```

Table 1. Overview of the supported types of path expressions adapted from JSONPath.

Path Expr.	Description
k	Key. Matches one field's name exactly (e.g., the `sightings` or `1674813931967` field).
␣ (space)	Child operator. Goes deeper into an object as indicated by the following path expression.
*	Wildcard. Matches all keys in the current object regardless of their name.
[∪ k_1 k_2 ...]	Union operator. Matches any of the given keys in the current object (e.g., k_1, k_2, etc).
[`f?` k] [`f?` [~ k ...]]	Expression operator. Matches all keys in the current object for which the predicate function `f?` holds, given each of the current object's keys as first argument, and the provided k as the second argument. An alternative form uses a nested "~" expression which looks up a value (e.g., given by the plain path (k ...)) in the user's environment. Supported predicates include = (object's key matches the value named by the lookup exactly) and ∈ (object's key is a member of the list named by the lookup).

Specifying privileges using only absolute paths is cumbersome, and in general even impossible given the dynamic nature of replicated JSON objects which can by modified continuously (e.g., when new sightings are made). For XML documents this problem is tackled by using the XPath query language [Clark and DeRose 1999] to specify which parts of an XML document can be accessed [Crampton 2006; Fundulaki and Marx 2004; Murata et al. 2006]. When using JSON's equivalent to XPath, called JSONPath [Friesen 2019; Goessner 2007], the paths above can be referred to via the 2 following path selectors that use a wildcard to abstract over multiple keys, thus capturing multiple teams and sightings.

```
(* sightings * *)
(* sightings * location *)
```

A wildcard is just one example of an expression that can capture multiple fields. All types of path expression that we adapted from JSONPath are described in Table 1. We will further explain them when they are used. In essence, we adapted JSONPath to S-expression syntax, and restricted the expressivity of the path expressions in 3 ways, namely:

(1) Path expressions cannot depend on a field's value (e.g., via JSONPath's "`@`" and "`$`" expressions). Instead, we introduce a restricted form (cf. Table 1) which depends only on the values of a per-user private object, which we call the *user environment*. The user environment can be thought of as an immutable dictionary (i.e., another JSON object) which is sent by the leader. Values from a user's environment can be read only by the user themselves and the leader using the syntax "[~ k ...]", chosen to invoke the intuition of a Unix user's "home directory".
(2) We do not support any expressions to manipulate arrays since SRDTs do not support array operations (e.g., `push`, `pop`, etc.). Instead, arrays can be represented as an object where the keys are numeric indices, and the fields' values serve as the array's values.
(3) JSONPath expressions such as "`..`" (a *recursive descent* operator) are an engineering effort for practical implementations, and are excluded for simplicity of the formalism.

The limitations (1) and (2) are further discussed in Section 7.5.

3.1.2 Specifying Security Policies. A security policy is a set of privileges. For example, the full security policy of the running example is implemented in Listing 1. Compared to the textual description of Section 2.2 they are ordered based on increasing complexity, starting with rule (5).

Rule (5) is implemented on Line 1. It grants all roles (denoted by the wildcard role) read access to the `name` field of all teams.

Rule (4) is implemented on Lines 2 to 4 using the path selectors previously shown in Section 3.1.1. The privileges grant the `biologist` role read access to the described paths. Note in particular that

```
1 (ALLOW *         READ OF  (* name))
2 (ALLOW biologist READ OF  (* sightings * *))
3 (ALLOW biologist READ OF  (* sightings * location *))
4 (ALLOW biologist WRITE OF (* sightings * [∪ points feedback]))
5 (ALLOW user      READ OF  (* sightings * [∪ photo points]))
6 (ALLOW user      READ OF  ([= [~ my-team]] sightings * feedback))
7 (ALLOW user      WRITE OF ([= [~ my-team]] sightings *))
8 (ALLOW user      WRITE OF ([= [~ my-team]] sightings * [∪ species photo]))
9 (ALLOW user      WRITE OF ([= [~ my-team]] sightings * location [∪ lat lng]))
```

Listing 1. Specification of the security policy of the running example (Section 2.2).

being granted access to a particular object does not mean that the role has access to all values within that object. In this case, Line 2 grants read access to all keys of sighting objects (e.g., the `1674813931967` field in Figure 1a). This includes the `location` key, but not `location`'s children `lat` and `lng`. A biologist can hence effectively *traverse* the `location` field, but still needs to be granted read privileges for its children on Line 3. The remainder of rule (4) is implemented on Line 4, which grants biologists write access to every sighting's `points` and `feedback` fields via a union path expression which selects both keys (cf. Table 1).

Rule (3) of the running example is implemented on Line 5. The privilege grants every volunteer (i.e., the `user` role) read access to the photo and points of all teams' sightings.

Rule (2) is implemented on Line 6, and gives users read access to the `feedback` field of only *their own team's* sightings. To encode this requirement of "a user's own team", the path selector uses an expression operator (cf. Table 1) which tests via a built-in predicate, in this case =, whether a key from the object should be included. In this case the keys in the object are teams, e.g., `team1` (cf. Figure 1a), and the expression with which it should match is specified using the ~ operator which looks up the current user's `my-team` field in the user's environment.[3]

Rule (1) is implemented on Lines 7 to 9 and gives users write access to their own team's sightings, excluding the `feedback` and `points` fields which only biologists can write to. Because users have wildcard write access to any key of the `sightings` object, it also grants them the permission to add new sightings (because they are covered by the wildcard). Note that a new sighting object that is added by a user can only include the fields to which users are explicitly granted write access, i.e., `species`, `photo`, and `location`. The `points` and `feedback` fields can be added later by biologists.

In general, Murata et al. [2006] state that access control policies should satisfy 3 requirements: *succinctness*, *least privilege*, and *soundness*. Succinctness means that policies should be expressible with a smaller number of privileges instead of having to specify every single field in the data. We satisfy this requirement by using path selectors modelled on JSONPath. Least privilege means that the security policy should grant the smallest possible access to a role, and soundness means that the security policy should always either allow or deny an access. Both are satisfied because all access to fields is denied unless access is explicitly granted via an `ALLOW` privilege. While we do not implement explicit `DENY` privileges for simplicity, they can be added using the "denial takes precedence" principle [Murata et al. 2006] where a `DENY` privilege removes any access that was previously granted. Note that in our model write permissions for a field imply that the role is allowed to read said field.

3.2 Projecting a Security Policy

The complete security policy is specified only on the leader of an SRDT. To support offline policy enforcement of SRDTs, clients must be able to check (offline) whether they are allowed to write to

[3] An example user environment is the object `((my-team := 'team1))`.

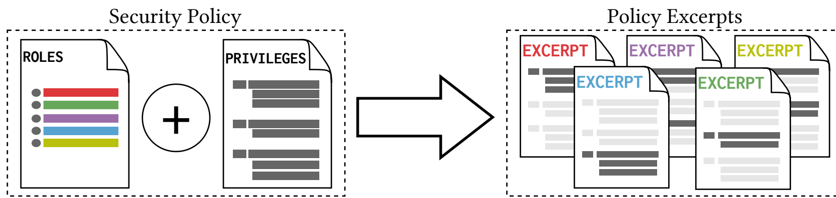


Fig. 2. The projection of a security policy based on each role yields a policy excerpt per role.

a field. Hence the leader compiles different projections of the security policy, namely one for each role. A high-level representation is depicted in Figure 2. The left depicts a security policy which is a set of roles and privileges, and the result depicted on the right is a set of *policy excerpts*. A policy excerpt contains the subset of privileges that (well-behaved) clients of a particular role are expected to adhere to locally. More specifically, the policy excerpt includes only a client's own privileges. Including other roles' privileges would enable a malicious client to know which fields exist that they do not have read access to, and what roles can access those fields. This poses a security risk in its own right.

3.3 Data Projection

To enforce that each client accesses only the data from the SRDT for which they have been granted read privileges, we ensure that each client is sent only the data it has read privileges for. We refer to the act of selecting the readable subset of an SRDT for a certain user as *data projection*. A correct data projection ensures that adversaries can never bypass privileges to extract data. However, a consequence of data projection is that each client has a potentially different subset of the complete SRDT. The leader manages cooperation between those ostensibly incompatible replicas.

Figure 3 depicts the interactions of clients and the leader, including the points where data projection occurs.

(1) First, a client authenticates with the leader (e.g., using a password, or public key authentication). The leader assigns a role to the client according to the security policy, which we depict as a red square (i.e., the "red square" role).
(2) Once authenticated, the client can ask the leader for a copy of the data to instantiate a local replica. The leader responds with the policy excerpt that corresponds to the client's role, a projected copy of the data that excludes all fields for which the client has no read privileges, and the user's environment needed to correctly interpret the policy excerpt.
(3) After initialisation, a client can perform operations on its local replica.

During periods without network connectivity a well-behaved client can check whether they are allowed to execute write operations by using the policy excerpt. Eventually, when network connectivity is restored, the local changes are sent to the leader, who verifies that the changes were permitted. Accepted changes are merged according to the semantics of the underlying RDT library. Other clients need to be (eventually) informed of this change in order for their replicas to (eventually) become consistent. In Figure 3 we depict 2 other clients with a "blue circle" and "green triangle" role. The leader cannot simply forward the change to both of them, because their roles may not have read privileges for the changed field. Instead the change is sent only to clients with the correct read or write privileges, which in Figure 3 includes the blue client but not the green client. Withholding the change for the green client prevents leaking sensitive data. It does not negatively impact the green client's normal operation since the changed field is not part of their local replica.

