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

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

1 INTRODUCTION

The paper associated with this artifact specifies the concept of an SRDT, a secure replicated data type. To specify exactly how SRDTs work and to verify that their approach is secure, the paper uses a formal specification implemented in Redex [Klein et al. 2012; Klein and Findler 2009], a library in the Racket language to specify *executable* formal semantics. The purpose of this artifact is to guide the reader on how to interacting with the formal semantics, such that they can explore exactly how SRDTs work, are able to verify the claims of the paper, and are able to reproduce SRDTs for other systems.

The paper makes the 3 following claims:

1. **Full formal specification:** Whereas the paper typesets the core parts of the formal semantics as figures in a typical formal notation, the artifact provides the full implementation in Redex which can be read, executed, and extended by domain experts. While we do not explain the implementation details, Section 3 explains how the paper maps to the Redex implementation, and how to typeset the figures from the paper with Redex.
2. **A solution for the problem statement:** The problem statement in the paper's Section 2.3 lists 3 main problems, namely Replicated Data Leaks, Data Contagion, and the Lack of Offline Policy Enforcement. To verify that SRDTs have solved these issues, one must inspect the formal specification, specifically the formal languages ReplicaLang and LeaderLang which model a client's and server's operations respectively. This is not easy to do, since ReplicaLang and LeaderLang do not directly interact with each other, i.e., given a certain input they model which steps the system undertakes for a single client and server respectively. This is sufficient to formally validate their security aspects, but the separate languages do not by themselves give rise to a usable system with an interface that can be programmed against (e.g., a library). Hence it is very difficult to observe, explore, and experiment with the security features of SRDTs. To overcome these issues, this artifact provides a reusable CLI tool that automates this laborious and error-prone task, and which allows users to interact via simple text prompts with a complete SRDT system that consists of multiple clients and a single server. Section 4 uses the CLI tool to run through different scenarios with which the paper's claims can be verified.
   2.1. *Absence of Replicated Data Leaks:* The scenarios in Sections 4.1 and 4.2 show that sensitive data is not leaked to clients which are not allowed to read said data according to the security policy.
   2.2. *Absence of Data Contagion:* The scenario in Section 4.3 shows that when a client turns malicious, their malicious changes will not affect well-behaved clients.
   2.3. *Availability of Offline Policy Enforcement:* The scenario in Section 4.1 will show that each client is sent a security policy excerpt, thus enabling Offline Policy Enforcement.
3. **Randomised testing:** The paper's Section 6 claims that we used randomised testing to uncover issues in the formalism. In Section 5 we show how to run these tests using the same testing framework that we used for the paper.

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

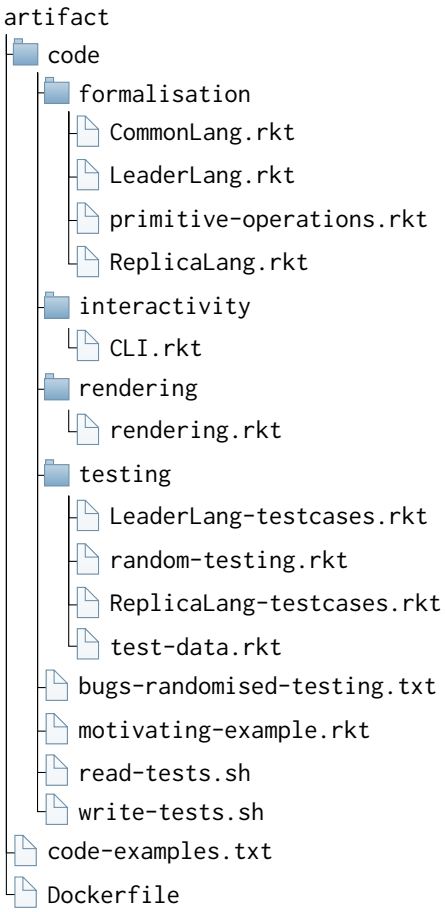

Fig. 1. Directory structure of the artifact.

For convenience, Figure 1 provides an overview of the files included in the artifact.

2 GETTING STARTED

The software artifact is written entirely in the Racket language and can be run directly using a standard installation of the DrRacket editor, without any additional packages or software. We include instructions to run Racket in Docker. Note that choosing to proceed with a Docker is sufficient to run and interact with all of the code examples of the artifact, i.e., to verify the claims of the paper. However, due to implementation details of the Racket software, generating the images of the formal specification (including those which we included in the main paper) is not possible from the Dockerized command-line version (which lacks a graphical environment).

