Improving rippled: Leveraging passive model inference techniques to test large decentralized systems

Sengim Karayalçin[1] , **Stefanie Roos**[1] , **Satwik Prabhu Kumble**[1]
[1]TU Delft
s.karayalcin@student.tudelft.nl, {s.roos, s.prabhukumble}@tudelft.nl

Abstract

Blockchains and cryptocurrencies, like Ripple, are becoming more widely used. Testing the large decentralized systems these technologies on is complex, as the behavior of the system is dependent on many external factors. We will examine the viability of using passive model inference techniques to test the systems based on the network traffic they produce. Passive inference techniques have been used extensively to model and test different types of systems. However, it is unclear how well passive model inference techniques work for inferring models of large decentralized systems based on the network traffic that these systems produce. Here we show that detecting bugs in the implementations of decentralized protocols is possible. These results were achieved by simulating a version of the Ripple network and modeling the workings of a node in this network. We also simulated the network with defective nodes and by observing the different models generated, were able to detect these bugs. Our results suggest that using passive model inference techniques on network traffic can help test large decentralized systems.

1 Introduction

Blockchains and cryptocurrencies have become more popular over the past years. While theoretical solutions to a lot of the problems that arise when deploying these technologies exist, the practical implementations of these solutions are often lacking [5]. The ripple network, which is a network implementing the XRP Ledger Consensus Protocol(XRP LCP) consists of nodes that maintain a ledger to provide a fast global payment system. There are a number of requirements for this network to function properly. First of all the network should not be able to fork, which is when two different sets of nodes continue working on different ledgers. In the second place, the network needs to be able to handle a number of faulty nodes. A node is considered faulty when it either cannot function properly due to technical reasons, or is acting maliciously. The third, and final condition is that it should be impossible for the network to stop making forward progress. These requirements are full-filled by the XRP LCP, assuming some conditions hold[7]. For the maintenance of a trustworthy and safe environment for payments, it is critical to guarantee the correct workings of this protocol. Because the communication between nodes is an essential part of the functioning of the network, analyzing this communication can provide valuable insights and help detect possible bugs.

Analyzing these communications will be done by using model inference techniques. Model inference is based on observing the behavior of a system, and analyzing it's in- and outputs to build a finite-state-machine modeling its behavior. Different ways to accomplish the modeling of state-machines have been developed [16]. Model inference has been used to model bank cards [1], but also to model larger, communication-based systems like bot networks [8]. These techniques have as yet not been used to model large decentralized systems.

This leads to our main research question: "Can passive model inference techniques be used to test large decentralized systems based on the network traffic these systems produce?" Answering this question will show how effective the tools for generating models are for decentralized networks, and provide a methodology for testing these systems by analyzing their network traffic.

The research question will be answered by first selecting a proper state-machine model for the workings of a validator node in the ripple network. Subsequently, a theoretical description of the XRP LCP will be given in the form of the previously selected state machine model. A version of the ripple network will then be simulated with correct and bugged implementations of the consensus algorithm. Finally, using passive inference techniques, models will be generated for the bugged and correct implementations, and these models will be analyzed to see if the bugs can be detected.

Our results show that the use of passive model inferencing techniques is a viable way of creating abstract models of systems implementing protocols like the XRP LCP. Additionally, we were able to detect small bugs in simulated versions of the network. Consequently, we conclude that using passive model inference techniques can be used to test large decentralized systems.

In section 2 some more background on the inference of state-machines will be provided. In section 3 our methodology will be provided. In section 4 the theoretical model will be built. In section 5 an overview of the experimental setup

and the results will be provided. In section 6 the reproducibility and ethical implications of this paper will be discussed. In section 7 the results of this experiment will be analyzed and discussed. Finally, section 8 will be the conclusion.

2 Background

2.1 Deterministic Finite Automata

Deterministic finite automata (DFA) are finite state machines, that either accept or reject a given input string. These types of finite state machines are widely used for creating formal models of computers.

A DFA is a tuple $\mathcal{M} = \langle Q, \Sigma, q_0, F, \delta \rangle$

- Q is the finite set of possible states and $q_0 \in Q$ is the initial state
- Σ is the input alphabet
- $\delta : \Sigma \times Q \rightarrow Q$ is the transition function mapping an input symbol and a state to a new state
- $F \subset Q$ is the set of accepting states.