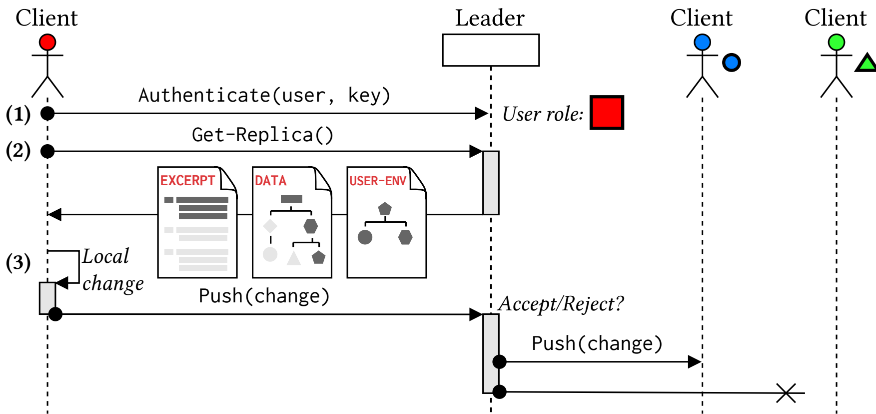


Fig. 3. Interactions between clients and a leader.

3.4 Overview of Assumptions

We briefly state the assumptions made (implicitly or explicitly) throughout Sections 2 and 3.

Adversary Model.

- All clients are untrusted. A client is supposed to check their security policy excerpt locally for correct operation, but the security model assumes that clients can disable any security feature included in their local software (e.g., in their web browser).
- The leader is completely trusted (trusted hardware and software).
- No defence against clients that leak data which they legitimately have access to.

Security Policy.

- The security policy is static (does not change at run-time).
- The security policy can only depend on a replica's field keys, not on their values (limitation further discussed in Section 7.5.4).
- Write access to a certain field also implies read access.

Data Model.

- A replica's data contains no arrays (limitation further discussed in Section 7.5.2).
- A replica's data contains no cycles (which are also not a part of standard JSON).
- No compound changes (limitation further discussed in Section 7.5.7).
- The underlying RDT is operations-based (further discussed in Section 7.5.8).

Distribution Model.

- There is a single leader per SRDT. Practical implementations may internally divide the work of supporting many clients, e.g., by using ordinary RDTs that synchronise peer-to-peer with each other, running on multiple trusted servers behind a load balancer.
- The leader is trusted by all clients.
- No direct (peer-to-peer) communication or synchronisation between clients (limitation further discussed in Section 7.5.1).

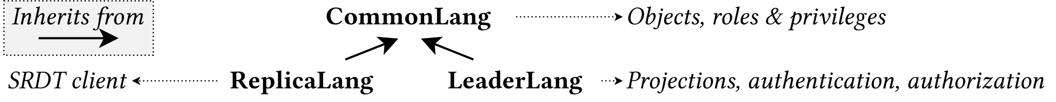


Fig. 4. Relation between the different formal languages.

```
      atom ::= number | boolean | string | quoted | ()        primitive values
    quoted ::= 'i                                             quoted symbols
         d ::= (kj ...)                                       json object
        kj ::= (k := json)                                    json field
         k ::= number | i                                     json field key
      json ::= atom | d                                       json field value
         p ::= (k ...)                                        field access path
         δ ::= (! p atom)                                     RDT delta (written value)
        ps ::= (p-exp ...)                                    path selector
     p-exp ::= k | ∗ | [script-op (~ k ...)] | [script-op k] | [∪ k ...]   path expressions
    p-role ::= role | ∗                                       security policy privilege role
      priv ::= (ALLOW p-role r/w OF ps)                       security policy privilege
       r/w ::= READ | WRITE                                   security policy permission
       env ::= d                                              user environment
 script-op ::= ∈ | ∋ | < | > | ≤ | ≥ | ≠ | =                  path expr. script operator
   role, i ::= variable-not-otherwise-mentioned               role name, identifier
```

Fig. 5. Semantic entities of CommonLang.

4 FORMAL SPECIFICATION OF SRDTS

We present a formal specification and implementation of SRDTs in PLT Redex [Felleisen et al. 2009; Klein et al. 2012], a domain-specific language in Racket [Felleisen et al. 2018] to specify operational semantics which are also executable. The formalism comprises 3 different languages as depicted in Figure 4: **ReplicaLang** and **LeaderLang** specify the behaviour of the client and leader respectively, and their commonalities are shared via a language called **CommonLang**. The complete implementation spans 2161 lines of Racket and Redex code (565 for the core formalism), and is available as a publication artifact.[4] All figures in this paper were generated by Redex to avoid the introduction of mistakes, sometimes with slight modifications in Adobe Illustrator to add explanatory notes or to reposition elements to fit within the allowable space. We use the same font convention in the text as in the figures. Non-terminals in the language grammar are typeset in italic in a serif font. Terminals are typeset in an upright monospaced non-serif font.

4.1 CommonLang: Objects, Roles and Privileges

The formalisation of the leader and the replicas share a common formal language called CommonLang. CommonLang specifies primitive atoms (numbers, booleans, strings, quoted symbols, and the empty object), and the definition of objects, roles and privileges. Its semantic entities are given in Figure 5, and will be introduced by example throughout this section.

4.1.1 Specifying Objects. The semantic entity *d* in Figure 5 defines the structure of JSON objects used to describe SRDT data. Every object is a list of terms *kj* of the form (*k* := *json*), where *k* is the name of a field and *json* its value. A list of terms is denoted in Redex via an ellipsis, which is a form of Kleene star which repeats the preceding term 0 or more times, e.g., (*kj* ...) denotes a list of 0 or more *kj* terms. For example, Listing 2 represents the JSON data of the running example in

[4]See data-availability statement at the end of the paper.

```
1 ((team1 := ((name := "The Fantastical Scouts")
2             (sightings := ((1674813931967 :=
3                              ((location := ((lat := 51.06038) (lng := 4.67201)))
4                               (species := "Fly Agaric")
5                               (photo := "blob:...")
6                               (points := 3)
7                               (feedback := "Do not eat this!")))))))
8   (team2 := <omitted for brevity>))
```

Listing 2. The data schema of Figure 1a in CommonLang syntax.

$$\texttt{excerpt-for-role}[\![role, ((\texttt{ALLOW} * r/w\ \texttt{OF}\ ps)\ priv_1 \ldots)]\!] = ((\texttt{ALLOW}\ role\ r/w\ \texttt{OF}\ ps)\ priv_2 \ldots)$$
where $(priv_2 \ldots) = \texttt{excerpt-for-role}[\![role, (priv_1 \ldots)]\!]$

$$\texttt{excerpt-for-role}[\![role, ((\texttt{ALLOW}\ role\ r/w\ \texttt{OF}\ ps)\ priv_1 \ldots)]\!] = ((\texttt{ALLOW}\ role\ r/w\ \texttt{OF}\ ps)\ priv_2 \ldots)$$
where $(priv_2 \ldots) = \texttt{excerpt-for-role}[\![role, (priv_1 \ldots)]\!]$

$$\texttt{excerpt-for-role}[\![role_1, ((\texttt{ALLOW}\ role_2\ r/w\ \texttt{OF}\ ps)\ priv \ldots)]\!] = \texttt{excerpt-for-role}[\![role_1, (priv \ldots)]\!]$$
where $role_1 \neq role_2$

$$\texttt{excerpt-for-role}[\![role, ()]\!] = ()$$

Fig. 6. Computing a policy excerpt for a particular role in LeaderLang.

Figure 1a, but conforming to the definition of a term *d*. The value of `team1` spans Lines 1 to 7, and `team2`'s on Line 8 has been omitted for brevity.

4.1.2 Specifying Security Policies. Each privilege from a security policy is represented by term *priv* in Figure 5. Note that the role can be a concrete identifier (e.g, `biologist`) or a wildcard (matching any role). The set of fields to which access is provided is given by a term *ps* that describes a path selector. Each path selector is a sequence of path expressions *p-exp* which we previously described in Table 1. To correctly assess whether a role is allowed to access a field, each path selector will expand to a set of concrete paths. Here, each concrete path conforms to semantic entity *p*, which is a sequence of keys that correspond to the fields of a (nested) object.

4.1.3 SRDT Deltas. Changes made by clients to the SRDT data structure are represented by a semantic entity δ that describes a single change (a written value) to a field. This can be considered as an operation from the underlying (operation-based) RDT (e.g., a JSON CRDT) that is sent from one replica to another to synchronise and (eventually) converge to the same state. Since the implementation of a full RDT library is outside the scope of this formalism, we represent such a change as a term that indicates that a value (an *atom*) was written to a path *p* that identifies the field it was written to.

4.2 LeaderLang: Securely Projecting Policies and Data

LeaderLang models the behaviour of the leader which is responsible for the authentication of users, projecting the security policy and SRDT data, and authorising changes to replicas made by clients. It is defined in 2 phases that correspond to the aforementioned projection phase and replica management phase (cf. Section 3). In the remainder of this section we discuss both phases in turn.

4.2.1 Projection Phase. We use Redex *metafunctions* to define computations on Redex terms. Essentially, a metafunction has a name and transforms the given input terms (its arguments) to an output term. Metafunctions pattern match on the structure of terms, optionally with side-conditions that must hold for a clause to match. Consider the metafunction called `excerpt-for-role` in

$$
\begin{array}{rcll}
\textit{request} & ::= & (\texttt{LOGIN}\ \iota^u\ \textit{auth-key}) & \text{client requests to log in} \\
& | & (\texttt{GET-REPLICA}\ \iota^s) & \text{client requests a replica} \\
& | & (\texttt{PUSH-}\Delta\ \iota^s\ \delta) & \text{client requests to push a change} \\
\textit{result} & ::= & (\texttt{ACCEPT}\ (\textit{action}\ .\ )) & \text{leader accepts a request, will perform actions} \\
& | & (\texttt{REJECT}) & \text{leader rejects a request} \\
\textit{action} & ::= & (\texttt{ACCEPT-LOGIN}\ \iota^s) & \text{leader accepts a new user session} \\
& | & (\texttt{INIT}\ \iota^s\ (\textit{priv}\ .\ )\ d\ \textit{env}) & \text{leader initialises a replica at a client} \\
& | & (\texttt{PUSH-}\Delta\ \iota^s\ \delta) & \text{leader pushes a change to a client} \\
\textit{excerpt} & ::= & (\textit{role}\ (\textit{priv}\ .\ )) & \text{security policy excerpt} \\
\textit{user} & ::= & (\iota^u\ \textit{role}\ \textit{auth-key}\ \textit{env}) & \text{user configuration} \\
\iota,\ \textit{auth-key} & ::= & \textit{string} & \text{id } (\iota^u \text{ user id, } \iota^s \text{ session id), user authentication key} \\
s & ::= & (\iota^s\ \iota^u) & \text{user session}
\end{array}
$$

Fig. 7. Semantic entities of LeaderLang.

