QUANTITATIVE VERIFICATION OF GOSSIP PROTOCOLS FOR CERTIFICATE TRANSPARENCY

by

MICHAEL COLIN OXFORD

A thesis submitted to the University of Birmingham for the degree of DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
December 2020

Abstract

Certificate transparency is a promising solution to publicly auditing Internet certificates. However, there is the potential of split-world attacks, where users are directed to fake versions of the log where they may accept fraudulent certificates. To ensure users are seeing the same version of a log, gossip protocols have been designed where users share and verify log-generated data. This thesis proposes a methodology of evaluating such protocols using probabilistic model checking, a collection of techniques for formally verifying properties of stochastic systems. It also describes the approach to modelling and verifying the protocols and analysing several aspects, including the success rate of detecting inconsistencies in gossip messages and its efficiency in terms of bandwidth. This thesis also compares different protocol variants and suggests ways to augment the protocol to improve performances, using model checking to verify the claims. To address uncertainty and unscalability issues within the models, this thesis shows how to transform models by allowing the probability of certain events to lie within a range of values, and abstract them to make the verification process more efficient. Lastly, by parameterising the models, this thesis shows how to search possible model configurations to find the worst-case behaviour for certain formal properties.

Acknowledgements

To Auntie Mary and Nanny Lee.

Writing this thesis could not have been accomplished after four tumultuous years alone. Firstly, I want to thank my co-supervisors, Dave Parker and Mark Ryan, for their unconditional support and always willing to make time for me despite having hectic work schedules. I could not have asked for more patient nor intelligent tutors.

I also want to thank: the Security Group for the interesting seminars and conversations over lunch; Graham Shaw and Adam Williams for overseeing my placement at Nettitude; Eike Ritter, David Galindo and Alice Miller for their useful comments on my work; Birmingham's BlueBEAR service for helping me complete some experiments; and the reviewers for the CNS'20 conference for their valuable feedback on the paper.

A number of close friends have helped me to overcome moments when I thought I would never finish this thesis. These people are: Paul Goldring; Kelvin Cheung; Pablo Vinuesa; Barri Matharu; Johnny Chan; and Susan Geng. Thank you all for some great memories and life advice.

Lastly, I want to thank my sisters, Rebecca and Sarah, and my wonderful parents, Agnes and Steven. I really cannot thank you all enough for what you have done for me to get here, so here is another - thank you.

Any mistakes found in this thesis are of course my own.

Contents

List of Figures **11**

List of Tables **13**

Glossary **17**

1 Introduction **21**

2 Related Work **27**

2.1 Transparency 27

2.2 Gossip and Auditing for CT 30

2.3 Probabilistic Model Checking 31

3 Background **37**

3.1 Certificate Transparency 37

3.2 CT Gossiping 50

3.3 Probabilistic Model Checking 55

3.4 Derivative-free Optimisation 74

4 Modelling and Verification of Gossip Protocols **81**

4.1 Network Topology 82

4.2 Modelling the Protocol 85

4.3 Specification of Protocol Properties 92
4.4 Server-to-server Gossip 94
4.5 Experimental Results 96
4.6 Summary 111

5 Tackling Uncertainty and Unscalability using IDTMCs 113

5.1 Using IDTMCs When Client Probabilities are Unknown 114
5.2 IDTMC Abstraction 121
5.3 Experimental Results 125
5.4 Summary 133

6 Model Parameter Optimisation 135

6.1 Deriving Network Model Parameters 135
6.2 Adapting the Black-box Optimisation Problem 139
6.3 Python Application 140
6.4 Experimental Results 146
6.5 Combining IDTMCs With SMBO 156
6.6 Summary 157