The workings of a DFA are fairly simple. Σ^* is an input string of arbitrary length. This string is then processed a symbol at a time by the DFA. When the string ends the DFA is in a state $q \in Q$. If this state is contained in the set of accepting states the string is accepted, and if the state is not the string is rejected.

A small example of a DFA is a machine that acts on the alphabet $\Sigma = \{0, 1\}$. The machine will accept any string where the final symbol is a 1. Examples of accepted string are $0000001, 1 \in \Sigma^*$. Examples of rejected strings are $1110, 0 \in \Sigma^*$.

2.2 Model inference

Model inference is a technique which is used to infer the workings of a state machine based on it's observed behavior. Broadly speaking, there are two approaches that accomplish this goal. Passive inference techniques [14], and active inference techniques [12].

In this paper, passive inference techniques will be used. Passive model inference is based on having a sample of traces that describe the execution of a system, and then using inference algorithms on them. These traces could be log files generated by a software system, or network traces captured during a program's execution.

A number of different algorithms and tools exist that use different approaches for passive model inference. Flexfringe [17] is a tool that uses the dfasat algorithm [11]. Dfasat is the algorithm that won the STAMINA competition [18]. An evaluation of the use of Flexfringe to model bank cards has been done by Wieman et. al 19. This research evaluates the use of Flexfringe on large production systems using the logs of the system. Wieman et. al conclude that Flexfringe can be used to model large scale systems, but that it is not a straightforward task. When the input alphabets of the automata that Flexfringe is modeling get large, the resulting graphs can become hard to interpret. Additionally, Wieman et. al conclude that the use of Flexfringe for large systems requires a good amount of knowledge on modeling algorithms. Flexfringe has, as far as we can find, not been used to model large, communication-based systems using only traces of the network traffic.

Other tools, like Synoptic 4 and InvariMint [2], can generate models directly from log files, only needing a set of regular expressions that describe the log format. In the active learning setting the tool LearnLib [15] implements a version of the L^* algorithm 3.

3 Methodology

To answer the research questions we first need to develop a theoretical model that clearly describes the workings of the XRP LCP. Creating this theoretical model establishes a benchmark for comparing the empirical models. Additionally, the abstractions of the concrete messages developed in the theoretical model will help when creating an appropriate empirical model. Creating this theoretical model will be done by analyzing the protocols defined by Chase [7].

Abstractions of the messages that are sent in these protocols will be created and a DFA modeling the behavior of a validator node in the XRP LCP will be built. The abstractions of messages are an integral part of the eventual model, and the abstractions are also a major part of generating input strings for our learner from the network traces [8]. Automatic methods to extract the format and semantics of a communication protocol exist [6; 9], but since the XRP LCP has a limited set of abstract messages, these are not used.

When the theoretical model has been built, data for learning an empirical model needs to be acquired. To collect this data, a simulated environment will be built as observing network traffic of a node in the live ripple network is difficult. Using simulation has additional benefits. It allows for a more controlled network environment. For example: In the live ripple network, network failures may occur occasionally, but capturing one of these events is unlikely and could require a large amount of data collection. In a simulated environment, we can easily simulate different types of failures, which will lead to a more complete model.

Once the data has been collected, it needs to be transformed into a format that can be used as an input for Flexfringe. The standard input format for Flexfringe is format used in the Abbadingo competition [13]. Transforming the data into this format requires clear labeling of the messages that are sent. Additionally, it may be required to model messages in a less general way than we will in the theoretical model. This new labeling will more closely reflect the workings of a rippled node. This relabeling is required as the general model will lead to a large input alphabet which can make the models generated by Flexfringe hard to understand. A more limited input alphabet can help improve the readability of the models. Limiting the alphabet will also not cause us to lose any information, as it only fails to describe sequences of messages that would not be sent in the ripple network due to the implementation of the XRP LCP.

When the final input file for Flexfringe has been generated, a model needs to be built from this. Flexfringe has a lot of different configuration options that can help generate an appropriate model. We will try various configuration options to

see which results in the best model. We will then compare the differences between the theoretical- and the empirical model and reflect on the viability of our methods in a larger network.

We will also generate models for bugged implementations of the XRP LCP. The bugged and correct models will then be compared. It will be examined whether the models of the bugged implementations can help detect the bugs. We will choose bugs which are fairly simple mistakes that could be made when implementing the consensus protocol. These bugs will also be chosen such that they should be detectable using our method. More subtle bugs which do not influence the output behaviour of a node should not be detectable and will therefore not be tested.