Figure 6 which implements the main logic of the projection phase. For each equation, the left-hand side denotes the application of the metafunction on the input terms specified between brackets "⟦⟧" (separated via commas). The right-hand side of the equation is the output term. When side-conditions apply for an input term to match, those are specified as *where* clauses.

The `excerpt-for-role` metafunction computes the policy excerpt for one role. It takes 2 arguments, namely a role and security policy (given as a list of privileges). The result of the metafunction is a list of privileges that apply only to the given role. Essentially, `excerpt-for-role` recurses over the security policy from left to right and keeps only the privileges that apply to the given role. From top to bottom, the 4 clauses of the metafunction are as follows:

(1) When the privilege in the first position has the wildcard role, then the output list contains the same privilege but with the wildcard role replaced by *role*. The term $priv_2$ in the output list is the result of applying `excerpt-for-role` to the tail of the input list.
(2) When the privilege in the first position has the same role as the *role* argument, then the privilege is copied to the output list.
(3) When the privilege in the first position matches a *different* role than the one given as argument, then do not include it into the output.
(4) When the security policy is empty, then the policy excerpt is empty too.

4.2.2 Replica Management Phase. LeaderLang models the behaviour of a leader by reducing terms that represent a request from a client to a term that represents the actions that a leader undertakes to correctly process said request. The semantic entities defined by LeaderLang are given in Figure 7. Supported requests are `LOGIN` to authenticate a client, `GET-REPLICA` to obtain an SRDT replica, and `PUSH-Δ` to send a local change to the leader. A response is computed by the `handle-request` metafunction in Figure 8. The arguments of `handle-request` represent the run-time state of the leader, which is a list of known users (*user* ...), a list of policy excerpts (*excerpt* ...), the leader's copy of the SRDT data *d*, a list of user sessions (*s* ...) (one per client), and finally a *request* term to reduce. The result is the updated application state: new SRDT data, a new list of sessions, and a *result* term that indicates whether the leader accepts or rejects the request.

Handling `LOGIN`. Clients authenticate via a `LOGIN` request that contains a correct username and password, as handled by the first case of `handle-request` (Figure 8). The term matches only when all of the *where* clauses hold: the user must not have an active session, and the *auth-key* must be valid. In the returned application state a fresh session ($\iota^s\ \iota^u$) is added to the preexisting sessions, and the `LOGIN` term is reduced to an `ACCEPT-LOGIN` term that contains the fresh session id. A real-world system will use existing authentication protocols, but we do not model them as they are well-established and orthogonal to our approach.

```
handle-request⟦(user ...), (excerpt ...), d, (s_old ...), (LOGIN ι^u auth-key)⟧ = (d ((ι^s ι^u) s_old ...) (ACCEPT ((ACCEPT-LOGIN ι^s))))
 where no-active-session-for⟦ι^u, (s_old ...)⟧,
       key-is-valid⟦(user ...), ι^u, auth-key⟧,
       ι^s = fresh-session-id⟦ι^u⟧

handle-request⟦(user ...), (excerpt ...), d, (s ...), (GET-REPLICA ι^s)⟧ = (d (s ...) (ACCEPT ((INIT ι^s (priv ...) d_projected env))))
 where (ι^s ι^u) ∈ (s ...),
       (ι^u role _ env) ∈ (user ...),
       (role (priv ...)) ∈ (excerpt ...),
       d_projected = readable-projection⟦d, (priv ...), d, env, ()⟧

handle-request⟦(user ...), (excerpt ...), d, (s ...), (PUSH-Δ ι^s (! p atom))⟧ = (d_new (s ...) (ACCEPT (action ...)))
 where (ι^s ι^u) ∈ (s ...),
       (role (priv ...)) ∈ (excerpt ...),
       (ι^u role _ env) ∈ (user ...),
       is-writable⟦d, p, (priv ...), env⟧,
       d_new = json-write⟦d, p, atom⟧,
       (s_other ...) = (s ...)\(ι^s ι^u),
       (action ...) = actions-per-session⟦(s_other ...), (user ...), (excerpt ...), d_new, (! p atom)⟧

handle-request⟦(user ...), (excerpt ...), d, (s ...), request⟧ = (d (s ...) (REJECT))
```

Fig. 8. Handling client LOGIN, GET-REPLICA and PUSH-Δ requests in LeaderLang.

Handling GET-REPLICA. A GET-REPLICA term is reduced by the second case of handle-request. The returned term contains the data that is used to initialise a client. In this case the specified *where* clauses that must hold are "lookups", typeset via the ∈ notation, to retrieve information from a list of terms. The first clause specifies that the session id ι^s in the request must be an active session ($\iota^s\ \iota^u$), and the second clause uses the matching user id ι^u to retrieve the user's role and user environment. The third clause retrieves the policy excerpt for that role, and the fourth clause calls another metafunction *readable-projection* that projects the complete SRDT data d to a new object $d_{projected}$ which only contains the fields that can be read by the user according to their privileges. The implementation of *readable-projection* will be shown in Lemma 5.4. It is essentially a tree recursive copy which omits the non-readable fields. The information returned to the client is an INIT term with the session id ι^s, the policy excerpt (*priv* ...), the projected data $d_{projected}$, and the user's environment *env*.

Handling PUSH-Δ. The third case of handle-request handles a change to the SRDT pushed by a client, represented by a PUSH-Δ term. The input PUSH-Δ term denotes that a client identified by session ι^s has written an *atom* to a particular field access path p. The computed response is a list of PUSH-Δ terms that indicate to which *other* clients the accepted change should be forwarded in order to reach eventual consistency. When all of the *where* clauses hold, the result of the metafunction is an updated copy of the data d_{new}, the unmodified sessions, and an ACCEPT response that contains a list of *action* terms. The first 3 clauses are the same as for GET-REPLICA to extract the user's role, policy excerpt and user environment. Additionally, the leader uses the is-writable metafunction to verify that the given user is allowed to write to the path p according to their privileges. When this condition holds, the data is actually written to the object on the leader (yielding a new object d_{new}), and a list of actions is computed via the metafunction actions-per-session. Essentially, this metafunction returns a list of PUSH-Δ terms for all other sessions[5], but only when those clients have read privileges for the path that was written to.

Any other request which does not match the first, second or third case of handle-request is rejected by the fourth case.

[5]The notation $(s \ldots) \setminus (\iota^s\ \iota^u)$ is shorthand for a list of sessions $(s \ldots)$ *without* the specified session $(\iota^s\ \iota^u)$.

program ::=	((*r* ...) *e*)	client program
v ::=	(λ (*x* ...) *e*) | *atom* | *c*	values
e ::=	*x*	variable expression
	| *v*	value expression
	| (*e e* ...)	application expression
	| (*op e* ...)	application expression (Racket)
	| (`if` *e e e*)	if expression
	| (`let` ([*x e*] ...) *e*)	let expression
	| (`root` *e*)	root expression (replica root cursor)
	| (• *e k*)	cursor read field expression
	| (•! *e k e*)	cursor write field expression
	| (`error` *string*)	error expression
op ::=	`+` | `-` | `/` | `*` | `and` | `or` | `not` | `<` | `>` | `=`	built-in Racket operators
c ::=	(ι^r *p*)	cursor into replica
r ::=	(ι^r (*priv* ...) *d env* (δ ...))	replica object
ι ::=	*string*	replica name
x ::=	*variable-not-otherwise-mentioned*	variable

Fig. 9. Terms of ReplicaLang. Terms from CommonLang such as *atom*, *priv*, *d* and *env* are inherited.

4.3 ReplicaLang: Secure Manipulation of Replicas

ReplicaLang is a formalism of (well-behaved) clients which intends to strike a balance between being simple enough, yet being representative for client-side manipulation of replicas. Because ReplicaLang and LeaderLang model the behaviour of the system instead of serving as an implementation, there is no direct communication between ReplicaLang and LeaderLang in the formalism, i.e., ReplicaLang does not "send" `LOGIN` requests to LeaderLang, and LeaderLang does not "send" a response back to ReplicaLang. Instead, ReplicaLang models a client that has authenticated with a leader, has acquired a local replica, and which is manipulating replicas through program expressions. The paper's code artifact contains an interactive tool that, given data, a security policy, and users, integrates the requests and responses of ReplicaLang and LeaderLang to model real interactions.

Figure 9 shows the semantic entities of ReplicaLang. A *program* term contains a list of replica objects ($r \ldots$) and an expression *e*. Each replica object has an identifier ι^r, a list of privileges ($priv \ldots$), the replicated data *d*, a user environment *env*, and a list of changes ($\delta \ldots$) which are performed throughout the evaluation of *e*, but which are not immediately sent to the leader (i.e., they are "offline"). The expressions *e* are modelled after a variant of the λ-Calculus with support for multi-argument lambdas (for convenience), `let`-bindings, some primitive Racket operators *op*, and operators for interacting with replicas.