Whenever we provide code samples or terminal commands to copy and paste, the used PDF reader may incorrectly copy the selected text. For example, by adding spaces where there should be none, by changing (unicode) characters, etc. Copied text should be checked for such errors before execution throughout the instructions of this artifact.

After choosing to run the software natively (using DrRacket) or in Docker in Section 2.1, and following the corresponding requirements, proceed to Section 2.2 (when running natively) or Section 2.3 (when running Docker) to perform a basic integrity check.

2.1 Software Requirements

We tested the software to run on MacOS 13.4.1, Windows 10 Home, and Arch Linux. The following software is required:

2.1.1 To Run Natively Using DrRacket. Installers are available for all major platforms (MacOS, Linux, Windows) and architectures (x86, Apple Silicon, ARM).[1] We tested the software to work on Windows 10 Home, MacOS 13.4.1 (Intel + Apple Silicon), and Arch Linux. We used DrRacket version 8.9 (May 2023). The basic integrity checks of Section 2.2 are known to fail with at least DrRacket v7.9 and v8.5 (other versions were not tested).

2.1.2 To Run Using Docker. Installers are available for all major platforms.[2] We tested the following versions and platforms:

- MacOS 13.4.1 (Intel + Apple Silicon): We used Docker Desktop 4.21.1. Since the used Docker image relies on the "`linux/amd64`" operating system and architecture, in addition to Intel-based MacOS we also tested it to work on MacOS 13.4.1 running on Apple Silicon, which automatically uses the standard Rosetta 2 emulation layer for x86 programs.
- Windows 10 Home: Docker Desktop 4.12.0.
- Arch Linux x86_64 6.4.1-arch2-1: Docker Engine version 24.0.2. Note that future versions of Docker on Linux will no longer ship with Docker build tools by default, and users are expected to install either the Docker Buildx[3] or Docker Desktop which includes Buildx by default.

2.2 Getting Started With DrRacket

Step 1: Open and run the file "`code/testing/LeaderLang-testcases.rkt`" in DrRacket. Running the file executes some test cases which we manually defined, and the last line of the output should print: "`All 9 tests passed.`"

Step 2: Open and run the file "`code/testing/ReplicaLang-testcases.rkt`" in DrRacket. The last line of the output should print: "`All 35 tests passed.`"

2.3 Getting Started With Docker

Step 1: Open a terminal in the root folder of the artifact (the folder which contains the `code` folder and `Dockerfile`).

Step 2: Build the docker image specified by `Dockerfile` via the following command:

```
docker build -t srdt .
```

The `-t` argument will tag the image such that it can be referred to via the name `srdt`. The Dockerfile will copy the "`code`" folder to the root of the filesystem, i.e., "`/code/`" in the created Docker container. This command can be repeated to refresh or update the code.

Step 3: Run and interact with the image via the following command. This will open a Racket read-eval-print loop (REPL) running in Docker. The `--rm` flag indicates to Docker that the container may be automatically removed when it stops running (e.g., by stopping the REPL).

[1] DrRacket: https://download.racket-lang.org/

[2] Docker: https://www.docker.com/get-started/

[3] Docker Buildx: https://docs.docker.com/build/architecture

```
docker run --rm -it srdt
```

Step 4: In the Racket REPL which was just opened, run a basic integrity test by executing some manual test-cases that we defined for the formal languages. Enter and run the following expression in the Racket REPL:

```
(enter! (file "/code/testing/LeaderLang-testcases.rkt"))
```

The last line of the printed output should read: "`All 9 tests passed.`" Now enter and run the following expression in the Racket REPL:

```
(enter! (file "/code/testing/ReplicaLang-testcases.rkt"))
```

The last line of the printed output should read: "`All 35 tests passed.`"

3 INSTRUCTIONS: FORMAL SPECIFICATION AND RENDERING IMAGES

The complete formal specification can be found in the "`code/formalisation`" folder of the artifact. The respective files `CommonLang.rkt`, `LeaderLang.rkt`, and `ReplicaLang.rkt` implement the 3 different formal languages that are introduced by the main paper in Section 4. Other than documenting this code and making it available for inspection (via DrRacket or an ordinary text editor), this artifact does not intend to elaborate further on the specifics of the implementation, since they require knowledge on the syntax and evaluation model of Redex. We recommend the Redex documentation for an introduction to Redex[4], the book by Felleisen et al. [2009] for in-depth knowledge on Redex, and the paper by Klein and Findler [2009] for more information about randomised testing in Redex.

