USENIX Security '25 Artifact Appendix: Recover from Excessive Faults in Partially-Synchronous BFT SMR

Tiantian Gong
Purdue University

Gustavo Franco Camilo
Purdue University

Kartik Nayak
Duke University

Andrew Lewis-Pye
LSE

Aniket Kate
Purdue University / Supra Research

A Artifact Appendix

A.1 Abstract

Byzantine fault-tolerant (BFT) state machine replication (SMR) protocols form the basis of modern blockchains as they maintain a consistent state across all blockchain nodes while tolerating a *bounded number* of Byzantine faults. We analyze BFT SMR in the **excessive fault** setting where the actual number of Byzantine faults surpasses a protocol's tolerance and in partial synchrony where the network experiences a known bounded delay after an unknown global stabilization time (GST). We devise the very first **recovery algorithm** for linearly chained and quorum-based partially synchronous SMR to recover from faulty states (i.e., correct replicas having equivocating states) caused by excessive faults.

We implement and evaluate the recovery procedure for HotStuff, addressed as "recover-HotStuff", in Rust. During implementation, we first test the performance of recover-HotStuff against vanilla HotStuff in the fault-free setting. The throughput resumes to the normal level (without excessive faults) after recovery routines terminate for 7 replicas and is slightly reduced by $\leq 4.3\%$ for 30 replicas. The latency is increased by 12.87% for 7 replicas and 8.85% for 30 replicas on average.

We then evaluate the performance of recover-HotStuff in the excessive fault setting. Each Byzantine replica is simulated with multiple consensus instances where it sends contradicting messages to honest replicas via different instances. We demonstrate that the end-to-end latency and throughput resume to normal level after recovery terminates. We then assess how the average recovery time changes according to different GST. The results indicate that the recovery time is roughly logarithmic in the length of contradicting chains (which is determined by GST).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The execution of the artifact poses no risks to the evaluators' machines, data privacy, or security. The evaluation is conducted within a controlled environment, specifically our own Byzantine Fault-Tolerant (BFT) system. The equivocation is simulated and executed solely on this system, which is isolated from external networks.

A.2.2 How to access

A stable version of the artifact is available at https://github.com/gFrancoCamilo/fault-free-recovery/tree/c9326eb07d347702b898e4c5910a44b3a86986a3 for the fault free setting and at https://github.com/gFrancoCamilo/one-shadow-recovery/tree/781e8b06145bf13c3ef53d54b325dca6c06fcc5b for the one shadow setting. A zip file containing both settings can also be found at https://zenodo.org/records/15133737.

A.2.3 Hardware dependencies

For optimal performance, we estimate that each client or node require at least one core. As a result, a machine with a minimum of 64 cores is recommended to reproduce the results. We also recommend at least 16 GB of RAM, an NVMe SSD, and a minimum of 30 GB to handle the code and the result files.

A.2.4 Software dependencies

Our code requires a Linux-based OS with bash. No specific distribution or version is required. Our experiments were run on Ubuntu 22.04 LTS. To compile and run our artifact on a local machine, the user must install the following softwares:

- Rust and Cargo: https://doc.rust-lang.org/cargo/getting-started/installation.html
- Python 3.6: https://www.python.org/downloads/
- Tmux: https://github.com/tmux/tmux/wiki/Installing
- Clang: https://clang.llvm.org/

In Ubuntu 22.04, the user can install the requirements by running:

```
$ sudo apt update
$ sudo apt-get install -y python3 tmux
clang python-is-python3 curl python3-pip git
$ curl https://sh.rustup.rs -sSf | sh
```

Make sure that `cargo` is in your `$PATH` after installation:

```
$ source $HOME/.cargo/env
```

In case Ubuntu 22.04 is not available, we recommend using Podman or Docker to set up an Ubuntu 22.04 container:

- Podman: https://podman.io/

Once podman is installed, the user should run:

```
$ podman pull docker://ubuntu:22.04
$ podman run -it ubuntu:22.04 /bin/bash
$ apt update
$ apt-get install -y python3 tmux clang
python-is-python3 curl python3-pip git
$ curl https://sh.rustup.rs -sSf | sh
```

Finally, make sure that `cargo` is in your `$PATH` after installation:

```
$ source $HOME/.cargo/env
```

A.2.5 Benchmarks

Our artifact uses require a synthetic workload of transactions to evaluate the performance of the system. Each transaction is a fixed-size byte sequence, with a size specified by the user. Therefore, no external benchmarks or datasets are required for the experiments.