4 Theoretical model

To build a theoretical model for the workings of the XRP LCP we need to develop proper abstractions of the messages exchanged between nodes. To do this we will first provide a brief overview of the XRP LCP in section 4.1. Then we will provide abstract representations of the messages and the states. Finally, we will develop a theoretical model.

4.1 Brief overview of XRP LCP

The XRP LCP has 3 primary protocols: The deliberation protocol, the validation component, and the preferred branch protocol. In one round of the deliberation protocol, all nodes propose transactions. If a transaction appears in enough proposals of trusted nodes this transaction will also be proposed by a node itself in the next round of the deliberation protocol. When the deliberation phase finishes a node broadcasts a validated ledger. In the validation protocol, a node checks how many validation messages they have received for a ledger. If this amount is above a certain number, the node will change its last fully validated ledger. Finally, the preferred branch protocol is used to decide between branches when there are several working branches.

4.2 Message types

For the Deliberation phase, we have two types of messages: Propositions and Validations. To build proper abstractions of these message types, we need some abstract types for denoting the internals of the XRP LCP more efficiently:

- τ is a transaction. We model transactions here as a tuple $\tau = \langle validity, identifier \rangle$. $validity$ is a boolean value denoting the validity of the transaction and identifier is a unique identifier for the transaction. We can ignore the specifics of a transaction as it does not really matter for our model what the exact state of the ledger is. All we need to know is whether a transaction is valid and whether it is different from another transaction. The difference is established by comparing the unique identifier.
- T is a list of transactions.
- L is a ledger. We represent it as a tuple here: $L = \langle identifier, s, ancestor \rangle$ where the $identifier$ is a unique identifier, s is a sequence number , and $ancestor$ is the previous ledger.As mentioned above the specific state of the ledger is not important for the model we will be constructing. We, again, really only need to be able to tell two different ledgers apart. The identifier is used for this. We keep the sequence number and the ancestor in as they could help in constructing the model.
- N is a node. We model a node as a tuple $N = \langle UNL, L, identifier \rangle$ where UNL is a list of the identifiers of trusted nodes, L is the identifier for the last fully validated ledger, and the identifier is a unique identifier.

In the XRP LCP there are only two types of messages that can be sent, a proposition message, proposing a new set of transactions, and a validation message, asserting the intent to validate a ledger. The proposition message is sent every round of deliberation and the validation message is sent after the deliberation phase is finished. For constructing our model we use the following abstractions of these message types:

- $P_{T,r,L,i}$, this is a proposition message. T is the list of proposed transactions, r is the round number of the deliberation phase the sending node is in, L is the previous ledger the sending node was working with, and i is the unique identifier of the sending node.
- $V_{L,i}$, this is a validation message. Here L is the ledger that will be verified, and i is the identifier of the sending node.

4.3 Construction of the Input Alphabet

Before we create our final model we need to construct an input alphabet. To construct this input alphabet appropriate groupings of some messages need to be created. As the preferred branch protocol used in the XRP LCP is very complex to model in a DFA, we will abstract the validation messages a node receives to one message representing the output of the preferred branch protocol. As for the proposition messages, we create a symbol that closely resembles the actual message.

Validation messages

For validation messages, the grouping is fairly simple. Only the validation messages that were received last from another trusted node is stored and used for anything other than fully validating a ledger. We ignore the full validation of a ledger for now as there is no output associated with it. This means we can model our validation messages as a list of the last validations sent by every trusted node. Additionally, the only thing these last received validations are used for is the preferred branch phase of the protocol. Because of this, from the list of last received validations, and knowing the current working branch of the node, we can derive the outcome of the preferred branch protocol. Therefore we can model our list of validation messages as a single abstract validation message with a Ledger:

- $PrefBranch(L)$ this message is an abstraction of a list of validation messages. The ledger L denotes the message that would be the outcome of the preferred branch protocol based on this list.

Proposition Messages

For the proposition messages, all of the necessary information is given by an integer representing the ledger the proposal is

based on, an integer representing a unique transaction set, and an identifier for the node that sent the message. The round of the deliberation process is not taken into account as the only thing that is important for the network to reach consensus is that a number of nodes propose the same transaction set. Therefore we model every proposition message as follows:

- $P_{T,L,i}$, this is a proposition message. T is the list of proposed transactions, L is the ledger the transaction set is based on, and i is the unique identifier of the sending node.