4.3.1 Interacting With Replicas. The expressions that interact with replicas are "•" to read a field of a replica object, and "•!" to write to a field. A reference into a replica object is represented as a "cursor", which is used to establish the full field access path whenever a field is written to. A real-world system might implement a cursor as a proxy that wraps the replicated object. A cursor (term *c*) stores the identifier of replica object it refers to and a path from the object's root. They are needed both for the normal workings of the underlying replica (e.g., cursors are also used by Kleppmann and Beresford [2017]), as well as specifically for our security policy enforcement.

Consider the example program in Listing 3 which is a valid *program* term of ReplicaLang that models an interaction with a replica. The first part of the *program* on Lines 1 to 10 contains the replica objects known to the program, in this case a replica called `"teams"`, which contains the policy excerpt for this user and the replica data. The expression on Lines 11 to 13 represents the program code that performs the interaction, which in this case adds feedback to a sighting. First, it navigates to the correct part of the replica object by obtaining a cursor `cr` to the root of the replica object (Line 11). Second, it navigates down into the object via the "•" operator to obtain a cursor

```
1  ((("teams" ((ALLOW biologist WRITE OF (* sightings * [∪ points feedback]))
2     ((team1 := ((name := "The Fantastical Scouts")
3                 (sightings :=
4                   ((1674813931967 :=
5                      ((location := ((lat := 51.06038) (lng := 4.67201)))
6                       (species := "Fly Agaric")
7                       (photo := "blob:...")
8                       (points := 3))))))))
9     ()
10    ()))
11  (let ((cr (root teams)))
12    (let ((sighting (• (• (• cr team1) sightings) 1674813931967)))
13      (•! sighting feedback "Do not eat this!"))))
```

Listing 3. Example program accepted by ReplicaLang.

```
1  ((("teams" ((ALLOW biologist WRITE OF (* sightings * [∪ points feedback]))
2     ((team1 := ((name := "The Fantastical Scouts")
3                 (sightings :=
4                   ((1674813931967 :=
5                      ((location := ((lat := 51.06038) (lng := 4.67201)))
6                       (species := "Fly Agaric")
7                       (photo := "blob:...")
8                       (points := 3)
9                       (feedback := "Do not eat this!"))))))))
10    ()
11    ((! (team1 sightings 1674813931967 feedback) "Do not eat this!"))))
12    "Do not eat this!")
```

Listing 4. ReplicaLang's reduction of Listing 3. The changes compared to Listing 3 are highlighted.

named `sighting` to the correct sightings object. Finally, the cursor is used to update (in this case, add) the `feedback` field via a "•!" expression.

The result of fully reducing the program is given in Listing 4, where we highlighted the parts of the term that have changed compared to Listing 3. First, the replica object's data was updated (Line 9) and the program's expression was reduced to `"Do not eat this!"` (Line 12), which is the return value of "•!" (the value that was written). Each replica object also stores a list of changes that occurred "while offline" and which need to be sent to the leader, in this case a single write to a field (Line 11).

4.3.2 Evaluation Semantics of ReplicaLang. Some of the reduction rules of ReplicaLang are standard for λ-Calculi, e.g., those for `if`, `let` and `apply` terms. Since they are not involved in upholding security we will not discuss them here. Beside the typical λ-Calculus rules, Figure 10 depicts the reduction rules for `root` terms (to obtain a cursor), "•" (read) terms and "•!" (write) terms. The format of each reduction rule $t_1 \longrightarrow t_2$ transforms a term that pattern matches t_1 into a reduced term t_2, optionally with side conditions specified via *where* clauses. In this case both t_1 and t_2 are *program* terms. For example, the `[root-cursor]` rule reduces a (`root` ι^r) term to a new cursor (ι^r ()) with an empty field access path. Note that the expression may be nested in another expression (e.g., a `let` or "•" term). These potential surrounding expressions are represented by the context E, which is a standard technique to abstract over compound expression as having a "hole" [Klein et al. 2012].

Consider the `[read]` rule. When a field k_2 is read from a cursor (ι^r (k_1 ...)), then evaluation will proceed by locating the data d of the replica object ι^r and using `json-read` on the full field access path ($k_1 \ldots k_2$) to retrieve the correct value from the replica. The metafunction `json-read` returns

$$((r \ldots)\ E[(\texttt{if}\ \#\texttt{f}\ e_1\ e_2)]) \longrightarrow ((r \ldots)\ E[e_2]) \qquad [\texttt{if \#f}]$$

$$((r \ldots)\ E[(\texttt{if}\ v\ e_1\ e_2)]) \longrightarrow ((r \ldots)\ E[e_1]) \text{ where } v \neq \#\texttt{f} \qquad [\texttt{if \#t}]$$

$$((r \ldots)\ E[(\texttt{let}\ ([x\ v] \ldots)\ e)]) \longrightarrow ((r \ldots)\ E[\texttt{substitute}[\![e, [x\ v], \ldots]\!]]) \qquad [\texttt{let}]$$

$$((r \ldots)\ E[((\lambda\ (x \ldots)\ e)\ v \ldots)]) \longrightarrow ((r \ldots)\ E[\texttt{substitute}[\![e, [x\ v], \ldots]\!]]) \qquad [\texttt{apply}]$$

$$((r \ldots)\ E[(op\ v \ldots)]) \longrightarrow ((r \ldots)\ E[\texttt{apply-racket-op}[\![op, v, \ldots]\!]]) \qquad [\texttt{apply-rkt}]$$

$$[(r \ldots)\ E[(\texttt{root}\ \iota^r)]] \longrightarrow [(r \ldots)\ E[(\iota^r\ ())]] \qquad [\texttt{root-cursor}]$$

$$[(r \ldots)\ E[(\bullet\ (\iota^r\ (k_1 \ldots))\ k_2)]] \longrightarrow [(r \ldots)\ E[v]] \text{ where } (\iota^r\ _\ d\ _\ _) \in (r \ldots),\ v = \texttt{json-read}[\![\iota^r, d, (k_1 \ldots k_2), (k_1 \ldots k_2)]\!] \qquad [\texttt{read}]$$

$$[(r \ldots)\ E[(\bullet\ (\iota^r\ (k_1 \ldots))\ k_2)]] \longrightarrow [(r \ldots)\ (\texttt{error}\ string)] \text{ where } (\iota^r\ _\ d\ _\ _) \in (r \ldots),\ (\texttt{error}\ string) = \texttt{json-read}[\![\iota^r, d, (k_1 \ldots k_2), (k_1 \ldots k_2)]\!] \qquad [\neg\texttt{read}]$$

$$[(r \ldots)\ E[(\bullet!\ (\iota^r\ (k_1 \ldots))\ k_2\ atom)]] \longrightarrow [((\iota^r\ (priv \ldots)\ d_{new}\ env\ (\delta \ldots\ (!\ (k_1 \ldots k_2)\ atom)))\ r_{other} \ldots)\ E[atom]] \qquad [\texttt{write}]$$
$$\text{where } r_c \in (r \ldots),\ (\iota^r\ (priv \ldots)\ d\ env\ (\delta \ldots)) = r_c,\ \texttt{is-writable}[\![d, (k_1 \ldots k_2), (priv \ldots), env]\!],$$
$$(r_{other} \ldots) = (r \ldots) \backslash r_c,\ d_{new} = \texttt{json-write}[\![d, (k_1 \ldots k_2), atom]\!]$$

$$[(r \ldots)\ E[(\bullet!\ (\iota^r\ (k_1 \ldots))\ k_2\ atom)]] \longrightarrow [(r \ldots)\ (\texttt{error}\ \text{“Write forbidden”})] \qquad [\neg\texttt{write-}\neg\texttt{w}]$$
$$\text{where } (\iota^r\ (priv \ldots)\ d\ env\ _) \in (r \ldots),\ \neg\texttt{is-writable}[\![d, (k_1 \ldots k_2), (priv \ldots), env]\!]$$

$$[(r \ldots)\ E[(\bullet!\ (\iota^r\ (k_1 \ldots))\ k_2\ v)]] \longrightarrow [(r \ldots)\ (\texttt{error}\ \text{“Write forbidden”})] \text{ where } \neg\texttt{is-atom}[\![v]\!] \qquad [\neg\texttt{write-}\neg\texttt{a}]$$

Fig. 10. The complete set of reduction rules for expressions in ReplicaLang.

an *atom* or an extended cursor if the read field is another object. The read expression is reduced to this value v. Note that security checks are not needed, since non-readable fields were removed by the projection on the leader. Whenever a read is not permitted (i.e., the field does not exist) then the rule `[¬read]` discards the entire context E and returns `error`.

Complementary to `[read]` is the `[write]` rule to write an *atom* to a replica. This rule uses the metafunction `json-write` to modify the replica's local data. The crucial difference is that `[write]` only proceeds when `is-writable` holds, i.e., that the client is authorised to write to the given field according to the privileges $(priv \ldots)$ of the policy excerpt. Otherwise, `[¬write-¬w]` rejects the write with an error. Finally, `[¬write-¬a]` rejects any write of a non-atom value.

5 FORMAL VALIDATION OF THE SPECIFICATION

In this section we prove the claim that the formal specification in Section 4 tackles the 3 problems outlined in Section 2.3, namely freedom from Replicated Data Leaks (as Theorem 5.6), freedom from Data Contagion (as Theorem 5.7), and Offline Policy Enforcement (as Theorem 5.8). Before we can prove these 3 theorems, we first have to prove some lemmas.

5.1 Correctness of Projections

We first prove the lemmas related to the correctness of the selection of policy excerpts (Lemma 5.1), correctness of the projection of deltas (as Lemmas 5.2 and 5.3), and correctness of the initial projection of data sent to replicas (as Lemma 5.4).

