ASBJØRN SALHUS, MAGNUS BRANDSEGG
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

University of Stavanger

Fuzz testing a BFT system

Bachelor's Thesis - Computer Science - May 2023

I, **Asbjørn Salhus, Magnus Brandsegg**, declare that this thesis titled, “Fuzz testing a BFT system” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelor’s degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

“There is always one more bug to fix.”

– Ellen Ullman

Abstract

Blockchain technology is becoming more and more prevalent, and it brings with it new security issues. Due to the structure of these distributed systems, manual testing methods are tedious and often miss a lot of edge cases because of the complexity of the systems. The development of automatic testing methods has been proven to be a more effective way of discovering security flaws, bugs, and crashes in distributed systems.

In our thesis, we present an automatic testing tool for the Relab/hotstuff system built with the Google/gofuzz fuzzing framework. The tool is built on the automated unit test generator, Twins. Using the existing scenarios that Twins create, we replace one of the messages that are being sent through the network with a randomly created fuzz message. The tool iterates through the messages with new fuzzed input to trigger a panic and to find out where the program crashes.

With the use of our tool, we have discovered 6 locations in the Relab/hotstuff system where the system crashes. Our tool presents relevant information as to where the crash happened, a full stack trace, and the fuzzed message that caused the crash; making the debugging process easier for the maintainers/bugfixers.

Acknowledgements

We would like to express our deepest appreciation to our supervisor, Leander Nikolaus Jehl, for his guidance, support, and invaluable feedback throughout the course of this thesis. His expertise and insight have been instrumental in shaping the direction and focus of this work, and we are grateful for the time and effort he has dedicated to our research.

We would also like to extend our sincere thanks to the Department of Computer Science at the University of Stavanger for providing us with the resources and facilities necessary to conduct this research. The department's commitment to excellence in research and education has been an inspiration to us, and we are proud to have been a part of this community.

We would be remiss in not mentioning our friends and family, who have provided us with unwavering support and encouragement throughout this journey. Their love, kindness, and understanding have been a constant source of strength and motivation, and we are deeply grateful for their presence in our life.

Finally, we would like to thank all the participants who took part in our research, as well as anyone who provided feedback, encouragement, or assistance along the way. Your contributions have been invaluable to this work, and we are humbled by your generosity and willingness to help.

Thank you all for your support, encouragement, and dedication to this project. We are honored and grateful to have had the opportunity to undertake this research.

Contents

Chapter 1

Introduction

Chapter 1 presents a short introduction of our thesis. It covers the background and motivation of the technologies and programs used, our objectives with this implementation, a short summary of the approach as well as the outline of the rest of the thesis.

1.1 Motivation

Blockchain technology has revolutionized the way we perceive and handle digital transactions. With its decentralized architecture and robust security features, it has gained widespread adoption in various domains, from finance [13] to healthcare [1] [11]. However, the increasing complexity of blockchain systems, coupled with the ever-growing number of potential attack vectors, poses significant challenges to their security and reliability. This might include malware such as ransomware and viruses, or new forks and updates that introduce new vulnerabilities to the system. The increasing complexity also makes the margin for human error bigger as the systems become harder to maintain.

Fuzz testing, a technique used to identify and eliminate software defects by generating random or semi-random inputs, has emerged as a promising approach to improve the quality and security of blockchain systems [5] [2].

Our thesis seeks to investigate the effectiveness of fuzz testing in identifying vulnerabilities in blockchain systems. While traditional testing techniques may be adequate for simple programs, blockchain systems are more complex, making them difficult to test exhaustively. By applying fuzz testing to blockchain systems,

we can simulate a wide range of potential inputs and edge cases, potentially uncovering previously unknown vulnerabilities.

The importance of this research lies in the need for reliable and secure blockchain systems. As blockchain technology continues to gain adoption, the consequences of a security breach or software defect could be significant, leading to loss of assets, reputation damage, and legal liability (examples: [4] [10]). By identifying and addressing vulnerabilities through fuzz testing, we can improve the resilience of blockchain systems and enhance their trustworthiness.

In this paper, we present a fuzzing tool designed to discover when a blockchain system panics and causes a crash. Our tool is implemented on the Relab/Hotstuff blockchain system [12] and uses the automated unit test generator, Twins [3]. Through fuzzing and deliberate altering of messages being sent through the Twins network, we are able to discover new bugs and flaws.

Overall, this thesis aims to explore the benefits of fuzz testing in improving the security and reliability of the Relab/Hotstuff blockchain system. By examining the current state of the art in blockchain testing, identifying potential vulnerabilities, and evaluating the effectiveness of fuzz testing, we can contribute to the development of more secure and robust blockchain systems.