4.4 Model

The DFA model, $M = \langle Q, \sigma, \delta, q_0, F \rangle$ that will be constructed in this section is based on a validator node with a UNL of $n \in \mathbf{N}_{>0}$ nodes. The quorom for validation is the percentage of receive transaction sets that need to be equal before consensus is reached. $quorum * n = a$ is the amount of nodes that need to send the same proposition message before consensus is reached.

An overview of the workings to the machine M is that the machine will model the workings of a node during one phase of deliberation. Accepting an input string means to validate a new ledger based on the transactions that were received. The working ledger, the ledger the node is proposing upon, of the machine $L' = 0$ as it is the first ledger that is encountered. This means that a preferred ledger message that does not have 0 as the preferred ledger, $PrefBranch(L), L \neq 0$ always results in a rejected input string. The way the machine handles proposition messages is that it stores the last received proposition message that it received from every other node in its UNL. Any state where a or more nodes sent the same transaction-set is a accepting state. Any state where this is not the case is not.

To model the set of states Q, first we define a special reject state $R \in Q$. This is the state that results from a preferred branch message that does not equal 0. This state is a sink state where no possible message can result in the state changing. All other states are defined as $Q' \subset Q$. Every state $q \in Q'$ is defined as $q = \{T_1, T_2, ..., T_n\}$, the set of the proposed transaction sets from every other node. There is a special transaction set $T = -1$ for when some node has not sent a proposition message yet. The start state $q_0 \in Q$ is the state where no propositions have been received yet, $q_0 = \{-1, -1, ..., -1\}$.

The definition of the transition function δ is fairly straightforward:

- $R, s \mapsto R$, this means that, when the state is in the reject state R, any input message $s \in \Sigma$, results in the machine staying in state R.
- $q, PrefBranch(L) \mapsto R$, this means that a preferred branch message, with a preferred ledger $L \neq 0$, brings the machine into the reject state R.
- $q, PrefBranch(0) \mapsto q$, this means that a preferred branch message, with preferred ledger $L = 0$, does not change the machine's state.
- $T_1, T_2, ..., T_i, .., T_n, P_{T',0,i} \mapsto T_1, T_2, ..., T'_i, .., T_n$, this means that a proposition message, received from node i, proposing transaction set T', will change the i'th entry describing the state to T'.

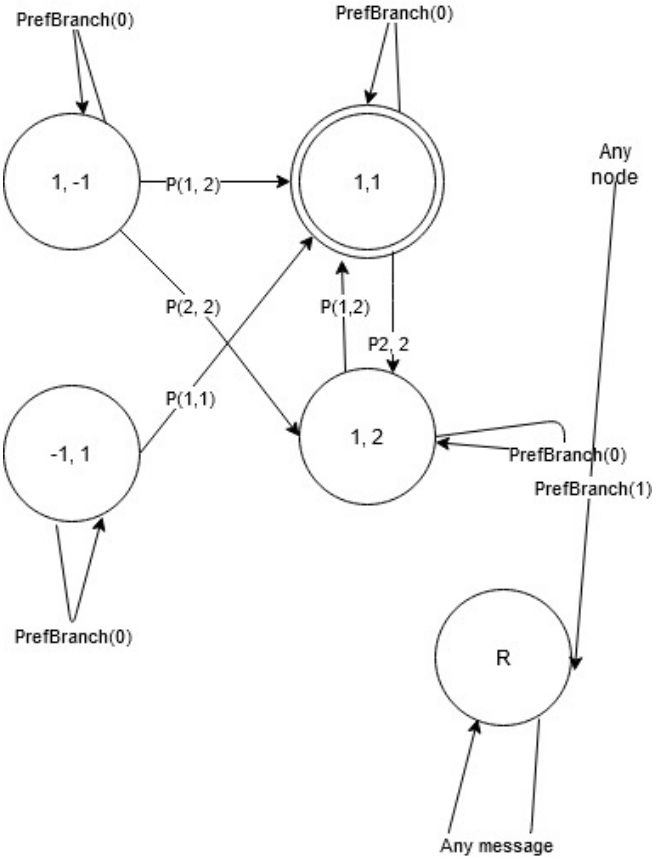

Figure 1: A zoomed in version of the model for 2 nodes. This ignores any transitions going to states not in the figure. Any node is represented with a tuple a, b where a and b are the last received transaction sets from node 1 and 2 respectively. The $PrefBranch(0)$ always results in a self loop, and the $PrefBranch(1)$ always results in the reject state R. Self loops resulting from propositions messages were left out.