What follows is a brief overview of the code in relation to the paper:

CommonLang.rkt Implements the paper's `CommonLang` from Section 4.1, as well as Figures 11-12 from Section 5.

LeaderLang.rkt Implements the paper's `LeaderLang` from Section 4.2, as well as Figure 13 from Section 5.

ReplicaLang.rkt Implements the paper's `ReplicaLang` from Section 4.3.

The figures from the paper were obtained from Redex itself, sometimes with slight modifications in Adobe Illustrator to add explanatory notes or to reposition elements to fit within the allowable space. The code responsible to typeset the paper's figures can be found in the file "`code/rendering/rendering.rkt`". Note that this file requires a graphical environment, and that we only provide instructions to reproduce the paper's images when using DrRacket. The fonts used in the paper are Linux Libertine[5] and JuliaMono[6]. When these fonts are not present on the system, Redex will silently substitute them with default fonts.

⚠ Due to implementation details of the Racket software, generating the images of the formal specification (including those which we included in the main paper) is not possible from the Dockerized command-line version (which lacks a graphical environment).

The file "`code/rendering/rendering.rkt`" defines various procedures that can be run from DrRacket's REPL. After opening and running the file in DrRacket, the following functions can be executed without arguments, e.g., `(render-commonlang)`.

[4]Redex documentation and tutorial: https://docs.racket-lang.org/redex/
[5]Linux Libertine: https://github.com/iliakonnov/linlibertine
[6]JuliaMono: https://juliamono.netlify.app/

- `render-commonlang` (Figure 5 from the paper)
- `render-replicalang` (Figure 9 from the paper)
- `render-leaderlang` (Figure 7 from the paper)
- `render-is-readable` (Figure 12 from the paper)
- `render-matches-in-env` (Figure 11 from the paper)
- `render-red-replica` (Figure 10 from the paper)
- `render-handle-request` (Figure 8 from the paper)
- `render-excerpt-for-role` (Figure 6 from the paper)
- `render-readable-projection` (Figure 13 from the paper)

Each function takes an optional argument to render the image as a PDF file and to save it to a path on the filesystem, e.g.:

```
(render-commonlang "./CommonLang.pdf")
```

4 INSTRUCTIONS: INTERACTING WITH SRDTS

Instead of requiring readers to look at the code of the formal specification and essentially "play interpreter" by mentally executing the code themselves (requiring in-depth Redex knowledge), we built an interactive CLI tool that allows a reader to interact with SRDTs. Given a configuration of users, a security policy, and a data model, the CLI tool is used simulate simulate SRDT interactions between multiple clients and a secure server, i.e., multiple instances of ReplicaLang and one instance of LeaderLang. In this section we explain how to interact with the motivating example from the paper, and in Section 4.4 how to run a custom example.

4.1 The Basics of the CLI Tool

The motivating example of the paper consists of a security policy (Listing 1 in the paper) and a data model (Listing 2 in the paper). The same is contained within "`code/motivating-example.rkt`". Execute this file to start the CLI tool.

When using Docker: Enter and run the following expression in the Racket REPL:

```
(enter! (file "/code/motivating-example.rkt"))
```

The expression may take a few seconds to execute before control is handed back to the user, while Racket compiles and runs the code.

When using DrRacket: Open and run the file "`code/motivating-example.rkt`".

The output printed by the CLI tool will match Listing 1 (note that the formatting and indentation may differ). Most of the items are given for information only: Lines 2 to 16 prints the complete data model (this will match Listing 2 from the paper), Lines 18 to 26 prints the complete security policy (this will match Listing 1 from the paper), and an interaction prompt by the CLI is displayed on Lines 28 to 34. You can now interact with the SRDT.