1.2 Background

One approach to achieving greater security and resilience in blockchain systems is through the use of Byzantine fault tolerance (BFT) consensus algorithms. BFT is a class of algorithms that are designed to enable a distributed system to continue functioning correctly even when some of the nodes in the system fail or behave maliciously. BFT algorithms are particularly important for blockchain systems because they are designed to ensure that the system can continue to operate correctly even if some of the nodes in the system are compromised. The term ”Byzantine fault” refers to a type of failure mode in a distributed system, where a node in the system may behave in an arbitrary or malicious manner. This type of failure can occur due to a variety of factors, including hardware failures, software bugs, or most threateningly, deliberate attacks on the system. The way BTF systems are built the most potent attack is one from within; one with the correct format that appears to follow protocol. Finding out where the system misbehaves, should it be exposed to an inside attack, is, therefore, an important part of improving

the overall security of distributed systems. Finding these vulnerabilities effectively involves emulating malicious behavior. An automated unit test generator of Byzantine attacks, Twins [3], does exactly that. Twins emulates byzantine attacks within a system by making a copy of a node with the same identity. Twins then sends both of the nodes through the system. The system can not distinguish between the two almost identical nodes and a single functioning node and this makes twins able to generate an emulation of some interesting byzantine attacks. Among these attacks are **Equivocation**; where a Byzantine node sends different messages to different recipients, **Amnesia**; where after a node has voted for a proposal, it forgets it has voted and votes again, or by **losing internal states**, especially locks. Twins is a central part of our thesis since we run our fuzz testing through these twins-scenarios.

1.3 Objectives

The goals of our study are:

- To investigate the current state of the relab/hotstuff system and its components
- To analyze the existing scenario-test framework for the relab/hotstuff system and assess its effectiveness in detecting vulnerabilities.
- To develop a method for recording and fuzzing the messages in the relab/hotstuff system's scenario-tests using an existing fuzz testing framework.
- To apply the developed methodology to the relab/hotstuff system and identify potential security vulnerabilities
- To evaluate the effectiveness of the developed methodology in detecting previously unknown vulnerabilities

1.4 Approach and Contributions

Our approach to solving the problem of finding bugs in the hotstuff system starts with selecting the proper fuzzing framework for the job. Once we have a fuzzing

framework that works well with our data, we implement it into the existing scenario-tests of the Twins system. This consists of taking the old tests run by Twins and rewriting them so that they deal with the randomly generated values from the fuzzer. These random values are then converted from proto messages to the messages the system uses. These new, fuzzer-generated messages are swapped with one of the original messages in a Twins-scenario every time the scenario is run. The fuzzed messages are designed to invoke and locate panics within the system and present potential vulnerabilities and bugs.

1.5 Outline

In our thesis, we use our acquired knowledge about BFT systems, The Twins testing system as well as an external fuzz framework to create a fuzzing tool on the Relab/Hotstuff blockchain.

Chapter 2 reviews relevant work in the field of tools for testing distributed systems.

In Chapter 3 we go over our approach to solving the main task for this thesis; namely creating and implementing the tool. The Chapter starts with the design of the tool and then moves on to how it's implemented, how errors are handled, and the fuzzing involved.

The results of our tool are presented in Chapter 4. Using snippets of code, profiling and benchmarking outputs, and statistics and graphs we present our findings using the tool on the hotstuff system.

In Chapter 5 we discuss the approach we chose, how adaptable it is to changes in the hotstuff system as well as potential improvements that could be made.

Our thesis concludes in Chapter 6 with a summary of all our work and results.

Chapter 2

Related Work

In this chapter, we review the existing literature related to fuzz testing and consensus algorithms in blockchain systems.

2.1 Tyr

Tyr: Finding Consensus Failure Bugs in Blockchain Systems with Behaviour Divergent Model

The Tyr system [5] proposed by Chen et al. (2021) is a novel approach to testing consensus algorithms in blockchain systems. The system uses a behavior divergent model to detect consensus failure bugs, which are bugs that can cause nodes in the network to diverge from the consensus.

Tyr works by simulating different behaviors of the nodes in the network and comparing their outputs to identify any divergences. The system was evaluated on six different commercial consensus systems and was shown to be effective in detecting consensus failure bugs that were not uncovered by existing testing methods. Tyr was also compared to other state-of-the-art testing tools (Peach, Fluffy, and Twins), and proved to cover more branches and perform better overall.

While our thesis has a lot in common with Tyr in the field of blockchain vulnerability discovery, Tyr focuses on finding consensus failure bugs and logical errors. Our thesis tries to find where the program panics and what caused the panic to discover potential security flaws.

Overall, the work by Chen et al. (2021) provides a valuable contribution to the field of consensus algorithm testing in blockchain systems and improving

blockchain security.

2.2 LOKI

LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols

LOKI is a state-aware fuzzing framework designed by Ma et al. (2023) that uses advanced state-aware fuzzing to discover bugs. Traditional fuzzing techniques use random inputs to test the system, which can miss certain edge cases that may cause bugs. In contrast, LOKI is state-aware, which means that it uses a combination of existing blockchain states and random inputs to test the system. This allows LOKI to explore a wider range of scenarios and increases the likelihood of finding bugs or vulnerabilities.

The LOKI framework has been tested on several popular blockchain consensus systems such as GoEthereum, Meta Diem, IBM Fabric, and WeBank FISCO-BCO. The results showed that LOKI was able to find previously unknown bugs and vulnerabilities in these protocols.