7 Discussion and Conclusion 161

Bibliography 163

Appendix A Constructing Components from IDTMCs and Updating ADTMCs 195

A.1 ConstructComponent 196
A.2 UpdateADTMC 196

Appendix B Deriving Distributions Using Surrogate Parameters 199

Appendix C Snapshots of PRISM Code **203**

C.1 Normal Scenario Model (Without Server Gossip) 203

C.2 Split-world Scenario Model With Intervals (Without Server Gossip) . . 212

List of Figures

3.1 A pair of Merkle hash trees 41

3.2 Communication flow of CT 46

3.3 Querying a certificate database 48

3.4 CT information for a certificate 49

3.5 An Illustration of a split-world attack 51

3.6 Illustration of the Chuat et al. CT gossip protocols 54

3.7 DTMC model example 57

3.8 Example abstraction of a DTMC 69

3.9 Demonstration of the Hyperopt library 79

3.10 Demonstration of the Benderopt library 80

4.1 Example of a network topology 84

4.2 Abstract representation of log growth 91

4.3 Model checking results for the normal scenario 98

4.4 Model checking results for the split-world scenario (1) 100

4.5 Model checking results for the split-world scenario (2) 101

4.6 Statistical results for the normal scenario 103

4.7 Statistical results for the split-world scenario 104

4.8 Box-and-whisker plots for randomly sampled data (Chapter 4) 105

4.9 Comparing statistical and verification results (normal) 108
4.10 Comparing statistical and verification results (split-world, init. design) 109
4.11 Comparing statistical and verification results (split-world, ext. design) . 110

5.1 Box-and-whisker plots for randomly sampled data (chapter 5) 119
5.2 IDTMC model checking results (normal) 120
5.3 IDTMC model checking results (split-world) 121
5.4 Abstraction process of an IDTMC 124
5.5 Comparing IDTMC and ADTMC verification results 132

6.1 Workflow of the optimiser code 145
6.2 Best result found for a fixed number of trials 149
6.3 Results for normal models using the suggested parameters 150
6.4 Results for split-world models using the suggested parameters 151
6.5 Statistical model checking results for larger models 152
6.6 Box-and-whisker plots for randomly sampled data (chapter 6) 154
6.7 Comparing verification results with simulation data 155
6.8 Investigation into the local behaviour of the objective function for normal scenario models 158
6.9 Investigation into the local behaviour of the objective function for split-world scenario models 159

List of Tables

4.1 Description of c_{sth}/s_{sth} variables 92

4.2 Initial modelling setup for both model types. For each client type, they connect with server types S^1 and S^2 with probabilities 0.02 and 0.28, respectively. They also connect with one other unique server with probability 0.7 e.g. the client type C^1 connects with server type S^3 with probability 0.7. 96

4.3 Model statistics (Chapter 4) 98

4.4 Client type frequency (Chapter 4) 102

4.5 Mean values for each proportion 106

5.1 Probability intervals (chapter 5) 116

5.2 Maximal/minimal values for each proportion (chapter 5) 119

5.3 Model statistics (chapter 5) 130

6.1 List of options for the Python application 143

6.2 Probability intervals (chapter 6) 147

6.3 Suggested modelling parameters 148

6.4 Client type frequency (chapter 6) 149

6.5 Maximal/minimal values for each proportion (chapter 6) 154

List of Algorithms

1 Generic sequential model-based optimisation (SMBO). 76

2 Constructing the abstract component 125

3 Building an ADTMC 126

4 Deriving a probability distribution from fixed intervals 138

5 Objective function to optimise 140

Glossary

$\mathbb{N}$	Set of natural numbers
$\mathbb{Z}$	Set of integer numbers
$\mathbb{R}$	Set of real numbers
PKI	Public Key Infrastructure
CA	Certificate authority
DNS	Domain name service
CT	Certificate Transparency
RFC	Request for comments
TLS	Transport Layer Security
h	Cryptographic hash function
URL	Uniform resource locator
SCT	Signed certificate timestamp
MMD	Maximum merge delay
STH	Signed tree head
OCSP	Online Certificate Status Protocol
HTTPS	Hypertext Transfer Protocol Secure
m	Gossip message data
sth	STH data
$\mathcal{M}$	Markov model

