

# Getting Started

We have packaged our artifact in several forms:

- An ISO image for x86\_64/aarch64 systems (**recommended**)
  - **file:** crdt-synthesis-iso-amd64.iso / crdt-synthesis-iso-aarch64.iso
  - No graphical environment, minimal dependencies
    - Launches an SSH server on port 22 (the default port)
    - Comes with a few terminal utilities:
      - htop
      - nano/vim/emacs
    - We recommend instantiating the VM with network access to make it possible to use SSH and copy files out using scp
      - ``ifconfig`` is available in the ISO to help identify the IP address of the VM
- A ZIP of the project sources
  - **file:** sources.zip
  - intended to be used to evaluate the claims in our paper about lines of specification code
  - can be used to evaluate the artifact without a VM, read on for further instructions

When launching a virtual machine, we recommend allocating at least 1 GB for every core made available to our system. The following instructions were tested on a VM with 8 cores and 8 GB of memory (the default VirtualBox configuration). Our synthesis algorithm uses half as many threads as the number of detected cores (to avoid performance degradation due to hyperthreading), so you may be able to allocate more vCPUs since they will not all be used.

**Our team has had trouble getting VirtualBox to function correctly on macOS. We were able to complete all of the evaluation steps on an M1 MacBook Pro by using [UTM](#) (latest version) and launching the ISO image.** Although the graphical environment won't be available, you should be able to perform all the steps by SSH-ing into the system. You may need to run ``chmod +rw path-to-the-iso-file.iso`` to get UTM to boot the ISO.

Across all the VM images, the username is **demo** and password is **demo**. The home directory of all three images directly places you in the base directory of our artifact sources, containing the following files/folders:

- **crdt\_synthesis/**: contains the core CRDT synthesis algorithm the paper introduces

- **crdt\_tests/**: contains the input sequential data types (**.c files**) and the entrypoint (**synthesize\_crdt.py**) for running the benchmarks
- **metalift/**: contains the framework for program synthesis our algorithm is built on top of
  - includes **auto\_grammar.py**, which contains our algorithm for creating grammars based on the input/output types (Section 5.2); this was created specifically for the CRDT synthesis work but contributed as a general utility for future verified lifting work

### **(Optional + not recommended) Evaluating without a VM**

If you would like to evaluate the artifact outside of a VM, you can set up an environment on Linux or macOS by installing some additional dependencies:

First, you will need to install Racket 8.5 (<https://racket-lang.org>) and install the Rosette package:

```
$ raco pkg install rosette
```

Next, you will need to install CVC5 1.0.0 (<https://cvc5.github.io/>), Clang/LLVM 11.1.0 (<https://llvm.org>), and CMake (<https://cmake.org>).

Finally, you will need to install Python 3.8 (<https://www.python.org>), then install Poetry (<https://python-poetry.org>), and finally set up the local environment:

```
$ pip install poetry
$ poetry install
$ poetry shell # places you in a new shell with all Python
dependencies available
```

With these steps, you should have a local environment that can be used to proceed with the artifact evaluation.

### **Building the Software**

Before running the benchmarks there are a few steps needed to build the custom LLVM pass used to analyze input code and to build the input sequential data types:

```
$ ./build_llvm_pass # builds the LLVM pass (sources in llvm-pass)
$ cd crdt_tests
$ ./compile-all # builds all the sequential data types
$ cd .. # get back to the root directory
```

## Simple Artifact Testing

To test that the artifact is functional, you can perform synthesis of the simplest benchmark:

```
$ python -m crdt_tests.synthesize_crdt synth-oplist g_set
```

*Note: it is expected to see several “Failed to synthesize” messages in the log, which are printed as the algorithm searches the space of candidate CRDTs. Eventually, the algorithm will terminate and print out a successfully synthesized CRDT.*

*We have noticed that occasionally the Rosette process may crash with an “unrecognized solver output” error when run in a VM. In this case, we recommend re-running the synthesis algorithm, since this crash appears to be rare.*

This should spin up several threads to execute the synthesis algorithm and in about a minute or less should print out synthesized code for the CRDT and the total time taken for synthesis.

## (Optional) Installing Additional Software

The ISOs boot NixOS, a Linux distribution that makes it easy to produce reproducible environments. However, this does mean that package managers such as apt are not available on the system.

The images come with standard software that we believe should be sufficient to interact with the artifact sources and test the software. If needed, you can install additional software using the “nix-env” command. Your VM must be connected to the internet for the following steps to work.

```
$ sudo nix-channel -update # update the Nix package collection (only
needs to be done once)
```

```
$ nix-env -i vim # see https://search.nixos.org/packages for a full listing of packages
```

## Step by Step Instructions

The evaluation in our paper focuses on the ability for our algorithm to synthesize CRDT designs and the effect of our pruning algorithm on performance. Therefore, the steps to reproduce our experiments are broken into four sections that map to the results presented in Section 7 of our paper:

1. Using the synthesis algorithm end-to-end to synthesize CRDT designs for each of the benchmarks in Table 2
  - a. This focuses primarily on the ability for the algorithm to find a suitable state type and runtime logic, but **will not necessarily produce CRDTs with identical state types to the ones in the paper** since there may be multiple correct designs (for example in the Two-Phase Set benchmark)
  - b. We **do not expect the synthesis time results to be reproducible** since the algorithm is run in a virtual machine with different resources, but we expect the general trends of differences between the default algorithm and the mode with pruning disabled to be visible
2. The ability of the synthesis algorithm to synthesize runtime CRDT logic when provided with a specific state type for the internal structure
  - a. this demonstrates that the specific CRDT designs we claim to have synthesized in Table 2 can indeed be produced by our algorithm
3. The effect of the pruning logic, as seen in Figure 12
  - a. As above, we **do not expect the raw synthesis times to match** because the algorithm is run in a VM, but we expect the distribution plot to show similar trends to the one in the paper
4. The lines of code used to specify each of the benchmarks
  - a. Most of the lines of code will be identical to the ones in Table 1, but due to Python auto-formatting the LoC have reduced for a few benchmarks. We discuss this in more detail in the section describing the evaluation steps.

## End-to-End Synthesis

To reproduce the end-to-end synthesis results, we can use the “synthesize\_crdt” script. **All of the following instructions are meant to be executed from the home folder.**

We include all of the benchmarks used in the paper, undering the following identifiers:

- “grow\_only\_counter”: Grow-Only Counter
- “general\_counter”: General Counter
- “flag\_ew”: Enable-Wins Flag
- “flag\_dw”: Disable-Wins Flag
- “lww\_register”: Last-Writer-Wins Register
- “g\_set”: Grow-Only Set
- “2p\_set”: Two-Phase Set
- “add\_wins\_set”: Add-Wins Set
- “remove\_wins\_set”: Remove-Wins Set

To show the effect of the pruning step in our algorithm, we can run all the benchmarks.

Our benchmarking script has two modes:

- **synth-oplist**: where the pruning algorithm is enabled
- **synth**: where the pruning algorithm is disabled

```
$ python -m crdt_tests.synthesize_crdt synth-oplist all
# writes results to results-all-bounded-pruning.csv

$ python -m crdt_tests.synthesize_crdt synth all
# writes results to results-all-direct-unbounded.csv
```

**These benchmarks take up to an hour with pruning and several (5+) hours without pruning on a machine with 24 cores.** To evaluate our claims in less time and on resource-constrained machines, we recommend evaluating individual benchmarks one-by-one and skipping the most expensive benchmarks (“general\_counter”, “add\_wins\_set”, and “remove\_wins\_set”, especially when pruning is disabled). When we attempted to run the “remove\_wins\_set” benchmark with pruning enabled on a VM with two cores (rather than the recommended 8), it took 2.5 hours to synthesize the CRDT.

To run an individual benchmark, we can use the same script as before. For example, to run just the “2p\_set” benchmark (which takes around a minute to run on a VM with 8 cores), we can run:

```
$ python -m crdt_tests.synthesize_crdt synth-oplist 2p_set
# writes results to results-2p_set-bounded-pruning.csv
```

```
$ python -m crdt_tests.synthesize_crdt synth 2p_set
# writes results to results-2p_set-direct-unbounded.csv
```

*Note: when running benchmarks where the synthesized CRDTs involve the Map lattice type, you may see a “WARNING: USING LARGE BOUND ROSETTE FOR VERIFICATION” message. This is expected, as we explain in the paper that CVC5 does not have a theory of maps so we perform verification with fixed-size data structures in Rosette instead.*

To check the results, we can inspect the “results-...” CSV file. The results file consists of rows for each of the benchmarks with the following columns:

- the identifier of the benchmark
- the time in seconds taken to perform synthesis
- the lattice type that is used for the internal state of the CRDT
- a semicolon-delimited list of the functions (in the Racket language) that have been synthesized:
  - test\_next\_state: the state transition function
  - test\_response: the query function
  - test\_init\_state: the initial state of the CRDT
  - equivalence: the inductive relation used to prove correctness of the CRDT
  - supportedCommand: the invariant used to reason about the history of operations in the unbounded verification

These results can be compared against Table 2 in our paper to verify the **general performance trends** (which benchmarks take longer than others, and the bounded-pruning algorithm generally being faster than the direct-unbounded one). The **raw synthesis times in seconds will likely differ** from those reported in the paper due to resource differences. As explained earlier, there **may be differences in the state type** compared to Table 2 because there may be several correct CRDT designs for each benchmark.

## Reproducing Specific CRDT Designs

Next, we can reproduce the specific CRDT designs in the paper by fixing the state type for each benchmark to the ones listed in the paper and having the synthesis algorithm only search for the appropriate runtime logic. **These benchmarks can take 10s of minutes** (when tested on a virtual machine running on an M1 MacBook Pro, it took 30 minutes). As before, to reduce the time taken to run the benchmarks, you can replace “all” with a specific benchmark identifier to run just those benchmarks.