While our thesis focuses on a fuzz testing tool for the HotStuff system, LOKI is a general-purpose fuzzing framework for blockchain consensus protocols. LOKI's state-aware approach makes it effective in identifying complex bugs that might be missed by other fuzzing tools. In contrast, our tool is specifically designed for the HotStuff system and may not be as effective in identifying bugs in other consensus protocols.

Similarly, as with Tyr, LOKI also focuses on finding logical errors and bugs in the consensus algorithm instead of panics and crashes.

2.3 Automated Vulnerability Discovery in Distributed Systems

Automated Vulnerability Discovery in Distributed Systems[2] is a paper by Banabic et al. (2011). The paper proposes a vulnerability discovery technique for distributed systems. The technique generates malicious entities in a distributed system and sees what impact the entities have on the system's behavior.

In the author's own words, "One can think of this approach as "fuzzing" at the

level of system nodes — akin to input fuzzing, but at a higher level of abstraction." [2] The paper also presents the automated vulnerability discovery platform AVD, which is an implementation of the technique.

The approach of this technique is used to find bottlenecks and bugs in distributed systems.

Overall, the paper presents a valuable tool in the field of automated testing of distributed systems. The research presented by Banabic et al. has provided us with insight into other fuzz-like approaches and has been a valuable resource for us in our own thesis.

Chapter 3

Approach

Chapter 3 explains the problems, solutions, and decisions made during the development period. As well as what the tool does on the surface and in-depth.

3.1 Existing Approaches/Baselines

Hotstuff's main existing testing tool is the twins testing tool. Twins uses different scenarios, a scenario consists of a set amount of nodes, a set of partitions, and a set amount of views. The tool goes through a scenario and chooses one node as the twin node. The twin nodes appear as a single node but do not communicate with each other, making the node seem defective as it sends messages twice and forgets some messages. Hotstuff also has convert functions that convert the protocol buffer structures to and from hotstuffs internal structures. Fuzz testing is supported by the go programming language and also has some additional APIs that make fuzz testing easier to implement.

3.2 Testing Hotstuff

In order to test Hotstuff we need something that takes input. Most of the functions in the Hotstuff implementation take inputs, but many of those functions aren't required to handle random inputs because they are only called from other functions, where the inputs are checked beforehand to be valid. What we need is a situation where an invalid input may occur in the Hotstuff implementation. Twins is a testing tool built for testing the Hotstuff implementation and has a sce-

nario where multiple network nodes are simulated. We use the Twins scenario to test Hotstuff, but we need to be able to include input in the testing somehow. The scenario in use sends a constant amount of messages between nodes and our fuzz tool will replace one of the messages in that scenario. A message is a set of values defining an action or event that a node wants to send to another. One example of what a message is used for is to make a proposal, the proposal message is meant to be sent to all other nodes. A counter is used to determine which message to replace, only one of the messages will be swapped, where the counter is a specific number. The function that receives the random message input will propagate it to other functions and any invalid input that is not handled correctly will cause a crash.

3.3 Fuzzing

Another requirement of a fuzz tester is a random value, which will be used as the input. Golang structs have private and public (exported) fields, and the message structures of the Hotstuff package contain many private fields, since we are not in the Hotstuff package we cannot access these fields and cannot fuzz them directly. One solution is to use the public functions in the Hotstuff package, for example, the block constructor 'NewBlock()' with random parameters to create a fuzzed block variable. Another solution is to use other parts of the Hotstuff project that construct Hotstuff structures to create random messages. The Hotstuffpb package uses protocol buffers[8] to serialize and deserialize the Hostsuff structures so they can be sent over the internet using gRPC[9]. We could send the Hotstuffpb a set of random bytes and ask it to deserialize it into a structure, but that is not necessary because the structures in Hotstuffpb use public (exported) fields and can be accessed directly. Hotstuffpb has functions that convert a Hotstuffpb structure to a Hotstuff structure, these will be used and also be included in the test since we are giving it random inputs.

3.4 Fuzzing Framework

Fuzz testing is an already established method of testing, and there are multiple frameworks and libraries that make creating fuzz tests easier and have a lot of

useful functionality. Each fuzz testing framework has its own strengths and weaknesses, some are more flexible, and others have many useful features. We have considered three different fuzzing frameworks that support golang for our fuzzing tool. Golang has a built-in fuzzing framework [6], google has a framework called gofuzz [7] and there is a framework by Dvyukov on GitHub called go-fuzz [14]. The built-in tool lets you fuzz based on an initial state of inputs with a small set of supported types. The supported types are bool, string, []byte, and int, unit, and float types of various sizes. When an error or panic is encountered, the framework creates a file with the fuzz input parameters used when the program crashes. You can rerun a previously generated fuzz test by giving the test tool a specific seed. The Google fuzzing framework supports a lot more types and has a specific chance to set a pointer to nil which can be configured at runtime. You can set your own fuzzing rules and constraints using the framework. Dvyukov's tool is straightforward, with a function called Fuzz that takes in a parameter of type []byte you can just use the parameter as fuzzing values. The framework handles panics and errors for you.