Lemma 5.1 (Correct Construction of Policy Excerpt). *Given a role and list of privileges* $(priv \ldots)$, `excerpt-for-role`$[\![role, (priv \ldots)]\!]$ *constructs the correct local policy excerpt for that role.*

Proof. It follows directly from the definition of metafunction `excerpt-for-role` (see Figure 6 on Page 11) that exactly those privileges that apply to *role* are selected for the policy excerpt. Since

$$\frac{}{\texttt{matches-in-env}[\![(), (), \mathit{env}]\!]}\ [\texttt{empty-selector}]$$

$$\frac{\texttt{matches-in-env}[\![(\mathit{p\text{-}exp}\ \ldots), (k_2\ \ldots), \mathit{env}]\!]}{\texttt{matches-in-env}[\![(k_1\ \mathit{p\text{-}exp}\ \ldots), (k_1\ k_2\ \ldots), \mathit{env}]\!]}\ [\texttt{literal-key}]$$

$$\frac{\texttt{matches-in-env}[\![(k_1\ \mathit{p\text{-}exp}\ \ldots), (k_2\ \ldots), \mathit{env}]\!]}{\texttt{matches-in-env}[\![([\cup\ k_1\ k_3\ \ldots]\ \mathit{p\text{-}exp}\ \ldots), (k_2\ \ldots), \mathit{env}]\!]}\ [\texttt{union-first}]$$

$$\frac{\texttt{matches-in-env}[\![([\cup\ k_3\ \ldots]\ \mathit{p\text{-}exp}\ \ldots), (k_2\ \ldots), \mathit{env}]\!]}{\texttt{matches-in-env}[\![([\cup\ k_1\ k_3\ \ldots]\ \mathit{p\text{-}exp}\ \ldots), (k_2\ \ldots), \mathit{env}]\!]}\ [\texttt{union-other}]$$

$$\frac{\texttt{matches-in-env}[\![(\mathit{p\text{-}exp}\ \ldots), (k_2\ \ldots), \mathit{env}]\!]}{\texttt{matches-in-env}[\![(*\ \mathit{p\text{-}exp}\ \ldots), (k_1\ k_2\ \ldots), \mathit{env}]\!]}\ [\texttt{wildcard}]$$

$$\frac{\texttt{script-holds}[\![\mathit{script\text{-}op}, k_3, k_1]\!] \quad \texttt{matches-in-env}[\![(\mathit{p\text{-}exp}\ \ldots), (k_2\ \ldots), \mathit{env}]\!]}{\texttt{matches-in-env}[\![([\mathit{script\text{-}op}\ k_3]\ \mathit{p\text{-}exp}\ \ldots), (k_1\ k_2\ \ldots), \mathit{env}]\!]}\ [\texttt{script}]$$

$$\frac{\texttt{env-script-holds}[\![\mathit{script\text{-}op}, k_1, (k\ \ldots), \mathit{env}]\!] \quad \texttt{matches-in-env}[\![(\mathit{p\text{-}exp}\ \ldots), (k_2\ \ldots), \mathit{env}]\!]}{\texttt{matches-in-env}[\![([\mathit{script\text{-}op}\ (\sim k\ \ldots)]\ \mathit{p\text{-}exp}\ \ldots), (k_1\ k_2\ \ldots), \mathit{env}]\!]}\ [\texttt{env-script}]$$

Fig. 11. Verifying whether a path matches a path selector in a user environment in CommonLang.

any occurrence of the role wildcard * in the privileges is replaced with *role* (see Section 4.2.1), no (side-channel) leaks of information on other roles' privileges exists. A *role* receives information only on privileges that apply to *role* itself. □

Lemma 5.2 (Delta Projection Contagion Safety). *The leader accepts and propagates writes to fields only when the role of the writer is permitted to write to the affected fields.*

Proof. To handle *request*s of the form (PUSH-Δ ι^s (! *p atom*)), LeaderLang's `handle-request` uses `is-writable` (see Figure 8). `is-writable` checks whether a privilege exists that assigns a `WRITE` permission to a path selector that matches the written path: `matches-in-env`⟦*ps*, *p*, *env*⟧. The metafunction `matches-in-env` (see Figure 11) formalises a direct implementation of the notion of paths matching path selectors described in Section 3.1. Jointly, that means that a change (! *p atom*) is propagated exactly when the write is permitted. □

Lemma 5.3 (Delta Projection Leak Safety). *The leader propagates writes to exactly those target replicas that are permitted to read the affected field.*

Proof. To handle *request*s of the form (PUSH-Δ ι^s (! *p atom*)), the `handle-request` metafunction constructs (*action* …) = `actions-per-session`⟦(s_{other} …), (*user* …), (*excerpt* …), *d*, (! *p atom*)⟧, a list of PUSH-Δ *action*s to propagate (see Figure 8). The `actions-per-session` metafunction filters each session based on whether `is-readable`⟦*d*, *p*, (*priv* …), *env*⟧ holds, given the session's privileges and user environment. As shown in Figure 12, `is-readable` checks whether the tested path (k_1 …) is a prefix of a path that exists in *d* (i.e., one in (p_{all} …)), and where at least one `READ` or `WRITE` privilege permits access to. Since `WRITE` implies `READ`, and since the same logic holds for `matches-in-env` as in the proof for Lemma 5.2, leak safety holds during delta projection. □

$$\frac{(p_{all}\,\ldots) = \texttt{data-to-paths}[\![d]\!] \quad (k_1 \ldots k_2 \ldots) \in (p_{all}\,\ldots) \quad \texttt{matches-in-env}[\![ps, (k_1 \ldots k_2 \ldots), env]\!] \quad r/w \in (\texttt{READ WRITE})}{\texttt{is-readable}[\![d, (k_1 \ldots), (priv_l \ldots (\texttt{ALLOW}\ p\text{-}role\ r/w\ \texttt{OF}\ ps)\ priv_r \ldots), env]\!]}$$

Fig. 12. Verifying whether a path p in an object d is readable according a privilege and a user environment env, by verifying whether p is a matching prefix of an accessible path of d (expressed in CommonLang).

$$\texttt{readable-projection}[\![json_0, (priv \ldots), d, env, (k_{accum} \ldots)]\!] =$$
$$\begin{cases} ((k_1 := json_2)\ kj_3 \ldots) & \text{where } ((k_1 := json_1)\ kj_2 \ldots) = json_0, \\ & \quad \texttt{is-readable}[\![d, (k_{accum} \ldots k_1), (priv \ldots), env]\!], \\ & \quad json_2 = \texttt{readable-projection}[\![json_1, (priv \ldots), d, env, (k_{accum} \ldots k_1)]\!], \\ & \quad json_2 \neq (), \\ & \quad (kj_3 \ldots) = \texttt{readable-projection}[\![(kj_2 \ldots), (priv \ldots), d, env, (k_{accum} \ldots)]\!] \\ (kj_3 \ldots) & \text{where } ((k_1 := json_1)\ kj_2 \ldots) = json_0, \\ & \quad (kj_3 \ldots) = \texttt{readable-projection}[\![(kj_2 \ldots), (priv \ldots), d, env, (k_{accum} \ldots)]\!] \\ json_0 & \text{otherwise} \end{cases}$$

Fig. 13. LeaderLang: Selecting the readable projection of a data structure for a certain role.

Lemma 5.4 (Data Projection Leak Safety). *Data projection contains exactly those fields that the target role is permitted to read, i.e., if d is readable, then readable-projection*$[\![d, (priv \ldots), d, env, ()]\!]$ *(see Figure 13) contains exactly the fields of d that* $(priv \ldots)$ *permits reading.*

Proof. The induction hypothesis is that if $(priv \ldots)$ in *env* permits reading *json* which is situated at path p inside a replicated data structure d, then *readable-projection*$[\![json, (priv \ldots), d, env, p]\!]$ returns a structure containing exactly the subfields of *json* that $(priv \ldots)$ permits reading. This hypothesis holds for the call *readable-projection*$[\![d, (priv \ldots), d, env, ()]\!]$ in `handle-request` since an SRDT's root is part of each replica's projection. Hence, any list of privileges trivially permits reading the initial *json*, as that *json* is the replicated object's root d. The 3 clauses of *readable-projection* (see Figure 13) uphold this induction hypothesis as follows:

(1) If a $json_1$ exists at key k_1 within $json_0$, and if for the accumulated path that ends with that k_1 it holds that `is-readable`$[\![d, (k_{accum} \ldots k_1), (priv \ldots), env]\!]$, then $json_1$ is readable (since the same logic holds for `is-readable` as in the proof for Lemma 5.3), and its readable projection must be included in the projection.
By the induction hypothesis, $json_2$ = *readable-projection*$[\![json_1, (priv \ldots), d, env, (k_{accum} \ldots k_1)]\!]$ contains exactly those fields of $json_1$ that $(priv \ldots)$ permits reading. Also by induction, kj_3 = *readable-projection*$[\![(kj_2 \ldots), (priv \ldots), d, env, (k_{accum} \ldots)]\!]$ contains exactly the subfields of the rest of $(kj_2 \ldots)$ that $(priv \ldots)$ permits reading. The induction hypothesis is upheld by combining both results into one data structure $((k_1 := json_2)\ kj_3 \ldots)$.
(2) In the second case, subobject $json_1$ needs not be included in the projection. By the induction hypothesis, kj_3 = *readable-projection*$[\![(kj_2 \ldots), (priv \ldots), d, env, (k_{accum} \ldots)]\!]$ contains exactly the subfields of the rest of $(k_{accum} \ldots)$ that $(priv \ldots)$ permits reading. The induction hypothesis is upheld by returning that data structure $(kj_3 \ldots)$.
(3) The induction hypothesis trivially holds for any readable, non-compound $json_0$. □

5.2 Correctness of Security Policy Enforcement Locally at the Replicas

We now prove the lemma related to the local enforcement of security policies in ReplicaLang.

Lemma 5.5 (Write Permissions are Enforced in ReplicaLang). *Let program be a program in ReplicaLang, let role be the role for which the replica evaluating program has authenticated, let*

($priv_{\text{local}}$ ...) be the privileges from the policy excerpt for role, and let (k_1 ...k_2) be a non-empty (potentially invalid) path into the replica's projection of the SRDT.