A.3 Set-up

A.3.1 Installation

You can get the code by cloning our git repositories:

```
$ git clone https://github.com/gFrancoCamilo/
one-shadow-recovery.git
$ git clone https://github.com/gFrancoCamilo/
fault-free-recovery.git
```

We also require a few additional Python libraries that assist in plotting graphs and configuring the environment. We provide a `requirements.txt` file containing all necessary dependencies. To install the required Python libraries inside the `benchmark` directory each installed directory, run:

```
$ pip install -r requirements.txt
```

A.3.2 Basic Test

Once all repositories are set up, navigate to the `benchmark` directory within each repository and execute the command `fab localmal`. If all prerequisites are correctly installed, the user should see a message indicating "Running benchmark" along with the duration of the experiment. After the experiment completes, the `logs` directory will contain log files for each client and node. In the fault-free setting, a summary of statistics will also be displayed, providing key metrics such as throughput and latency. This output confirms that the system is functioning correctly

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *In the fault-free setting, our recovery protocol presents a slight decrease in throughput compared to the regular consensus protocol and presents a slight increase in the end-to-end latency.*

(C2): *In the excessive fault setting, our protocol returns the latency and throughput to its normal state after the recovery procedure is done.*

(C3): *The recovery time increases linearly with the logarithm of the number of blocks to retrieve.*

A.4.2 Experiments

Our results are all obtained from Cloud-based experiments on AWS. For the convenience of the artifact evaluation, we provide comprehensive instructions for users to run the experiments locally. The previous claims hold true in both local and cloud-based experiments. The last claim (C3) depends on the real-world network conditions, and the recovery time evaluated from local experiments. For completeness, we have included the instructions for running the experiments on AWS in the wiki of our Github repositories to facilitate replication of our results.

(E1): *[Fault free-setting compared to vanilla Hotstuff] [1 human-minute + 2 compute-hour]: For this experiment, we compare the throughput and latency of Hotstuff with our protocol implementation in a fault-free setting and the regular Hotstuff protocol.*

Preparation: *From the `fault-free-recovery` directory:*

```
$ cd benchmark
$ chmod +x get-results-fig1.sh
```

Execution: *From the `benchmark` directory, run:*

```
$ ./get-results-fig1.sh 10_000 120_000
7 5
```

This script executes our recovery protocol with 7 nodes, starting at an input rate of 10,000 transactions per second and incrementing by 10,000 transactions per second up to a maximum of 120,000 transactions per second. Each input rate is tested 5 times to ensure robust and reliable results.

Additionally, the script clones the vanilla Hotstuff codebase, runs it with the same parameters and copy the results to a `results` directory.

Run the same script for 30 nodes:

```
$ ./get-results-fig1.sh 10_000 120_000
30 5
```

Results: *Change the plot settings in* `fabfile.py` *to* `'nodes': [7, 30]`, `'faults': [0, 1]`, `'max_latency': [9_000]`. *Finally, run:*

```
$ fab plot
```

The graph should be in the `plots` *directory under the name of* `latency.pdf`. *The user will observe a graph similar to* ***Figure 2*** *with our results following the vanilla Hotstuff with a slight decrease in throughput and latency. This supports* ***C1***.

(E2): *[Normalized throughput and latency after recovery] [1 human-minute + 20 compute-minutes]:*

Preparation: *From the* `one-shadow-recovery` directory:

```
$ cd benchmark
```

We created a `setup-env.py` *script to assist in generating the required setup files. The script has multiple options, which can be checked with the* `-h` *flag. To reproduce our settings, run the command:*

```
$ python3 setup-env.py -n 31 -l 7 -c 2
-a 60
```

Then, change the `localmal` *settings in* `fabfile.py` *to* `'nodes': 31` *and* `'duration': 300`.

Make sure that the execution script has the correct permissions:

```
$ chmod +x run-fig3-and-fig4.sh
```

Execution: *From the* `benchmark` *directory, run:*

```
$ ./run-fig3-and-fig4.sh
```

Results: *The results for* ***Figures 3*** *and* ***4*** *can be found in the* `plots` *directory, under the filenames* `tps-recovery-node0.pdf`, `tps-recovery-node1.pdf`, *and* `latency-comparison.pdf`. *In the* `tps-recovery graphs`, *users will observe periodic spikes at fixed intervals. The intervals become shorter towards the end of the graph and after the second recovery procedure. In the latency comparison graph, a significant reduction in latency after the recovery is noticeable. This supports* ***C2***.

(E3): *[Linearity of the recovery time] [1 human-minute + 2 compute-hour]:*

Preparation: *From the* `one-shadow-recovery` directory:

```
$ cd benchmark
```

Execution: *We created a* `run-experiments.py` *script to assist in generating the results. Run the command:*

```
$ python3 run-experiments.py
```

Results: *Finally, run:*

```
$ python3 plot_fig5.py
```

The graph for ***Figure 5*** *should be on the* `plots` *directory with the name* `recovery-time.pdf`. *The graph should be similar to Figure 5 and the user should observe a linearly increasing recovery time. This supports* ***C3***.

A.5 Notes on Reusability

We provide the `setup-env.py` script to enhance the reusability of our artifact, enabling users to customize experiments beyond the paper's scope. This script allows users to configure the number of honest and malicious replicas, set network delays for local testing, and define a custom number of chains for equivocation scenarios. Additionally, as the benchmark vanilla HotStuff codebase, users can test multiple input rates for transactions and adjust transaction sizes to fit their specific use cases. The code is modularized, with easily identifiable components such as consensus, network, and store, making it easy for researchers and developers to modify, extend, or adapt it to their preferred tools and scenarios.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.