We have chosen to use the fuzzing framework gofuzz made by Google because of the nil chance and the better support for structs. Gofuzz gives us more freedom in terms of writing log files because the framework doesn't do it for us.

3.5 Error Handling

When the Hotstuff implementation fails we want to catch the error and collect meaningful information about the error. If we stopped at the first error then a different error might show up for each test, which is unintuitive and annoying if you want to fix a bug and you are looking for a specific error, that is why the test continues to run even if the tests crash. A lot of the crashes will be from the same line of code and have the same error message as a previous crash. Having a way to filter out duplicate errors is important to avoid having to output a ton of information at once. Grouping together errors by the error message and location and only outputting one error from each group of errors ensures that a lot of duplicate information gets filtered out. The crash with the smallest fuzz value string is used because it gives the smallest and simplest output. Having a sorted list of unique errors ensures the error info is as predictable as it can be, the order the error list is sorted in is not important as long as it is consistent between runs.

A stack trace after the panic occurred will reveal a lot of debug information, and the stack trace line right below the panic routine part is extra useful because it describes the exact line that triggered the panic. Conversions and scenarios are different and are recovered in two different places, where the panic gets recovered is output alongside all the other information.

3.6 Design

Our tool is an automatic unit test designed to find potential vulnerabilities and bugs in the Relab/Hotststuff system using fuzz testing. The test can be run at any time, takes a couple of seconds to complete, and gives useful information about some of the errors that occurred during testing. The tool is designed to catch and locate crashes also known as panics, but is not built to find other errors that result in unexpected incorrect results. Random values are used to test different parts of the Hotstuff implementation. The random values are used in normal tests or procedures in the program to try to trigger a crash, these will be called fuzz values. In a real network, the nodes would send what we call messages to each other by serializing them using protocol buffers. The data structures used by protobuf are different from the data structures used internally by Hotstuff, and the Hotstuff implementation has functions that convert them. By creating random protobuf data as fuzz values we convert them as part of the fuzz test. The output of the conversion is used in the other part of the test. The Twins framework has a scenario test with a simulated network that runs and tests for incorrect results. A slightly modified version of the Twins scenario allows us to test the scenario with the converted fuzz values, and also test for crashes. This routine runs for multiple iterations, if an error occurs during the conversion or scenario test the program will recover and skip to the next iteration. The errors encountered are summarized and output with relevant information like the error message, stack trace, and the fuzz values which caused the crash for each relevant crash.

3.7 Implementation

We start by initializing the error info structure that will be used to collect all the relevant errors. The error info structure keeps one error for every error that

crashes in different places in the program. The structure also counts the number of errors that occur, one counter for the messages that crash during message conversion, and one for the scenarios that crash during the scenario test. The fuzzing itself runs for multiple iterations and each time a random fuzz message is made based on Hotstuff's protocol buffer structures.

The gofuzz framework uses a random number generator along with the Golang reflect package to fill entire structures with random values automatically. It is also configured to have a 10% chance to set any pointer to nil. Gofuzz does not support assigning a random type, so it crashes when it encounters an interface, and it cannot choose a random message type. Gofuzz has support for custom functions based on specific types, so you can create a function for the problematic interfaces to fix them. There are two custom functions, one is for choosing a random message type, and the other is for choosing a random signature type. There are four different types of messages, ProposeMsg, VoteMsg, NewViewMsg, and TimeoutMsg. There are two different signature types which are BLS12Sig and ECDASigs.

With a fuzz message with random values based on the protocol buffer structures, we can convert the fuzz message to a Hotstuff message based on the message types defined manually in the Hotstuff project. There are already functions made to convert the proto messages to Hotstuff messages, The conversion functions do not include the NewViewMsg because the protobuffers don't have that type, but a NewViewMsg is just a SyncInfo with an ID, and there is a conversion function for SyncInfo. All of the protocol buffer message types are missing an id, so a random number is chosen, the random number was created in the fuzz message creation part, and is assigned to the message after it has been converted.