If and only if a replica's user's role is permitted to write to the field at (k_1 ...k_2), then ReplicaLang's reduction relation $\rightarrow$ reduces a write operation (•! (ι^r (k_1 ...)) k_2 atom) to atom and updates the replica object's inner state by adding one δ to the local list of changes. Otherwise, the write operation reduces the program's entire expression to an (`error _`) expression, and leaves the list of replica objects unchanged. Any expression other than a locally permitted write operation does not modify a replica's local state, nor logs changes.

Proof. Consider the reduction relations of ReplicaLang, shown in Figure 10. The write operation we are interested in is only performed in `[write]`. In that clause, a write operation to replica ι^r is reduced. The replica object corresponding to ι^r is bound to r_c. The local privileges of ι^r are retrieved from r_c. By definition of "local privileges", the privileges (*priv* ...) in r_c are the local privileges ($priv_{\text{local}}$ ...). If at least one privilege in (*priv* ...) grants a `WRITE` permission to a path selector that matches (k_1 ...k_2) in environment *env*, then the judgment `is-writable`$[\![d, (k_1 \ldots k_2), (priv \ldots), env]\!]$ holds (see proof for Lemma 5.3). Hence, by definition, the judgment holds if *role* is permitted to write to the field at (k_1 ...k_2) by the local privileges. Hence, in this clause the premise of Lemma 5.5 holds, and we must therefore prove that in this clause exactly one corresponding δ is logged.

On the right-hand side of the reduction relation, a new δ = (! (k_1 ...k_2) *atom*) is logged in the replica object's list of changes. This δ corresponds to the write: it records the correct *atom* for the correct path (k_1 ...k_2) of the correct replica object ι^r. The requirement is thus met.

Every other clause has the form ((*r* ...)(*in-hole E e*)) $\rightarrow$ ((*r* ...) e'). In the absence of a write, no δ should be logged. Since (*r* ...) is left unchanged, the requirement is trivially met. □

5.3 Main Theorems

Finally, we prove the main three theorems as explained at the start of Section 5:

Theorem 5.6 (Freedom from Replicated Data Leaks). *A replica receives information only on the fields of an SRDT that the replica is permitted to read by the security policy. A replica is not informed of the fields that it is not permitted to read, nor on the privileges of other roles.*

Proof. Let ($priv_{\text{global}}$ ...) be the privileges that make up the full security policy for a replica identified by ι^r, and let *role* be a role specified in the security policy for ι^r. Let (ι^r ($priv_{\text{local}}$ ...) *d env* (δ ...)) be a replica object that is in scope at a ReplicaLang *program* that has authenticated for *role*. To prove Theorem 5.6 we prove that all members of the replica object are free from data leaks:

(1) This holds trivially for ι^r, the identifier of the local replica.
(2) ($priv_{\text{local}}$ ...) is the policy excerpt for *role*, listing only the name of *role* itself and the readable and writable (and hence also readable) fields (Lemma 5.1).
(3) The fields of *d* can come from three origins:
  (a) if the field was sent during initialisation, it is part of the readable projection, hence *role* is permitted to read the field according to ($priv_{\text{global}}$ ...) (Lemma 5.4),
  (b) if the field was pushed by the leader, it is part of the delta projection, i.e., *role* is permitted to read the field according to ($priv_{\text{global}}$ ...) (Lemma 5.3),
  (c) if the field was locally written to, no new non-local data is introduced. Freedom from Replicated Data Leaks holds trivially for locally produced data.
(4) This holds trivially for *env*, the environment of the local replica.
(5) The list (δ ...) contains only changes made locally by *program*. Freedom from Replicated Data Leaks holds trivially for locally produced data.

□

Theorem 5.7 (Freedom from Data Contagion). *A replica has write access only to those fields of an SRDT that the replica is permitted to write to by the security policy.*

Proof. Let ($priv_{\text{global}}$ ...) be the privileges that make up the security policy, and let *role* be a role specified in the security policy. A ReplicaLang *program* only writes to the fields of *d* which are writable according to ($priv_{\text{local}}$ ...) (Lemma 5.5). Since ($priv_{\text{local}}$ ...) is the correct policy excerpt of ($priv_{\text{global}}$ ...) for *role* (Lemma 5.1), it holds that ReplicaLang programs only write to fields of *d* which are writable according to ($priv_{\text{global}}$ ...). Finally, delta projection in LeaderLang only accepts and propagates writes that are permitted (Lemma 5.2).

Since no disallowed writes are performed by well-behaved clients, and since malicious clients' writes are rejected by the leader, disallowed writes to an SRDT do not impact the leader nor other replicas. Effectively, replicas only have write access to those fields of an SRDT that they are permitted to write to according to the security policy. □

Theorem 5.8 (Offline Policy Enforcement). *A replica retains offline availability, including eventual data consistency with the leader and the other replicas of the SRDT, even when enforcing the SRDT's security policy.*

Proof. Each locally permitted write in a ReplicaLang *program* is locally logged as a δ (Lemma 5.5). When those deltas are pushed to the leader, the leader propagates the deltas to all replicas permitted to read the field that was written to (Lemma 5.3). SRDTs hence offer the same form of eventual data consistency as the underlying replication mechanism as long as all locally performed writes are accepted by the leader during delta projection. Since the correct policy excerpt is contained in a ReplicaLang *program* (Lemma 5.1), all writes permitted by well-behaved clients are accepted by the leader. Since Theorems 5.6 and 5.7 also hold, the leader will eventually converge on a consistent state which accounts for all permitted writes on all replicas, and all replicas eventually see their role's data projection of that consistent state. □

6 RANDOMISED TESTING USING PLT REDEX

One of the main benefits of Redex is that the formal semantics becomes executable, and thus testable. Redex's randomised testing has been used successfully by Klein et al. [2012] to find errors in formal specifications and proofs (including mechanised proofs) in all of the 9 considered ICFP papers. In the same spirit, in Section 6.1 we briefly explain our suite of randomised testing to gain additional confidence in our claims from Section 5, and in Section 6.2 we discuss the issues uncovered during development that would likely have slipped into our formalism. While most of them are minor implementation bugs, some were unlikely to be found manually, and 1 bug had concrete security implications.

6.1 Randomised Testing of Read and Write Privileges

We designed 2 automated tests to verify the correct enforcement of read privileges (cf. Theorem 5.6) and write privileges (cf. Theorem 5.7). We briefly discuss the experimental set-up for both.

6.1.1 Randomised Verification of Read Privileges. For all possible security policies and objects, the goal is to verify that a client with a particular security role: (1) *can* read the fields allowed by the security policy, i.e., they are not accidentally omitted by LeaderLang, and (2) *cannot* read any other fields, i.e., they are not accidentally included by LeaderLang, which is a security violation.

Redex has features to generate random terms that adhere to the semantic entities of a Redex language, and to verify certain properties about those terms [Klein and Findler 2009]. However, completely random generation of *program* terms in ReplicaLang is extremely unlikely to yield *meaningful* objects, policy excerpts, and read expressions which adequately test a security policy.

Hence, we guided the generation of test cases by starting from a completely random generated (nested) object, and then programmatically extracting a random (but correct) security policy (including all types of path expressions) which grants read access to a random subset of the object. The object is then projected according to the security policy, and the test verifies for all possible paths in said object that: (1) every readable path according to the security policy *can* actually be read by a ReplicaLang *program*, and (2) that every other path is *not* present in the projected object (because it was correctly removed by LeaderLang's projection).

6.1.2 Randomised Verification of Write Privileges. The test setup to verify write privileges is similar to that for read privileges, but more convoluted because it also involves randomly generated roles and clients. Essentially, starting from a randomly generated (nested) object and multiple roles (including the wildcard role), we extract a random security policy from the object that contains both `READ` and `WRITE` privileges for a random subset of (some valid and some invalid) paths in the object. The security policy is used to project the generated object, and to verify that all roles are correctly able to read or write the fields that they should be able to access according to the policy. This means to verify that paths which are read-only or non-readable cannot be written to, and that all writable paths can actually be written to by a ReplicaLang *program*. Additionally, we verify that the returned list of `PUSH-Δ` terms by LeaderLang is correct, i.e., that if one client writes to a field, then the clients who are informed of the written value must have read privileges for the field.

6.2 Issues Detected Through Randomised Testing

We repeated the read and write tests 1,000,064 times (7813 tests per program instance, ran 128 times on a 64 core, 128 thread CPU), which we feel was more than sufficient to uncover any issues. Randomised tests found 10 problems in total, which were either found immediately, or in the worst-case after a couple thousand iterations of a single program instance. We categorise 7 of those problems as minor issues that constitute small implementation bugs in the Redex formalism, but which did not endanger the security of the model. For example, in one case, when trying to read a non-existing field from an object, a reduction of a term in LeaderLang would get stuck instead of rejecting the program. More interestingly, randomised testing also revealed 2 implementation bugs which we were unlikely to find manually, and 1 bug which had consequences for data security.

The most important of the identified bugs is the one which impacted data security. It relates to the erroneous handling of wildcards within the *readable-projection* metafunction. In essence, consider the following object which is a slightly reduced variant (for brevity) of the actual counterexample found by a random test, and read privileges for the path selector `(* JIvt)`.

```
1 ((7 := 0)
2  (r := ((x := JJ) (JIvt := #t))))
```

The expected projection is `((r := ((JIvt := #t))))`, such that a client can read the value of the path `(r JIvt)`. However, the projection erroneously included the top-level field 7 as well despite it not having a `JIvt` subfield. The formalism was meant to specify `(* JIvt)` to mean "traverse any field to find subfield `JIvt`", but the behaviour of our implementation was "traverse any field to find subfield `JIvt`, *or* admit access if the field contains an atom", thus erroneously revealing the data stored in the 7 field. If another client would write to the object such that the 7 field becomes `(7 := ((JIvt := 0)))`, then reading the content of that subfield of 7 would be permitted by the path selector `(* JIvt)`.