DTMC	Discrete time Markov chain
MDP	Markov decision process
Act	Set of actions
S	State space
E	Transition relation function
S_i	Set of initial states
P	Probability transition function
ι_{init}	Initial distribution function
AP	Set of atomic propositions
L	State labelling function
π	State-sequenced path
$FPath_{\mathcal{M},s}$	Set of all finite paths starting from state s
$FPath_{\mathcal{M}}$	Set of all finite paths
$IPath_{\mathcal{M},s}$	Set of all infinite paths starting from state s
$IPath_{\mathcal{M}}$	Set of all infinite paths
σ	Adversary function
Adv	Set of all adversaries
PCTL	Probabilistic Computational Tree Logic
ϕ	State-based PCTL formula or a general PCTL formula depending on the context
Φ	Path-based PCTL formula
$\mathbf{P}$	Probabilistic path operator
$\mathtt{X}$	Next operator
$\mathtt{U}$	Until operator
$\models_{Adv}$	Satisfied under Adv
$\mathbf{F}$	Future operator

$\Diamond$	Future operator (alternative)
r_{state}	State reward function
r_{action}	Transition reward function
T	Target set of states
$Prob$	Probability of an event
$\mathbf{I}^{=k}$	Instantaneous reward after exactly $k \in \mathbb{N}$ steps
$\mathbf{C}^{\leq k}$	Cumulative reward after exactly $k \in \mathbb{N}$ steps
R^T	Reachability reward before reaching target set T
$\mathbf{R}$	Reward operator
IDTMC	Interval DTMC
ADTMC	Abstract DTMC
F	Objective function
SMBO	Sequential model-based optimization
$\mathcal{N}$	Surrogate model
A	acquisition function
$\mathcal{H}$	Observation history set
TPE	Tree-structured Parzen estimator
NT	Network topology
$\mathcal{G}$	Gossip rate function
$\mathcal{P}$	Client type profile function
$\mathcal{C}$	Set of client types in NT
C^i	Client type i
$\mathcal{S}$	Set of server types in NT
S^j	Server type j
$\mathcal{M}_{NT}$	DTMC network model for network topology NT
δ	Probability distribution function

Pmin	Minimal probabilistic path operator
Pmax	Maximal probabilistic path operator
s_{abs}	abstract state (i.e a subset of concrete states)
$\underline{\mathbf{F}}$	Client type frequency
$\underline{\mathbf{I}}$	Set of possible initial states for $\mathcal{M}_{NT}$
$\underline{\mathbf{G}}$	Gossip rate vector
$\underline{\mathbf{x}}$	Surrogate choice vector
$\underline{\mathbf{X}}$	Surrogate choice matrix
χ	Modelling/configuration space

Chapter 1

Introduction

Certificate transparency (CT) [42] has shown to be a promising system that promotes the public auditing of public key certificates, which are hugely important for web domains to identify themselves when forming secure Internet connections with visitors. Supported by both the Google Chrome browser [168] and the Apple Safari browser [6], it continues to show promise in exposing any negligent actions by certificate issuers [176].

An open security issue with CT is that it doesn't prevent an attacker, namely a log maintainer, from broadcasting their erroneous version of their CT log containing fraudulent certificates used to stage man-in-the-middle attacks, convincing a segregated population of internet users that it is legitimate. This type of attack, called a *split-world attack*, involves forking and hosting a rogue version of a log to targeted victims who then willingly accept anything included in this log because they have no way to validate amongst themselves that their view of this log is the legitimate version hosted to non-victims.

To retroactively detect split-world attacks in CT and deter any misbehaviour from a log maintainer, one solution is to make devices perform random peer-to-peer

communication to disseminate data via *gossip protocols*, so that they can exchange log-generated data with each other. That way, devices will be able to utilise log data coming from multiple sources and collectively agree on what the correct state of the log should be, making sure that the non-equivocation property is being respected by the log maintainer i.e. it is not trying to host multiple contradicting states of the log to different sets of users simultaneously.