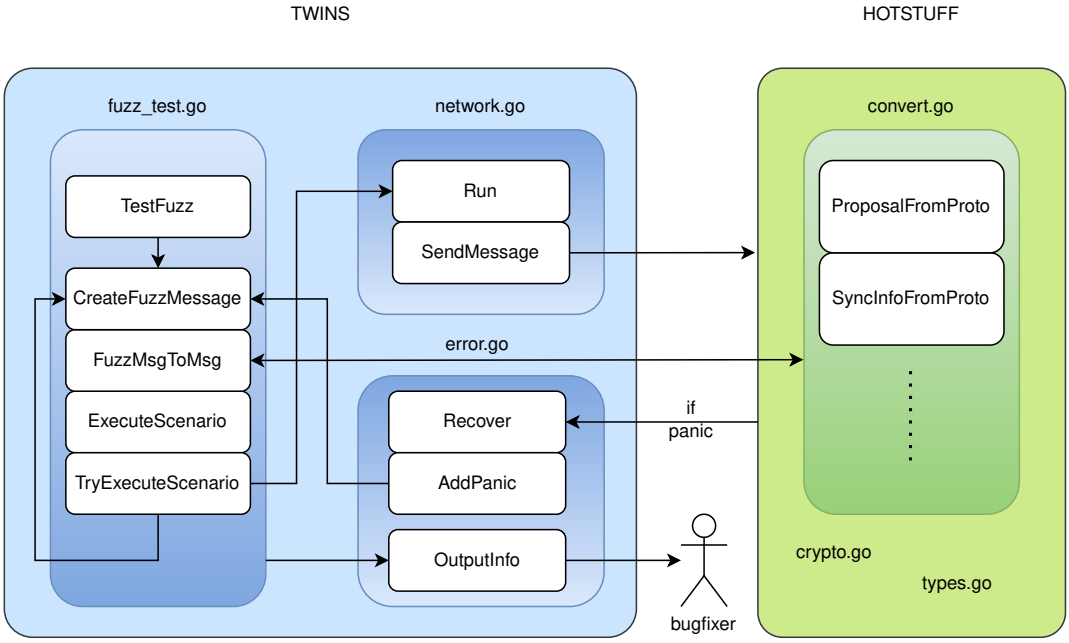
Calling functions outside the fuzz tester with random values may result in crashes, these crashes are potential vulnerabilities to the stability of the program and it is the job of a testing tool to find these errors. Using Golang's `recover` function we can keep running the program, as well as gather information about the crash. A stack trace, the error text, and the fuzz message itself is combined into a structure which is then saved to the error info structure using the hash map. The crash line and file, as well as the panic message and recovery location, are combined to create the key. If there already is an error info with the same key, then both the info's fuzz message sizes get compared and the error info with the shortest fuzz message will be kept.

If the fuzz message got converted successfully, then a Twins scenario without a twin node will be run. The scenario structure has a network structure where the random Hotstuff message will be included, and the first message that is sent will be swapped with the random message from the network. The rest of the scenario runs more or less normally unless the program crashes. The same type of recovery and error info collection as last time is used here.

After all the iterations are done, all there is to do is to output useful information and save the results. For every unique error, the error message, location, and recover location are displayed along with the full stack trace and the fuzz message. The fuzz message gets made into a string that looks similar to how you would declare a structure in golang. The number of times the program crashes, as well as how many were because of the conversion and how many for the scenario are also displayed. Protocol buffers have a built-in marshal and unmarshal system that converts a proto message to and from a byte slice, which allows us to write it to a file. The byte slices from each fuzz message get converted to base64 with newlines between and get saved to a single file. When loading the fuzz messages from a file the base64 gets split by a newline, converted back into a byte slice, and then into a fuzz message. In addition to saving the fuzz messages themselves to a file, we also save the random number generator seed used to create the fuzz message as an alternative.

Figure 3.1 shows roughly how the test runs. It starts in the `TestFuzz` function and calls `CreateFuzzMessage`, `FuzzMsgToMsg` and `TryExecuteScenario` gets called multiple times until it ends with the `OutputInfo` function. `CreateFuzzMessage` creates the fuzz message and `FuzzMsgToMsg` converts it using Hotstuff's own convert implementation, logging any crash that might occur. `TryExecuteScenario` runs the scenario and the scenario sends the fuzz message to the rest of the Hotstuff implementation, where any crash goes to `recover` and gets saved by `AddPanic`. When enough iterations have run `OutputInfo` outputs the saved info.

Figure 3.1: Diagram of the design



Chapter 4

Results

Our fuzz tester gives a lot of info and may be unreliable, and may also take a long time to complete. Increasing the number of iterations for one test increases reliability, as well as the time taken before it completes. Our experimentation explores how different configurations affect reliability, performance, and usability.

4.1 Output From Run

Running the fuzzer for a thousand different fuzz messages reveals six different errors found in different parts of the hotstuff source code. The fuzzing tool outputs the error message of each different error, as well as a stack trace and the message structure used when the error occurred, and the line and file where the program crashes. After outputting info about each individual error, it says it found six different errors, that 388 of the fuzzed values failed to convert to messages, and that 8790 of the 18360 scenarios that ran crashed.

Table 4.1: Error list

Error nr	**Error type**	**Error location**
item 1	invalid memory address or nil pointer dereference	crypto/crypto.go line 128
item 2	invalid memory address or nil pointer dereference	hotstuffpb/hotstuff.pb.go line 612
item 3	invalid memory address or nil pointer dereference	crypto/cache.go line 92
item 4	invalid memory address or nil pointer dereference	hotstuffpb/hotstuff.pb.go line 605
item 5	invalid memory address or nil pointer dereference	types.go line 144
item 6	invalid memory address or nil pointer dereference	crypto/crypto.go line 112

Table 4.1 shows a list of all the errors found as well as the location where the program crashes. All of the errors had the error message ”`invalid memory address or nil pointer dereference`”.