Since the motivating example has 3 users, the CLI specifies 3 options related to the users that take part in the replication. Users 0 (Alice) and 1 (Bob) are ordinary users with the `user` security role, and user 2 (Erin) has the `biologist` role. Based on these users we can verify one of the basic claims of the paper, namely that users who do not have `READ` privileges for certain fields of the SRDT will not be able to read said data. For example, according to the security policy Alice is an ordinary user, and their user environment says that they are part of `team1`. The CLI can be used to inspect which data is visible to Alice.

```
<<< SRDT Interaction Simulator >>>
>>> The full program data is:
'((team1 :=
   ((name := "The Fantastical Scouts")
    (sightings := ((1674813931967 :=
                    ((location := ((lat := 51.06038) (lng := 4.67201)))
                     (species := "???")
                     (photo := "blob:...")
                     (points := 3)
                     (feedback := "Do not eat this!")))))))
  (team2 :=
    ((name := "The Brilliant Bunch")
     (sightings := ((1674814951967 :=
                     ((location := ((lat := 51.06066005703071) (lng :=
                         4.674296375850668)))
                      (species := "Spicy fly")
                      (photo := "blob:...")))))))

>>> The full security policy is:
'((ALLOW * READ OF (* name))
  (ALLOW biologist READ OF (* sightings * *))
  (ALLOW biologist READ OF (* sightings * location *))
  (ALLOW biologist WRITE OF (* sightings * (∪ points feedback)))
  (ALLOW user READ OF (* sightings * (∪ photo points)))
  (ALLOW user READ OF ((= (~ my-team)) sightings * feedback))
  (ALLOW user WRITE OF ((= (~ my-team)) sightings * (∪ species photo)))
  (ALLOW user WRITE OF ((= (~ my-team)) sightings * location (∪ lat lng))))

Select an option.
   [0] Interact as Alice (role: "user", user environment: '((my-team := 'team1))).
   [1] Interact as Bob (role: "user", user environment: '((my-team := 'team2))).
   [2] Interact as Erin (role: "biologist", user environment: '()).
   [3] Display the leader's replica data
   [4] Display the leader's changes which are not yet pulled by a user (0 changes)
   [5] Exit program
Enter a number >
```

Listing 1. Initial output of the CLI tool when running the motivating example.

- Insert the number 0 and press enter to execute it.[7] You are now interacting with the SRDT as Alice, and are presented with 7 new options.

Now inspect Alice's local replica data (this corresponds to the paper's Section 3.3 about projecting replica data).

- Execute action 0. The CLI prints the output starting with "`>>> Alice's replica's data is:`" followed by the data that is visible to Alice, as well as a new interaction prompt with the same options as previously (for Alice). Take note that Alice can see only a limited set of fields of the `team2` object, because (sub-)fields like `location` and `species` were removed by the leader when it projected the data according to Alice's security privileges.

Now inspect Alice's security policy excerpt (this corresponds to the paper's Section 3.2 about projecting a security policy).

[7] Henceforth we will simply say "Execute action x" to perform an interaction in the interactive prompt.

- Execute action 1. The output displays Alice's security policy excerpt, which contains only the privileges which are relevant for Alice.

4.2 Running a Well-Behaved ReplicaLang Program

Now that the basics of the CLI have been established, we will execute a scenario presented in Section 4.3 of the paper that explains how to go from the paper's Listing 3 to Listing 4. In the scenario, a user with the `biologist` role adds feedback to a particular sighting by running a ReplicaLang program. Since the data in the running program already contains the exact `feedback` field which was written to in the paper, we will perform the same interaction but on `team2`'s data instead of `team1`.

⚠ Make sure you are at the interaction prompt's main menu. When proceeding from Section 4.1, go back by executing action 7.

Step 1: Execute action 2 to interact as Erin the biologist.

Step 2: Execute action 3 to start executing a ReplicaLang program as Erin. You are prompted to enter a ReplicaLang program.

Step 3: Enter and execute the following ReplicaLang program which performs the interaction of the paper. Due to the special characters within the code which are often incorrectly copied from a PDF file, it is best to copy this code from "code example 1" stated in the artifact's "`code-examples.txt`".

```
(let ((cr (root "my-replica")))
  (let ((sighting (· (· (· cr team2) sightings) 1674814951967)))
    (·! sighting feedback "This is actually an Andrena (solitary bee)")))
```

When the program has executed, the output printed above the new interaction prompt consists of the fully reduced ReplicaLang program (after "`>>> Executing ReplicaLang program...`", as well as the result which the code snippet reduced to (after "`>>> The expression evaluated to`"). The reduced ReplicaLang program is similar to the paper's Listing 4. In particular, compare the last few lines which indicate that a value was written (the term starting with "`!`") as well as the final reduction result (the value that was written).

```
((!
   (team2 sightings 1674814951967 feedback)
   "This is actually an Andrena (solitary bee)"))))
 "This is actually an Andrena (solitary bee)")
```

The CLI tool assumes that all users are "offline" by default, and that any local changes such as the one Erin just made are not synchronised with the server. You can inspect these "pending" changes as follows:

- Execute action 2. For convenience, this prompt's text already shows that Erin has 1 pending change to their local replica.