7 DISCUSSION AND RELATED WORK

7.1 Access Control in Replicated Databases

RDTs are used in the implementation of (geo-)replicated databases [Nadal 2023; Redis 2020; Riak 2013; Shukla 2018]. In such databases it is possible to encrypt data before passing it on to users [Barbosa et al. 2021; GUN 2022]. Encrypted database fields can be locally read and modified by clients with the decryption key, or by applying operations on homomorphically encrypted data (i.e., apply operations on encrypted data without having to decrypt it first). The advantage is that no synchronisation with a central authority is required. However, the approach is difficult to use for fine-grained access control due to difficulties in distributed key management. For example, when multiple clients have read access for one field, then they require the encryption/decryption key only for said field. When roles have access to multiple fields, then it is up to the developer to figure out which (potentially overlapping) fields must be encrypted with different keys, how those keys are distributed to each client, and tracking which keys must be used for which fields. This approach does not scale beyond a couple of fields with little or no overlap between roles.

Rather than replicating a database to users, many distributed database management systems (DBMS) have the option of using replication internally to increase availability and reduce latency. In this case replication is an implementation concern, and not part of the programming model. Data is not locally available at the clients, which instead must query the DBMS. Clients could use RDTs to make the data available offline to them, but then they again face the original problems of insecure RDTs.

7.2 Access Control for XML Documents

In the early to mid 2000's XML was thought to be the future format for data interchange between systems. There is a body of work to enforce access control for XML documents such that only parts of documents are exposed to different users [Crampton 2006; Damiani et al. 2002; Fundulaki and Marx 2004; Murata et al. 2006]. We based the specification of our security policy language on this work, resulting in our policy language using JSONPath versus their use of XPath, as well as similar security policy semantics. The run-time enforcement of security for XML documents is not directly applicable to offline available replicated JSON data because security constraints are imposed only when an XML document is fetched, whereas replicated data is continuously (locally) modified.

7.3 Multitier Programming

Several tools and languages exist to develop (web) applications as a single code base which is automatically split into the multiple tiers of a distributed application (client, server, …), e.g., Hop.js [Serrano and Prunet 2016], ScalaLoci [Weisenburger et al. 2018], and Stip.js [Philips et al. 2018]. Some of this work explicitly targets security, such as Swift [Chong et al. 2009] and Fabric [Liu et al. 2017], which use source code annotations to specify security constraints on code and information flow. These tools are an alternative programming model for developing secure-by-design distributed (web) applications. This paper departs from the fact that using replication to offer offline availability is a given, as it has been widely motivated and is being used in practice. Multitier programming does not tackle the same concerns as SRDTs, namely offline availability, eventual consistency, and secure access control for replicated data.

7.4 Byzantine Fault Tolerance for RDTs

Malicious entities can try to circumvent an RDT's security mechanisms by attacking the algorithm that the RDT uses to establish consensus. In the CRDT literature, a CRDT that can retain a mutually agreed ordering of data updates among replicas (needed for eventual consistency), even in the

presence of malicious clients, is said to be *Byzantine fault tolerant* [Kleppmann 2022; van der Linde et al. 2020]. In principle, the challenges posed by Byzantine faults are orthogonal to the approach in this paper. A solution that combines our approach (to enforce access control on the data on the application-level), with a technique such as the one by Yactine et al. [2021] on the implementation level would yield a Byzantine fault tolerant SRDT. However, since SRDTs require a central authority, in their current form there is no need for Byzantine fault tolerance to maintain eventual consistency.

7.5 Limitations and Future Work

The approach outlined in this paper can serve as a foundation for building advanced security features for RDTs. We briefly discuss the main limitations and possible avenues for future work.

7.5.1 Central Authority. One of the main limitations compared to other work on RDTs is the assumption of a single leader. Whereas RDTs are frequently used *because* they support decentralisation, there is also a need for RDTs in centralised designs. For example, in academia there is AutoCouch, a JSON CRDT framework which combines Automerge with CouchDB to support offline availability in client-server web applications [Grosch et al. 2020]. In industry, a central authority is already available and often desired. A prime example is Figma, a collaborative web application for user interface design (acquired in 2022 by Adobe for around $20 billion [Adobe 2022]), which uses CRDTs for offline availability and conflict resolution, while explicitly omitting decentralisation. They note that: "*Even if you have a client-server setup, CRDTs are still worth researching because they provide a well-studied, solid foundation to start with.*" [Wallace 2019] A full peer-to-peer implementation of SRDTs is interesting, but requires additional research (e.g., possibly to relax the security guarantees), and is outside the scope of this paper.

7.5.2 Arrays. There are unsolved semantic issues when applying access control to arrays, e.g., to access only a part of an array. The problem is that the contents of the array can change at any moment in time, including when a client is offline. Questions arise such as how the accessible parts of an array are represented on a client (e.g., are non-accessible entries removed and indices remapped?), and how to deal with retractions of access, e.g., when an object in an array is moved from an accessible part to a non-accessible part. How the programming model should be adapted to solve or avoid these semantic issues is an open problem. Rather than ignoring these issues, for now SRDTs cannot contain arrays. Note that SRDTs can still be used to build collections of items, but using an object's fields as opposed to an automatically indexed array.

7.5.3 Extended Policy Language. We deliberately kept the policy language small by offering only simple permissions (read or write). Real-world security policies specify privileges that currently cannot be expressed in the policy language, such as role hierarchies [Sandhu 1998], `DENY` privileges, and inter-field constraints [Oostvogels et al. 2017].

7.5.4 Expanded Set of Policy Attributes. The current security policy language disallows privileges to depend on data in the RDT because this causes unresolved semantic difficulties. More specifically, an update to an RDT field whose value is used in a security policy could cause a client (including the one who updated the field) to lose access to fields. How to deal with retractions of access is an open issue. Additionally, malicious users may try to widen the scope of their privileges by writing to fields that are used in the security policy. A solution can be a part of a larger effort towards Attribute-Based Access Control, where a security policy may depend on run-time data, resources, system environment, connection, and administrative decisions [Servos and Osborn 2017].

7.5.5 Static Enforcement. Clients are expected to dynamically check whether they are allowed to write to fields of an SRDT. Including such a security check (e.g., as a library call) is undesirable

because it mixes the enforcement of security policies and application logic. Practical solutions should statically enforce the security policy, e.g., via a type system that rejects programs that write to read-only fields. Similar to the original work on JSON CRDTs [Kleppmann and Beresford 2017], we leave the specification of a data schema as future work.

7.5.6 Schema Migration. Support for schema migration and updates to the security policy of an already deployed SRDT are open problems.

7.5.7 Compound Changes. The language features supported by ReplicaLang are hampered by the lack of support for writing compound data structures to fields, or moving subtrees in an RDT. Recent work by Kleppmann et al. [2022] adds these features for JSON CRDTs, but our formalism cannot yet verify these features' security.

7.5.8 State-based RDTs. SRDTs assume an underlying operations-based RDT like Automerge [Automerge Contributors 2023] or Yjs [Jahns and Yjs Contributors 2023]. Future work can apply the results to state-based or delta-based [Rinberg et al. 2022] implementations as well, though 2 challenges need to be addressed. First, for state-based RDTs, computing "state deltas" at the leader to verify that all changed fields were permitted to be written to. Second, designing a consistency mechanism which merges a partial state (namely, the part of the state that is in the projected data of some client) with the complete state at the leader.

8 CONCLUSION

This paper proposes SRDTs, a data type that specifies role-based access control for RDTs. This is an important step towards practical implementations of RDTs for applications with extra security constraints such as business applications, especially when parts of the application are ran in unsecured environments such as ordinary web browsers. Concretely we identified 3 problems, namely (1) Replicated Data Leaks, where sensitive data is inadvertently replicated to clients which should not have that data, (2) Data Contagion, where modifications of a client to a local replica will be merged with the replicas of other clients as well regardless of whether those changes were permitted, and finally, (3) the Lack of Offline Policy Enforcement, where any enforcement mechanism must be available offline.

To overcome the identified problems, SRDTs demonstrate a combination of Role-Based Access Control and offline-available JSON data to securely replicate said data over a network. To prevent Replicated Data Leaks, a leader defines multiple projections to exclude any data for which a client with a particular role does not have read privileges. To prevent Data Contagion, a leader acts as an intermediary between all clients to prevent malicious writes (that do not conform to the security policy) from reaching other clients. Finally, to enable Offline Policy Enforcement, each client receives an excerpt of the global security policy which contains the privileges that apply to their role, such that it can be enforced locally.

An operational semantics of SRDTs was implemented in PLT Redex. We validated this specification via formal proofs that verify that SRDTs do not suffer from the identified problems, and that the underlying properties of RDTs (such as eventual consistency) are unaffected. Furthermore, we used randomised testing to experimentally check the absence of the identified problems, which uncovered multiple bugs and 1 security problem that existed in earlier versions of the formal specification.

DATA-AVAILABILITY STATEMENT

The full executable implementation of the formal specification in Redex (Section 4) is available as a software artifact [Renaux et al. 2023]. This specification was used for the randomised testing of

Section 6. Furthermore, we provide an easy to use command-line interface (not discussed in this paper) to interact with SRDTs (e.g., via the running example of Section 3). The artifact is available on Zenodo via the following link: https://doi.org/10.5281/zenodo.8310917.

ACKNOWLEDGMENTS

Sam Van den Vonder was funded by the Flanders Innovation & Entrepreneurship (VLAIO) "Cybersecurity Initiative Flanders" program. Thierry Renaux was partly funded by the Flanders Innovation & Entrepreneurship (VLAIO) "Cybersecurity Initiative Flanders" program, and the Innoviris "SWAMP" project.

REFERENCES

Adobe. 2022. Adobe to Acquire Figma. https://web.archive.org/web/20230619114811/https://news.adobe.com/news/news-details/2022/Adobe-to-Acquire-Figma/default.aspx Accessed on 2023-06-19.