4.2 Benchmarking

```
go test -benchmem -run=^$ -bench ^BenchmarkFuzz$ -benchtime 1000x -count 5

goos: linux
goarch: amd64
pkg: github.com/relab/hotstuff/fuzz
cpu: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx
1000    3.458626722s    3458644 ns/op    276450 B/op    3979 allocs/op
1000    3.304062133s    3304078 ns/op    260070 B/op    3735 allocs/op
1000    3.767412629s    3767429 ns/op    290373 B/op    4221 allocs/op
1000    3.458368587s    3458384 ns/op    267844 B/op    3849 allocs/op
1000    3.489068640s    3489086 ns/op    275929 B/op    3983 allocs/op
```

Figure 4.1: benchmark results

Running the fuzz test normally with 1000 iterations takes about three and a half seconds. Figure 4.1 shows the execution time varies between 3.3 and 3.8 seconds, with allocations varying from 3735 to 4221 per iteration. The execution time of each fuzz test varies because of the random nature of the fuzz tester. Iterations that crash early in the tests will speed up the execution time, the scenario tests only run after successful conversion attempts, so for every conversion test that crashes there is one less scenario test and the program takes a little bit less time. The execution time is expected to increase once most of the bugs have been fixed, as fewer crashes will occur meaning that more iterations run until the end.

4.3 Fixing An Error

Figure 4.2: function with error

```
106 // VerifyQuorumCert verifies a quorum certificate.
107 func (c crypto) VerifyQuorumCert(qc hotstuff.QuorumCert) bool {
108     // genesis QC is always valid.
109     if qc.BlockHash() == hotstuff.GetGenesis().Hash() {
110         return true
111     }
112 >   if qc.Signature().Participants().Len() < c.configuration.QuorumSize() {
113         return false
114     }
115     block, ok := c.blockChain.Get(qc.BlockHash())
116     if !ok {
117         return false
118     }
119     return c.Verify(qc.Signature(), block.ToBytes())
120 }
```

Figure 4.3: function with fixed error

```
106 // VerifyQuorumCert verifies a quorum certificate.
107 func (c crypto) VerifyQuorumCert(qc hotstuff.QuorumCert) bool {
108     // genesis QC is always valid.
109     if qc.BlockHash() == hotstuff.GetGenesis().Hash() {
110         return true
111     }
112 +   if qc.Signature() == nil {
113 +       return false
114 +   }
115     if qc.Signature().Participants().Len() < c.configuration.QuorumSize() {
116         return false
117     }
118     block, ok := c.blockChain.Get(qc.BlockHash())
119     if !ok {
120         return false
121     }
122     return c.Verify(qc.Signature(), block.ToBytes())
123 }
```

One of the errors outputted was in crypto/crypto.go on line 112 which is shown in the figure 4.2. The error message tells us that a `nil pointer dereference` occurred, and the line in which the error occurred contains a lot of dereferences. By breaking up the line into smaller lines we find out that 'qc.Signature()' is `nil` every time the program crashes. By checking for `nil` and returning early we avoid the `nil pointer dereference`. The function that crashes verifies a quorum certificate, we decided to return `false` if the signature was `nil`, the changes are shown in figure 4.3. After fixing one of the errors and running the fuzzing test again we find that it reports five different errors this time. Fixing the other errors eventually results in new errors showing up in the tests that weren't there before.