To enable this synthesis mode, we simply add an additional command line argument “fixed”. To run all benchmarks with the fixed state type, we can run:

```
$ python -m crdt_tests.synthesize_crdt synth-oplist all fixed
# writes results to results-all-bounded-pruning.csv
# (note this is the same file as before, so you may want to back up
the previous results to another file)
```

To check the results at this stage, you will need to again inspect the output CSV file. Each benchmark should have a successfully synthesized CRDT that demonstrates that the designs claimed to have been synthesized in the paper can indeed be produced by our algorithm. The synthesized Racket code in the third column may not be identical to the examples presented in the paper due to non-determinism in the synthesis algorithm, but the implementations can be checked to see if the general logic is similar to the paper examples.

## Checking the Pruning Algorithm

Next, we will generate a figure similar to Figure 12 in the paper, which plots the differences in the distribution of synthesis times with the pruning algorithm enabled or disabled for the Add-Wins Set benchmark. To collect the data for this figure, we have an additional mode that collects the time required to evaluate each of the first N candidate structures for a given benchmark.

In our case, we are interested in the Add-Wins Set benchmark, so we can collect the raw data. We can collect data for the first 20 candidates, which may take a few minutes (7 min on an 8 core VM) so you may want to adjust this number based on the available resources (note that the trend will be less noticeable / disappear with too few candidates because they are enumerated in increasing complexity), with:

```
$ python -m crdt_tests.synthesize_crdt synth-oplist add_wins_set
first_20
# writes results to
benchmarks-add_wins_set-bounded-pruning-first_20-distribution.csv

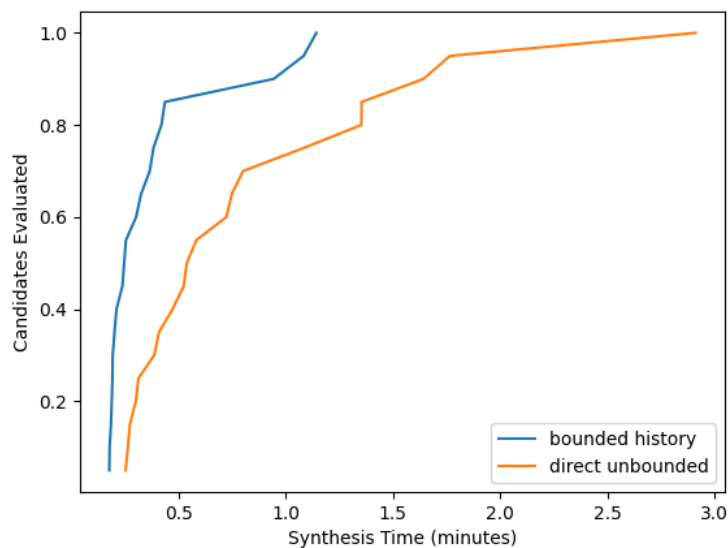
$ python -m crdt_tests.synthesize_crdt synth add_wins_set first_20
# writes results to
```

```
benchmarks-add_wins_set-direct-unbounded-first_20-distribution.csv
```

Then, we can plot a graph similar to Figure 12 using:

```
$ python -m crdt_tests.plot_distribution add_wins_set first_20
```

This will automatically write the plot to “distribution-add\_wins\_set-first\_20.png”. We **do not expect the raw performance numbers to match Figure 12 (which is based on the first 100 candidates)**, but the general trends of the pruning algorithm evaluating more candidates in less time should be visible. For example, this is a plot we produced:



## Checking Lines of Code

Finally, we can verify the claims in the paper that the benchmarks can be specified in a few lines of code. Each specification consists of two components: the C file that describes the sequential data type and the Python code that defines the operation ordering.

To measure the lines of code for each benchmark, we add the lines of C code in the sequential data type and the lines of code for the “inOrder” function specified in “crdt\_tests/synthesize\_crdt.py”. In the Python sources, we ignore lines that simply consist of a closing parentheses since they do not contribute to the complexity of writing the



specification. Note that the exact lines of code **may not exactly match** the values in Table 1 and may be slightly lower due to auto-formatting of the Python sources.

<b>Benchmark</b>	<b>Benchmark ID</b> (in synthesize_crdt.py)	<b>Sequential Data Type</b> (in crdt_tests/)	<b>Expected LoC With Auto-Formatted Sources</b>
Grow-Only Counter	grow_only_counter	sequential2.c	21 (paper: 21)
General Counter	general_counter	sequential2.c	19 (paper: 20)
Enable-Wins Flag	flag_ew	sequential_flag.c	19 (paper: 21)
Disable Wins-Flag	flag_dw	sequential_flag.c	19 (paper: 21)
Last-Writer-Wins Register	lww_register	sequential_register.c	13 (paper: 16)
Grow-Only Set	g_set	sequential1.c	24 (paper: 24)
Two-Phase Set	2p_set	sequential1.c	24 (paper: 24)
Add-Wins Set	add_wins_set	sequential1_clock.c	24 (paper: 24)
Remove-Wins Set	remove_wins_set	sequential1_clock.c	24 (paper: 24)