Defining the set $F \subset Q$ of accepting states is also fairly straightforward. $F = \{q \in Q | \exists T_i \neq -1 \in q, |\{T_j \in q : T_j = T_i\}| \geq a\}$. This means that there exists one proposed transaction set that has reached a or more propositions.

Finally it needs to be shown that the set Q is finite as otherwise the DFA is not a finite automation. We will show that Q is finite $\iff$ n is finite, and the number of rounds is finite.

Let Q be the set of states and let $n, r \in \mathbf{N_{>0}}$. The set of different transactions that can be sent is T and the order of this set $\#T = n * r + 1$ (+1 resulting from the 'empty' transaction). As n and r are finite $\#T$ is also finite. Now the states $q \in Q$ are defined as a Cartesian product of n T's, therefore the order of $\#Q \leq T^n + 1$ (+1 resulting from the reject state). As $\#T$ is finite $\#Q$ is also finite.

The number of possible states is not necessarily equal to $T^n + 1$ as there are states that could be unreachable. If, in our model, the choice is made that the first transaction set received is always 1, a lot of transitions and states can be eliminated, as only states like $q = \{-1, -1, ..., -1, 1, -1, ... - 1\}$ are reachable from the start state. Additionally, if we chose

to label the node from which the first message is received as node 1 only $q = \{1, -1, -1, ...-1\}$. Other similar reductions should be possible , resulting in a much less large set of states and transitions from every state, but for the sake of completeness and simplicity, all states and transitions were left in our model.

A small example is provided in Figure 1, where the amount of nodes is set to 2, the amount of rounds to 1, and the accepting quorum a is equal to 2.

5 Experimental Setup and Results

In this section, the setup for the experiment will be discussed. First, we will discuss the setup for the simulations. Then we will discuss the data format that will be used for Flexfringe and how we derive the data from the simulation. Then we will discuss the models that resulted from Flexfringe and how these were achieved.

5.1 Simulation Setup

To simulate a small test version of the rippled network, we implemented the algorithms used in the consensus protocols as provided by Chase [7]. Then 4 of these nodes were set up to communicate through an environment set up to deliver messages after all nodes have had time to process the previous set of messages. Every node then logs the messages it receives and these strings of messages are subsequently classified based on whether the round of the consensus algorithm results in a new closed ledger.

The nodes were set up in a fairly simple manner. A python class was created to represent a node. This class implemented all the algorithms required for the XRP LCP. Whenever the consensus process started on a new ledger the node randomly generated a number of transactions. The transactions were represented as integers between 0 and 20. After generating these transactions the node broadcasts them across the network by sending them to an environment class that managed all nodes in the network. After every node sequentially executes their update routine and broadcasts new proposition or validation messages, the environment makes all nodes receive the messages that were sent.

A benefit of this is that all messages arrive in a timely manner. This is also slightly problematic for the generation of the model. When all messages arrive, and no nodes are acting maliciously, the consensus protocol will always result in a new closed ledger. Additionally, with enough overlap between UNL's all nodes will also always have the same new closed ledger. This means that the $PrefferedBranch(L)$ message, with a ledger that is not the current working ledger of the node, will never be received. To make sure these shortcomings do not result in an incomplete model some basic scenarios were added to the simulation. These scenarios make sure that the node we are testing experiences every scenario.

We also implemented different bugged nodes. The first bugged implementation we used was a version of the consensus protocol where consensus is reached when 3 nodes send the same proposition message. The second bugged implementation we considered was a version where consensus is always reached after 3 rounds no matter what messages are sent in that round.

5.2 Data generation

To be able to learn accurate and readable models from data generated by our simulation setup a small alphabet setup needs to be used. When larger input alphabets are used the models generated by FlexFringe become very hard to read. Using the same input alphabet which was used for the theoretical model would result in an input alphabet of 16*4 different symbols for the proposition messages. To combat the problem of having a very large input alphabet, we make some assumptions about the timing of messages that a node can receive. These assumptions are based on the setup used in the specification of the XRP LCP and are mainly based on the fact that a node will receive messages from a single round of the consensus algorithm sequentially. This means that no messages from later, or earlier rounds are received during a round of propositions. When we place these limitations on our input model we can limit our input alphabet to only having 4 symbols to represent different proposition messages. If we use an additional message to indicate a new round of propositions starting the strings will have a fairly straightforward form, represented here as a regular expression:

PB[0-1](P0 P[0-1] P[0-2] P[0-3] RR)+

Here the first PB[0-1] represents a PrefferedBranch message. This is either PB0 if the prefferedbranch is equal to the working ledger of the node, or 1 if it is not. The second part of the string is the sequence of proposition messages. The first proposition message is always P0. The second message is P1 if the proposed transaction set is different from the one received earlier or P0 if it is the same. For every message the number after the P represents whether the proposed transaction set was proposed before or not. If it was the number will be the same as for the proposition message where it was proposed before. If the transaction set has not been proposed then the number will be an integer number higher that every number that was used in this round. The RR message does not represent an actual message, but it represents a new round of propositions starting. This resets the numbering for unique proposals. The strings of the form P0 P[0-1] P[0-2] P[0-3] RR are repeated at most 4 times as the protocol only uses 4 rounds.

An example accepted string would be:

PB0 P0 P1 P2 P3 RR P0 P0 P0 P0 RR

This example string represents first receiving a Preferred branch message for the current working ledger. Then the first round of the consensus protocol, 4 different proposals are received. In the second round every node proposed the same transaction set, consensus was reached and a new ledger was closed.

An example rejected string would be:

PB0 P0 P1 P2 P3 RR P0 P1 P2 P3 RR
P0 P1 P2 P3 RR P0 P1 P2 P3 RR

In this string the first round is the same as above. After the first round of proposals every node keeps proposing a different set of transactions. This means consensus is never reached, and therefore the string is rejected.

5.3 Results

After generating a number of different strings using the simulation setup described above, we tried creating models using some of Flexfringe default configurations. Some of these were inappropriate for our data, and as such did not result in useful models. The batch-overlap.ini configuration resulted in a fairly clear model of a validator node. The model can be seen in Figure 2.

An overview of the model can be given most clearly through walking through the way the model handles some of the common input strings. First we will look at the way the model handles a string where consensus is reached after two rounds:

PB0 P0 P1 P2 P3 RR P0 P0 P0 P0 RR

The first input symbol PB0 jumps the machine to the start_consensus state. From there the strings P0 P1 P2 P3 RR lead it back to the start_consensus state. After that, the sequence of symbols P0 P0 P0 P0 RR leads the machine into the accept_state. As this state is an accepting state and all of the input has been processed the string is accepted.

The example above illustrates the workings of the machine. There are two clear paths: the path where all proposals are equal, and the path where there are different proposals. Unless 4 of the same proposals are received the RR message will send the machine back to the start_consensus state. As the input string will end on the RR message when consensus is reached the machine will end in the accept_state when 4 of the same proposals are received.

In figure 3 and figure 4 the models of the bugged implementations can be seen. In these models, the erroneous paths to the consensus_reached state are marked red, and the correct paths are marked green. The first bugged implementation in figure 3 will result in an accepted state even when one P1 message is received during a round. The path this results in is represented in red. The second bugged implementation will always result in an accepting state after 4 rounds. This can be seen in the model as every path leads to the consensus reached state. This means that the model will only result in a rejecting state when a PB1 message is received.

As can be seen, the model generated by Flexfringe is a fairly accurate representation of the workings of a validator node. Most common input will be classified appropriately by the model. No clear bugs or unexpected paths can be seen in this model. This was to be expected as the simulated node is programmed to work as specified by the algorithms in the paper [7]. The models generated by bugged implementations are different and the erroneous paths are fairly simple to spot.