Optionally, you can now go back to the main menu to verify that neither the leader nor all other users see the change made by Erin. When you are done, navigate back to the interaction prompt you are in now to interact as Erin.

We will now push Erin's local changes to the leader, and check which other users receive those changes (this corresponds to the paper's Section 3.3 about data projection). Since Erin's change concerns a `feedback` field on one of `team2`'s sightings, the security policy dictates that the change

should only be visible for members of `team2` and not `team1`, i.e., Bob but not Alice. We will push the data, and verify that this is the case.

Step 1: Execute action 5 to push Erin's local change to the leader. For convenience, the output printed above the new interaction prompt already indicates that the leader will only send this change to Bob when they next synchronise with the leader.
Step 2: Execute action 7 to go back to the main menu.
Step 3: For convenience, you can execute action 4 to print any pending changes on the leader which are not yet pushed to clients.
Step 4: Execute action 0 to interact as Alice.
Step 5: Execute action 6 to pull any changes (for Alice) from the leader (there should be 0).
Step 6: Execute action 0 to display Alice's replica data. You can verify that Alice's data about `team2` does not include a `feedback` field.
Step 7: Execute action 7 to go back to the main menu.
Step 8: Execute action 1 to interact as Bob.
Step 9: Execute action 6 to pull any changes (for Bob) from the leader (there should be 1).
Step 10: Execute action 0 to display Bob's replica data. You can verify that Bob's data about `team2` now includes a `feedback` field.

4.3 Running a Malicious ReplicaLang Program

By construction any clients (i.e., users of ReplicaLang) are well-behaved, and cannot execute ReplicaLang programs which contain actions that are forbidden according to the security policy. The CLI tool has an option to circumvent any security checks built into ReplicaLang, which allows users to act maliciously. We will use this option to play out a scenario where user Alice tries to maliciously increase the numbers of points given to their sighting.

⚠ Make sure you are at the interaction prompt's main menu. When proceeding from Section 4.2, go back by executing action 7.

Step 1: Execute action 0 to interact as Alice.
Step 2: Execute action 4 to run a *malicious* ReplicaLang program.
Step 3: Enter and execute the following ReplicaLang program. Due to the special characters and the difficulty of copying text from PDF files, it is best to copy this code from "code example 2" stated in the artifact's "code-examples.txt". The program will evaluate to 11 (the value which was written by "`·!`").

```
(let ((cr (root "my-replica")))
  (let ((sighting (· (· (· cr team1) sightings) 1674813931967)))
    (·! sighting points 11)))
```

Step 4: Execute action 2 to display Alice's local changes that are not yet pushed to the leader. This will show 1 change which would normally not be allowed.
Step 5: Execute action 5 to push this change to the leader. The output above the new interaction prompt will show that this change is "`-Rejected by the leader-`".
Step 6: Execute action 7 to go back to the main menu.
Step 7: Execute action 3 or 4 to inspect the leader's data, or to display the leader's pending changes that are not yet pushed to all users. Both will show that, indeed, the leader has not accepted the change.

4.4 Interacting With Your Own Example

The CLI tool can be used to interact with any (correct) program and corresponding data. The code of the motivating example is completely contained within "`code/motivating-example.rkt`" which can be opened with any text editor. To build your own example, you can modify the motivating example or copy the file content content to a new ".rkt" file in the same directory. Within the file you will find the configuration of the application:

Lines 25–37: The complete data model that is stored on the leader. This code corresponds to the paper's Listing 2.

Lines 44–47: The configuration of all users on the leader, for example:

```
("Alice" user "stored-AliceKey" ((my-team := 'team1)))
```

The values in this list denote the following (from left to right):

(1) The name of the user.
(2) The user's assigned security role.
(3) The user's password to authenticate with the user. In principle it does not matter what is stored here, except that the string must have the prefix "`stored-`".
(4) The user environment (an object). For Alice this is an object with a key `my-team` and a value `'team1`. This key `my-team` can be used in the security policy.

While the motivating example specifies 3 users, in general the CLI adapts to 0 or more users.

Line 54: Specifies all of the different security roles in the application.