In 2015, Chuat et al. [48] designed and tested protocols which make clients gossip data through domain servers they regularly connect with using HTTPS connections, making each type of entity validate for log consistency and update their local states whenever they discover newer data. While they demonstrated the effectiveness of the protocols using simulations, their methodology lacked any manner of formal analysis and were not able to measure how effective the protocols were in detecting inconsistencies in the log data when a split-world attack was occurring.

Formal verification uses mathematically-based techniques to prove that a particular algorithm or protocol meets formally defined specifications. While formal verification comes in many forms [154, 122, 201], the one which we focus on in this thesis is known as *model checking* [49, 179]. Using model transition systems [133], distinct states of a system are encapsulated and the transitions from each of them determine what may or must follow from them (or there is an absence of transitions, in which case nothing happens). By characterising a desired specification formally using a temporal logic, the transition system is explored exhaustively to determine if it is possible to reach states that meet or violate that specification. To sufficiently model systems which exhibit stochastic behaviour, we can augment transition systems further by assigning a probability distribution to the set of outgoing transitions for each state, representing a Markovian process where the likelihood of an event happening only depends on the current state of the model. This sub-field of model checking is known as *probabilistic*

model checking [130].

This thesis attempts to bridge the gap between probabilistic model checking and CT using the Chuat et al. gossip protocols (for convenience, we call call them the *Chuat protocols*) as a case study to investigate how effective they are at disseminating log-based data to retroactively detect split-world attacks. We note that the objective of this thesis is not to prove the correctness of the protocol logic so that it satisfies formal security properties, but instead employs a set of metrics computed iteratively when a unit of time has passed to evaluate protocol performance.

The thesis consists of three parts: the first part (Chapter 4) explains how we can model a network using discrete time Markov chains, having clients randomly connect and gossip with servers who then update themselves with the gossip messages they receive. We describe some formal specifications to measure the security and efficiency aspects of the Chuat protocols, suggesting how we can improve on the initial designs and validate our claims using model checking. We use the PRISM model checker [129] to give a rich analysis of the protocols we analyse.

The second part (Chapter 5) looks at how to accommodate for *uncertainty in models* when it is hard to capture exactly the random behaviour of gossiping clients, and *model scalability* when the model checking process becomes too expensive to perform. To resolve the uncertainty issue, we use interval discrete time Markov chains (IDTMCs) [117], replacing exact probabilistic values with intervals to show under- and over- approximations of transition probabilities, thereby providing us with lower and upper bounds on the quantitative properties we wish to analyse. To resolve the scalability issue, we show how to transform our IDTMC models appropriately via abstraction into smaller models which probabilistically simulate the former, called abstract discrete time Markov chains (ADTMCs), which are less costly to perform verification on but as a trade-off lose accuracy in the results [119]. We apply

extensions of PRISM which are capable of building and verifying IDTMC models, additionally abstracting them by dynamically mapping states to their abstract counterparts in the resulting ADTMC model.

The third and final part (Chapter 6) explains how to parameterise our DTMC models and search over their possible configurations using *sequential-based model optimisation* (SMBO) [109] to identify a configuration that best optimises particular quantitative properties. While we take into account the possible transition probabilities in the model like with IDTMCs, we have the added complexity of searching over all initial setups of the network we are trying to model i.e. the set of possible initial states for the probabilistic model, which is expensive in terms of model checking effort. We describe an algorithm which takes as input a set of modelling parameters, constructs a DTMC model based on those parameters and then perform verification on it via PRISM using a pre-determined property, outputting a real value. By treating this algorithm as a 'black-box' function, we use SMBO to try and fit a cheaper surrogate function to it using a finite series of inputs chosen by SMBO and their respective outputs, finally deciding on a worst-case network configuration which maximises the quantitative result of our chosen property. We show empirically how this technique produces better findings compared to using random sampling of client data.