4.4 Probability

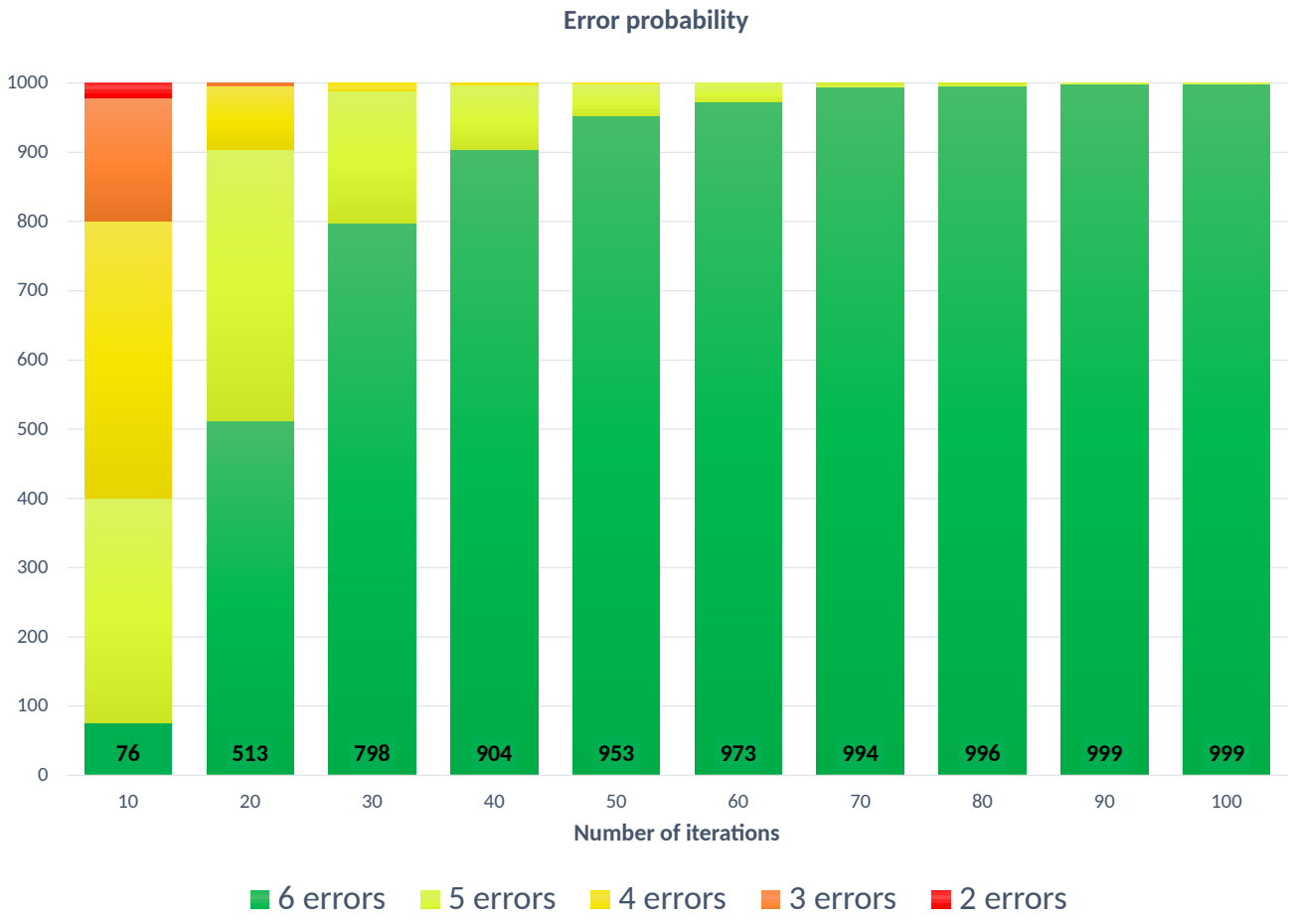


Figure 4.4: Bar graph showing the probability of finding all the errors

During the program evaluation, we're interested in seeing how many iterations we have to run to find all the errors reliably. Running the program 1000 times, each with a different number of iterations ranging from 10-100, provides us with the data shown in Figure 4.4. Looking at Figure 4.4 we can see that running the program with 10 iterations fails to find all the errors in the majority of the runs and only finds all the errors 76 out of 1000 times, or 7.6% of the time. The results

quickly improve with more iterations, and the program finds all errors in over 99% of cases with 70 or more iterations. Running the program with 100 iterations finds all errors in 99.9% of cases and only misses one error about 1-2 times per 1000.

With the information on how often all the errors are found based on the number of iterations, it's also interesting to see what errors are found more often, and which ones are usually missed. Looping through the program and mapping the error info to a hashmap with the location of the panic as the key, gives us information about which of the errors show up more often. We've put this information in a table.

Table 4.2: Error frequency

Error Location	**Recovery Location**	**Frequency**
hotstuff.pb.go:605	convert	1592
hotstuff.pb.go:612	convert	1606
types.go:144	convert	1471
cache.go:92	scenario	962
crypto.go:128	scenario	654
crypto.go:112	scenario	1231

The table 4.2 shows how often the different errors show up when running the program 10000 times. The errors that get recovered from the convert part of the program, occur more often. This is because the convert part of the program happens before the scenario functions.

4.5 Loading From File

After running a normal fuzz test in which many fuzz messages are created, only one of each unique error gets saved. The fuzz messages get saved in two different ways, through the seed that was used to create them, and the binary representation of the fuzz message itself. Running a normal fuzz test results in six different unique errors, and six corresponding fuzz messages get saved to a file. Loading the random source values of each fuzz message as an int64 works as expected,

with each fuzz message being identical and crashing the exact same way as before. Loading the fuzz messages from their binary representation on the other hand gave different results from the original test. In the replicated test there were only four unique errors, the two missing errors were from 'hotstuff.pb.go'. Through experimentation, we concluded that the protocol buffers were the cause of the changes, each call to `proto.Marshal()` gave a different binary. This is probably because of the unpredictability of hash maps and because of how `nil` pointers are handled, sometimes a pointer to a structure with only `nil` will be converted to a pointer to `nil` and vice versa. Feeding random input to a function is likely to produce unexpected results, but we will keep the extra functionality since at least nothing crashed. Both of the missing errors are from the file 'hotstuff.pb.go', which is a file automatically created by the protocol buffers themselves. There is a chance that none of the protocol buffer functions, including grpc, produce a message that triggers these crashes, which may mean that the crashes never would happen in a real consensus because the messages always get created from grpc functions. In order to avoid these errors in the standard fuzz test also, the standard fuzz tester would have to be modified to serialize and deserialize the fuzz message after it is created.