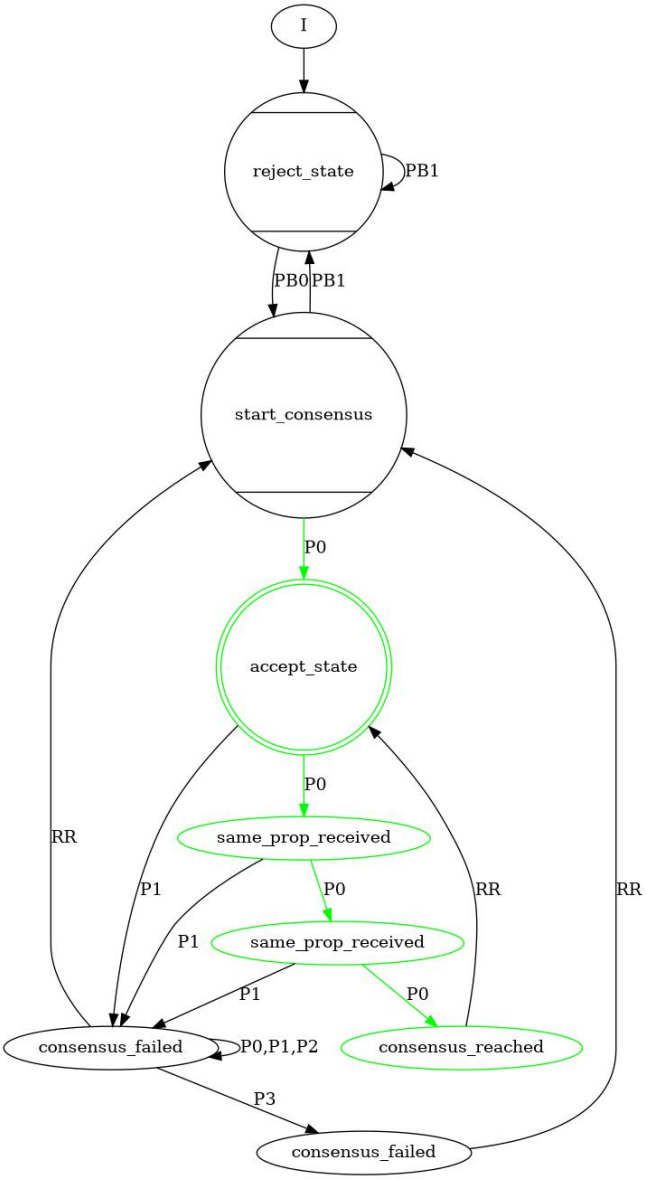

Figure 2: Model of normal node.

6 Responsible Research

There are not many ethical considerations that are associated with this research. The methodologies described could theoretically be used to aid the reverse engineering of software systems while only having access to the network traffic these systems produce. The reverse-engineered model could then be used to find vulnerabilities the system may have. However, reverse engineering the system using the methods we describe in this paper requires a large amount of knowledge

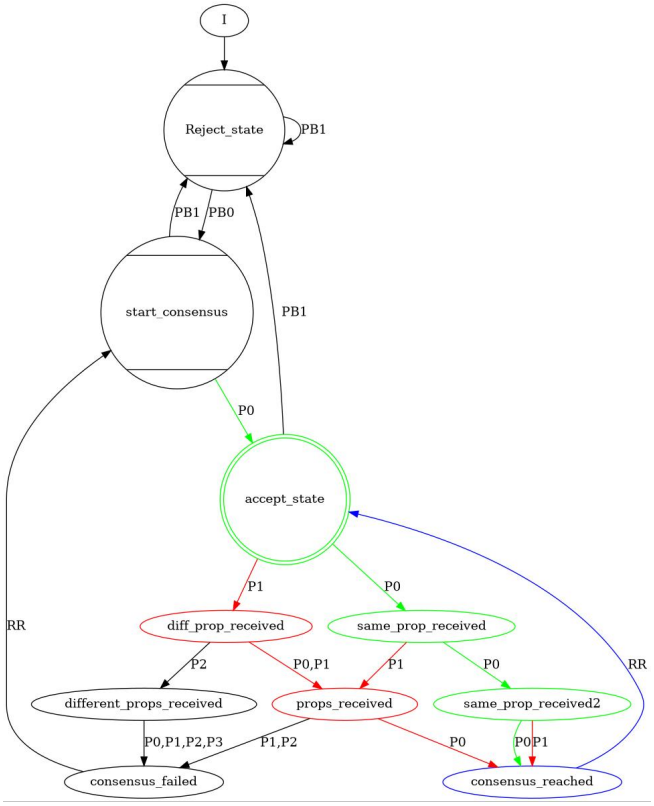

Figure 3: Model of the first bugged node.

Figure 4: Model of the second bugged node.

on how the system operates. If an attacker already has this information there are numerous ways attacks could be executed. Additionally enabling engineers working on the system to use the methods we describe in this paper to fix bugs and patch security vulnerabilities should result in more secure systems rather than less secure systems.

The methodology used in this paper should be fairly simple to reproduce. As only simulated data was used, the code (and seeds) used to generate every data-set used and the data itself can be shared to test the results. Additionally, Flexfringe is free to download and, as such, should not provide any barrier for a researcher wanting to reproduce our work. In general, none of the work done in this paper should be hard to reproduce.