Automerge Contributors. 2023. Automerge CRDT. http://web.archive.org/web/20230118145244/https://automerge.org/ Accessed on 2023-01-18.

Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. 2021. Secure Conflict-free Replicated Data Types. In *ICDCN '21: International Conference on Distributed Computing and Networking, Virtual Event, Nara, Japan, January 5-8, 2021*. Association for Computing Machinery, New York, NY, USA, 6–15. https://doi.org/10.1145/3427796.3427831

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2009. Building secure web applications with automatic partitioning. *Commun. ACM* 52, 2 (2 2009), 79–87. https://doi.org/10.1145/1461928.1461949

James Clark and Steven DeRose. 1999. *XML Path Language (XPath) Version 1.0*. W3C Recommendation. W3C. http://web.archive.org/web/20230209115333/https://www.w3.org/TR/1999/REC-xpath-19991116/ Accessed on 2023-02-09.

Jason Crampton. 2006. Applying hierarchical and role-based access control to XML documents. *Comput. Syst. Sci. Eng.* 21, 4 (2006), 325–338.

Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. 2002. A fine-grained access control system for XML documents. *ACM Trans. Inf. Syst. Secur.* 5, 2 (5 2002), 169–202. https://doi.org/10.1145/505586.505590

Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. 2019. The role of the adversary model in applied security research. *Comput. Secur.* 81 (2019), 156–181. https://doi.org/10.1016/j.cose.2018.12.002

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press, Cambridge, MA, USA.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (2 2018), 62–71. https://doi.org/10.1145/3127323

Jeff Friesen. 2019. *Extracting JSON Values with JsonPath*. Apress, Berkeley, CA, USA, Chapter 10, 299–322. https://doi.org/10.1007/978-1-4842-4330-5_10

Irini Fundulaki and Maarten Marx. 2004. Specifying access control policies for XML documents with XPath. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies, SACMAT 2004, Yorktown Heights, New York, USA, June 2-4, 2004, Proceedings*. Association for Computing Machinery, New York, NY, USA, 61–69. https://doi.org/10.1145/990036.990046

Stefan Goessner. 2007. JSONPath - XPath for JSON. http://web.archive.org/web/20230208122216/https://goessner.net/articles/JsonPath/ Accessed on 2023-02-09.

Pascal Grosch, Roman Krafft, Marcel Wölki, and Annette Bieniusa. 2020. AutoCouch: a JSON CRDT framework. In *7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2020, Heraklion, Greece, April 27, 2020*. Association for Computing Machinery, New York, NY, USA, 6:1–6:7. https://doi.org/10.1145/3380787.3393679

GUN. 2022. GUN: How to add security. http://archive.today/2022.10.13-090614/https://gun.eco/docs/Auth Accessed on 2022-10-13.

ISO Central Secretary. 2017. *Information technology – The JSON data interchange syntax*. Technical Report ISO/IEC 21778:2017. International Organization for Standardization, Geneva, CH. http://web.archive.org/web/20230118145850/https://www.iso.org/standard/71616.html

Kevin Jahns and Yjs Contributors. 2023. Yjs – Shared data types for building collaborative software. https://web.archive.org/web/20230622134343/https://github.com/yjs/yjs Accessed on 2023-06-22.

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run Your Research: On the Effectiveness of Lightweight

Mechanization. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Association for Computing Machinery, New York, NY, USA, 285–296. https://doi.org/10.1145/2103656.2103691

Casey Klein and Robert Bruce Findler. 2009. *Randomized testing in PLT Redex.* Technical Report. California Polytechnic State University, CA, USA. 26–36 pages. Proceedings of the ACM SIGPLAN Workshop on Scheme and Functional Programming, Northeastern University, Boston, Massachusetts, August 22, 2009, Cal Poly TR CPSLO-CSC-09-03.

Martin Kleppmann. 2022. Making CRDTs Byzantine Fault Tolerant. In *PaPoC@EuroSys 2022: Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data, Rennes, France, April 5 - 8, 2022*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3517209.3524042

Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Trans. Parallel Distrib. Syst.* 28, 10 (10 2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

Martin Kleppmann and Alastair R Beresford. 2018. Automerge: Real-time data sync between edge devices. 1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018), Computer Laboratory, Cambridge, United Kingdom, September 12-13, 2018.

Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. 2022. A Highly-Available Move Operation for Replicated Trees. *IEEE Trans. Parallel Distributed Syst.* 33, 7 (2022), 1711–1724. https://doi.org/10.1109/TPDS.2021.3118603

Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. 2017. Fabric: Building open distributed systems securely by construction. *J. Comput. Secur.* 25, 4-5 (2017), 367–426. https://doi.org/10.3233/JCS-15805

Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2020. A Semantics for the Essence of React. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Wadern, Saarland, Germany, 12:1–12:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.12

Meta Platforms. 2023a. React – A JavaScript library for building user interfaces. http://web.archive.org/web/20230116084323/https://reactjs.org/ Accessed on 2023-01-19.

Meta Platforms. 2023b. React Native: Learn once, write anywhere. http://web.archive.org/web/20230119102108/https://reactnative.dev/ Accessed on 2023-01-19.

Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. 2006. XML access control using static analysis. *ACM Trans. Inf. Syst. Secur.* 9, 3 (8 2006), 292–324. https://doi.org/10.1145/1178618.1178621

Mark Nadal. 2023. GUN: An open source cybersecurity protocol for syncing decentralized graph data. http://web.archive.org/web/20230309093226/https://github.com/amark/gun Accessed on 2023-03-09.

Natuurpunt. 2023. Over Natuurpunt. http://web.archive.org/web/20230210103528/https://www.natuurpunt.be/pagina/over-natuurpunt Accessed on 2023-02-10.

Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *Engineering the Web in the Big Data Era*. Springer International Publishing, Cham, Switzerland, 675–678. https://doi.org/10.1007/978-3-319-19890-3_55

Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. 2017. Inter-parameter Constraints in Contemporary Web APIs. In *Web Engineering*. Springer International Publishing, Cham, CH, 323–335. https://doi.org/10.1007/978-3-319-60131-1_18

OWASP Foundation. 2021. OWASP Top 10 Web Application Security Risks. http://web.archive.org/web/20220823114651/https://owasp.org/Top10/ Accessed on 2022-08-23.

Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2018. Search-based Tier Assignment for Optimising Offline Availability in Multi-tier Web Applications. *Art Sci. Eng. Program.* 2, 2 (2018), 3. https://doi.org/10.22152/programming-journal.org/2018/2/3

Redis. 2020. Diving into CRDTs. http://web.archive.org/web/20220311150856/https://redis.com/blog/diving-into-crdts/ Accessed on 2022-03-11.

Thierry Renaux, Sam Van den Vonder, and Wolfgang De Meuter. 2023. Secure RDTs: Enforcing Access Control Policies for Offline Available JSON Data (Artifact). https://doi.org/10.5281/zenodo.8310917

Riak. 2013. Introducing Riak 2.0: Data Types, Strong Consistency, Full-Text Search, and Much More. https://web.archive.org/web/20211217132134/https://riak.com/introducing-riak-2-0/ Accessed on 2022-03-11.

Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma. 2022. DSON: JSON CRDT Using Delta-Mutations For Document Stores. *Proc. VLDB Endow.* 15, 5 (2022), 1053–1065.

Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

Ravi S. Sandhu. 1998. Role Activation Hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control, RBAC 1998, Fairfax, VA, USA, October 22-23, 1998*. Association for Computing Machinery, New York, NY, USA, 33–40. https://doi.org/10.1145/286884.286891

Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *IEEE Computer* 29, 2 (1996), 38–47. https://doi.org/10.1109/2.485845

Manuel Serrano and Vincent Prunet. 2016. A glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Association for Computing Machinery, New York, NY, USA, 180–192. https://doi.org/10.1145/2951913.2951916

Daniel Servos and Sylvia L. Osborn. 2017. Current Research and Open Problems in Attribute-Based Access Control. *ACM Comput. Surv.* 49, 4, Article 65 (jan 2017), 45 pages. https://doi.org/10.1145/3007204

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. INRIA – Centre Paris-Rocquencourt. 50 pages. http://web.archive.org/web/20220706130124/https://hal.inria.fr/inria-00555588

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

Dharma Shukla. 2018. Azure Cosmos DB: Pushing the frontier of globally distributed databases. http://web.archive.org/web/20220311153520/https://azure.microsoft.com/en-us/blog/azure-cosmos-db-pushing-the-frontier-of-globally-distributed-databases/?cdn=disable Accessed on 2022-03-11.

Albert van der Linde, João Leitão, and Nuno Preguiça. 2020. Practical Client-Side Replication: Weak Consistency Semantics for Insecure Settings. *Proc. VLDB Endow.* 13, 12 (7 2020), 2590–2605. https://doi.org/10.14778/3407790.3407847

Katrin Vohland, Anne Land-Zandstra, Luigi Ceccaroni, Rob Lemmens, Josep Perelló, Marisa Ponti, Roeland Samson, and Katherin Wagenknecht. 2021. *The Science of Citizen Science* (1 ed.). Springer Nature Switzerland, Cham, Switzerland. https://doi.org/10.1007/978-3-030-58278-4

Evan Wallace. 2019. How Figma's multiplayer technology works. https://web.archive.org/web/20230619114122/https://www.figma.com/blog/how-figmas-multiplayer-technology-works/ Accessed on 2023-06-19.

Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 129:1–129:30. https://doi.org/10.1145/3276499

Houssam Yactine, Ali Shoker, and Georges Younes. 2021. ASPAS: As Secure as Possible Available Systems. In *Distributed Applications and Interoperable Systems*. Springer International Publishing, Cham, Switzerland, 57–73. https://doi.org/10.1007/978-3-030-78198-9_4

Received 2023-04-14; accepted 2023-08-27