Lines 56–65: Specifies the entire security policy stored on the leader. This code corresponds to the paper's Listing 1.

Line 72: Start the CLI program with all of the provided data.

> **i** When using Docker, and you have modified "`motivating-example.rkt`" or you have defined a new file, run the following command to refresh the data in Docker:
>
> ```
> docker build -t srdt .
> ```

5 INSTRUCTIONS: RANDOMISED TESTING

> **i** When proceeding from Section 4 and you are currently running the CLI tool, you may safely break out of program via the keyboard shortcuts Ctrl+C (Windows, Linux) or Command+C (MacOS). You may then exit Racket itself by running the `(exit)` expression, which brings you back to a terminal on the host operating system.

As claimed in the paper's Section 6, we ran 1,000,064 iterations of randomised testing for both read and write tests (7813 tests per program instance, ran 128 times on a 64 core, 128 thread CPU). We include 2 bash scripts to reproduce these tests, namely `read-tests.sh` and `write-tests.sh`. Both have 2 mandatory command-line arguments.

(1) The number of `racket` program instances to start.
(2) The number of iterations to perform per program instance.

Both assume that `racket` is available in the terminal's environment variables (e.g, PATH).

⚠ When running DrRacket natively, the `racket` executable is not added to the terminal's environment variables by default. DrRacket has a menu-option to do so, namely "`Help > Configure Command Line for Racket...`". Sometimes users have also installed a different version of Racket from a different source (e.g., the Homebrew package manager). This can result in errors when trying to run the tests from this section (e.g., "*testing/random-testing.rkt:3:9: collection not found*"). You can check the version that is being used in the terminal by running:

```
racket --version
```

Randomised testing can be started as follows:

Step 1: Open a terminal in the "`code/`" directory of the artifact (where the `.sh` files are).

ℹ When using Docker, to open a new terminal in the Docker `srdt` image, execute the following command in a terminal on the host machine (in the root folder of the artifact, where the artifact's `Dockerfile` is). The working directory is automatically set to "`/code`".

```
docker run --rm -it srdt bash
```

Execute the following terminal command to execute 1 program instance (i.e., for 1 CPU thread) with 10 iterations. This process is expected to stop relatively quickly (e.g., a 3-5 seconds on an ordinary laptop).

```
./read-tests.sh 1 10
```

After completion the terminal will print "`Read tests succeeded`", or an error if a test did not succeed. Write tests can be started in the same way, but using `write-tests.sh`. This process is expected to take slightly longer (e.g., 30 seconds).

```
./write-tests.sh 1 10
```

To reach the 1,000,064 iterations of the paper for both the read and write tests, we ran 128 program instances that each run 7813 iterations on an AMD Ryzen Threadripper 3990X (a 64-core, 128-thread CPU). Note that both will take multiple hours to complete, and that we chose such a high iteration count "just to make sure" due to the nature of randomness. In practice most counterexamples were found by randomised tests after a couple hundred iterations of a single program instance.

```
./read-tests.sh 128 7813
./write-tests.sh 128 7813
```

For thorough testing on a normal laptop or desktop computer, we suggest running a couple thousand iterations spread over the available CPU threads. For example, on a machine with an 8-core, 16-thread CPU (such as our Intel i9-9880H), we tested 10 program instances each with 500 iterations and noted the following run-times:

- `read-tests.sh`: 2 minutes 30 seconds (Docker), 1 minute 50 seconds (Native)
- `write-tests.sh`: 38 minutes (Docker), 30 minutes (Native)

Step 2: A directory "`./logs`" was created to trace the output of each program instance. For the above execution with 1 program instance, this directory contains only a file "`reads-1.txt`" and "`writes-1.txt`". These log files are solely for us to be able to reproduce any counterexamples found by randomised testing.

The paper's Section 6 states that 10 problems were identified using randomised testing. For full transparency we include a (verbose) log of the counterexamples found using randomised testing in the artifact: "`code/bugs-randomised-testing.txt`".

6 CLEANING UP

When using Docker, you may close any Docker sessions that were created during this artifact (via `docker run`), and remove the `srdt` image using Docker Desktop (when installed) or via the command-line:

```
docker image rm srdt
```

You can verify that all of the artifact's Docker containers and images were removed by listing all containers and images on the system:

```
docker container ls
docker image ls
```

When using DrRacket, you may remove it from your system using the DrRacket uninstaller.

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

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

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.