Chapter 5

Discussion

5.1 Adaptability and Changes

Our fuzzing tool was made specifically to test hotstuff, but it was testing its robustness as a bft system. Other systems, protocol implementations, or frameworks that could benefit from a tester that focuses on crashes caused by random inputs would benefit from this fuzz tester. The fuzzing tool needs to be changed if big changes are made to the current hotstuff implementation, or if it would be used on a different distributed system. There are custom functions given to the gofuzz framework in order for it to not crash, this is caused by the `oneof` keyword in protocol buffers which creates an interface. There is an extra function that randomly chooses a message type, this function would have to be expanded or changed if more message types were to be introduced. The part of the program that creates a readable string from a fuzz message is made manually and is completely dependent on the structure of the protocol buffer message types and would have to be changed for almost every change to the protocol buffers. This could be improved by using the same techniques that gofuzz uses, namely reflection, where a smaller set of additional context or custom functions are needed. Even though Google's gofuzz already uses reflections, it doesn't uses the specialized protocol buffer reflections, which has support for `oneof`. With protocol buffer's reflection library both the fuzzing and the readable string generation could be improved. Using the string function provided by protocol buffers is also a valid option but is not as readable as it does not have indentation or line breaks. The saved fuzz messages should be taken with a grain of salt after changing some of the code as the fuzz

messages made from the saved binaries or seeds will be different than the original if the structures have changed. Running a normal fuzz test is recommended to ensure that no new errors have popped up, and to ensure that there aren't other messages that still trigger the same error. Old stack traces are invalid once the code has been changed because lines will be moved around once new lines are added, and any changes to the file system will not be accounted for.

5.2 Potential Improvements

The output of the fuzzing tool holds back a lot of information because of the number of errors it encounters, and a lot of them are duplicates. Grouping crashes and only showing each error group once is necessary but showing only the info from one panic for each group is not. The fuzz messages can be compared to each other to find similarities and differences, and the fuzz tool can find which of the values needs to be nil or a certain value to crash. Additional probability information can be given based on how often the crash occurs, and how many of each fuzz message type crashed that way. Seeing as the scenario tests are being used for the fuzz testing, but not being tested itself, an alternative can be used. Sending a message and running it as an event without the use of a scenario would make the test faster and there would be no dependencies on the twins test.

Chapter 6

Conclusions

We successfully created a testing tool for the Hotstuff implementation, and the test found panics in six different parts of the source code. The six different crashes get shown as a list to the maintainers with all kinds of relevant information about each crash, with a summary at the end. The maintainer may rerun the previous test by using the automatic saving functionality that we created. After a benchmark, we conclude that the test takes on average 3.5 seconds to complete when running for 1000 iterations. We also present relevant information as to where all the panics happen and how reliably our tool finds them on each run. We hope that our testing tool will help improve the security and robustness of the hotstuff implementation when it is used in a real-life situation, and not just in a scenario simulation.

Appendix A

Instructions to Compile and Run System

Write your Appendix content here.

```
    cd twins
go test -v -run TestFuzz
```

Bibliography

[1] Asaph Azaria et al. “MedRec: Using Blockchain for Medical Data Access and Permission Management”. In: (2016). DOI: 10.1109/OBD.2016.11.

[2] Radu Banabic, George Candea, and Rachid Guerraoui. “Automated Vulnerability Discovery in Distributed Systems”. In: (2011), pp. 188–193. DOI: doi.org/10.1109/DSNW.2011.5958811.

[3] Shehar Bano et al. “Twins: BFT Systems Made Robust”. In: (2022).

[4] Ryan Browne and MacKenzie Sigalos. *CNCB: Hackers have stolen 1.4 billion this year using crypto bridges*. URL: https://www.cnbc.com/2022/08/10/hackers-have-stolen-1point4-billion-this-year-using-crypto-bridges.html. (accessed: 05.04.2023).

[5] Yuanliang Chen et al. *Tyr: Finding Consensus Failure Bugs in Blockchain System with Behaviour Divergent Model*. DOI: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00068. (accessed: 16.02.2023).

[6] Google. *Golang built-in fuzz tool*. URL: https://go.dev/security/fuzz/. (accessed: 09.02.2023).

[7] Google. *Google/gofuzz*. URL: https://github.com/google/gofuzz. (accessed: 09.02.2023).

[8] Google. *Google/protocol-buffers*. URL: https://protobuf.dev/. (accessed: 12.05.2023).

[9] Google. *gRPC*. URL: https://grpc.io/. (accessed: 14.05.2023).

[10] Sandali Handagama. *Coinbase: Crypto Exchange Coinbase Faces Class Action Lawsuit Over Alleged Lapses in Security*. URL: https://www.coindesk.com/policy/2022/08/23/coinbase-faces-class-action-lawsuit-over-alleged-lapses-in-security/. (accessed: 05.04.2023).

[11] Amit Juneja and Michael Marefat. “Leveraging blockchain for retraining deep learning architecture in patient-specific arrhythmia classification”. In: (2018). DOI: 10.1109/BHI.2018.8333451.

[12] Hein Meling and Leander Jehl et al. *Relab/Hotstuff*. URL: https://github.com/relab/hotstuff. (accessed: 13.01.2023).

[13] Jayanth Rama Varma. “Blockchain in Finance”. In: (2019). DOI: 10.1177/0256090919839897.

[14] Dmitry Vyukov. *dvyukov/go-fuzz*. URL: https://github.com/dvyukov/go-fuzz. (accessed: 09.02.2023).

University of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: Asbjørn Salhus