7 Discussion

The results that we found are fairly common in literature. Models of large, communication-based systems have been created using active learning techniques [8]. Additionally passive learning techniques have been used to model several different types of systems [10] [19]. However, passive learning techniques have not been used extensively to model large, decentralized, communication-based protocols based only on network traffic. Our results show that passive learning techniques can be used to create models of decentralized systems effectively. Additionally, we have shown the effectiveness of using these same techniques to find bugs or other unintended behaviors.

The results were found by simulating nodes of the XRP LCP, and introducing bugs into the implementations of these nodes. By comparing the models of the correct and incorrect implementations, we were able to detect the mistakes. Whilst finding implementation mistakes in this way was successful, we knew what bugs we were looking for, as they were introduced artificially. If we did not know what bugs to look for, it might have been significantly more difficult to locate, and eliminate incorrect paths in the models.

Furthermore, some notes should be made on the simulations. Whilst the simulated network traffic is fairly realistic, as it is modeled after the traffic we expect to see from a rippled node. Constructing the abstract versions of the messages that were used as an input for Flexfringe will be more complex, when messages are not specifically constructed to accommodate the abstractions. Furthermore, large scale implementations of the XRP LCP use additional messages to establish connections. To create a model similar to the models we created would require ignoring messages that do not map to messages in our theoretical model. However, ignoring messages that might influence the behavior of the system could impact the accuracy of the model that is generated. Finally, the simulated environments allowed us to make the node that we were testing experience a varied amount of behaviors. In a live environment, failures to close a new ledger will not happen as frequently. Therefore collecting data to generate a complete model may take a significant amount of time, and some scenarios may never occur under normal circumstances, resulting in models that do not display the full behavior of a system.

8 Conclusions and Future Work

The main question we sought to answer is: "Can passive model inferencing techniques be used on network traffic to find bugs or unintended behaviour in large decentralized protocols, like the XRP LCP?" To answer this question we first created a theoretical model of a validator node in the XRP LCP. We then used this model to build a simulation of a network of nodes in the XRP LCP and used the simulated network traffic between these nodes to create empirical models using Flexfringe. After building a model of a correct implementation we introduced several different bugs into our implementation, and created new models of these bugged nodes. When we compared these different models, we were able to observe differences in the models of the correct and incorrect implementations of the XRP LCP. Using these differences we were able to detect the bugs in the implementations. Therefore we conclude that passive model inference techniques can be used to detect bugs in large decentralized applications based on network traffic.

As we used a simulated environment, it is still an open question how well the methods presented in this paper will work in practice. Future research could focus on applying the techniques presented on live network traffic from a node in the rippled network. An additional avenue for future research is to utilize active model inferencing techniques to create models of the XRP LCP.

To conclude, our research provides methods to create models of decentralized applications using only the network traffic. The models generate by these methods could be used to detect bugs in the applications. However, it is unclear how viable these methods will be when trying to model practical applications, and more research is required to review practical applications of our method.

References

[1] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal models of bank cards for free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468. IEEE, 2013.

[2] Jenny Abrahamson. Invarimint: Modeling logged behavior with invariant dfas. 2012.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[4] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277, 2011.

[5] Vitalik Buterin. Hard problems in cryptocurrency: Five years later, 2019.

[6] Juan Caballero and Dawn Song. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Computer Networks*, 57(2):451–474, 2013.

[7] Brad Chase and Ethan MacBrough. Analysis of the xrp ledger consensus protocol. *arXiv preprint arXiv:1802.07242*, 2018.

[8] Chia Yuan Cho, Domagoj Babi ć, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 426–439, 2010.

[9] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14, 2007.

[10] Christian Hammerschmidt, Samuel Marchal, Radu State, Gaetano Pellegrino, and Sicco Verwer. Efficient learning of communication profiles from ip flow records. In *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, pages 559–562. IEEE, 2016.

[11] Marijn JH Heule and Sicco Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.

[12] Falk Howar and Bernhard Steffen. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 123–148. Springer, 2018.

[13] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*, pages 1–12. Springer, 1998.

[14] Kevin P Murphy et al. Passively learning finite automata. Santa Fe Institute, 1995.

[15] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, 2005.

[16] Frits Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.

[17] Sicco Verwer and Christian A Hammerschmidt. flexfringe: a passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642. IEEE, 2017.

[18] Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. Stamina: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering*, 18(4):791–824, 2013.

[19] Rick Wieman, Maurício Finavaro Aniche, Willem Lobbezoo, Sicco Verwer, and Arie van Deursen. An experience report on applying passive learning in a large-scale payment company. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 564–573. IEEE, 2017.