We intend for our primary audience to be security researchers wishing to see how verification could be applied to investigating gossip protocols for decentralised systems. However, this work should also be of interest to users of model checking across a wide range of application domains.

The content of this thesis is structured as follows: in Chapter 2, we list related work and point to other areas of research the interested reader can pursue further. Chapter 3 covers the fundamental background material for this thesis including CT,

gossiping, probabilistic model checking and SMBO. Chapter 4 shows how to model a type of gossiping network in two different scenarios using DTMCs, and uses temporal logic to describe the properties we want to analyse. Chapter 5 discusses how to address model uncertainty and scalability by introducing extensions of PRISM which performs verification on IDTMCs embedded with uncertainties reflecting the random behaviour of the clients. We show how we can reduce the complexity of this process via abstraction which uses state partitioning to collapse the size of the models. Chapter 6 shows an application of SMBO to maximise certain quantitative properties of the gossip protocols by exploring a 'model space' which characterises all the possible network scenarios. Chapter 7 concludes this thesis and possibilities for future work are discussed.

Some of the content of this thesis, in particular from Chapters 4 and 6, is published in the following paper:

- M. Oxford and D. Parker and M. Ryan; *Quantitative Verification of Certificate Transparency Gossip Protocols*. In: *The Sixth International Workshop on Security and Privacy in the Cloud (SPC'20)*, July 2020, IEEE.

Supporting material, including PRISM files and source code, for this thesis can be found at [202].

Chapter 2

Related Work

2.1 Transparency

The monitoring of public key certificates, first introduced by Kohnfelder [126] in his bachelor's thesis, has remained a difficult problem to solve since the early days of the Internet, largely because Internet usage and human reliance on digital services has significantly increased in recent decades [112]. Some early attempts included the SSL observatory project [208] conducted in 2010 by the Electronic Frontier Foundation which scanned datasets of publicly-visible certificates, real-time notary services which periodically scanned areas of the Internet [5, 158, 205], incorporating mechanisms into DNSSEC to create records which associated certificates with hosts which anyone could check [187, 98] and public key pinning (PKP) [76]. Due to low adoption rates and issues regarding how operators would be unable to maintain website content should they lose any of their public keys, PKP has now been deprecated in Google Chrome since version 67 [172].

Certificate transparency (CT), devised by Laurie et al. [139] in 2013, is designed to be an open and scalable system where anyone can participate in the auditing of

certificates. A collection of client tools and libraries for interacting with CT logs can be found online [87] and CT has already been enabled for Google Chrome; since April 2018, it is required that all TLS certificates be appended to certificate transparency logs for Chrome to accept any connections [168]. There are plans to update and replace some of the existing CT formats and mechanisms, all of which are still in the experimental phase [140]. Work has also been done on CT outside of Google to extend its capabilities and validate its effectiveness. In 2014, Ryan [184] extended CT to efficiently handle certificate revocation and applied it to devise end-to-end encrypted email without the need for complex key-signing arrangements or trusted third parties. In 2016, Dowling et al. [66] proved security properties for CT under particular modelling assumptions, showing how it prevents log misbehaviour and protects honest log maintainers from misbehaving log monitors. In 2017, Eskandarian et al. [74] used zero-knowledge proofs to show how browsers could audit CT logs for misbehaviour without risking user privacy and how CT can be compatible with private domains without leaking information about them or any 'short-lived' certificates they used which lasted only a day. In 2018, Madala et al. [148] supplemented CT with blockchains with the aim of giving domain owners more authority over which certificates get issued in their name, claiming that it makes the detection of fraudulent certificates more proactive.

There have been studies, in particular by Gasser et al. and Vandersloot et al. [186, 85, 212], looking at how CT has impacted the public key certificate ecosystem and showing how powerful it is in collating certificate data. Furthermore, in 2017, Gustafsson et al. studied the characteristics for different CT logs and compared their contents for differences and similarities, especially between Google-operated and CA-operated logs [95]. More recently, in 2019, Li et al. [142] gave a sceptical analysis of how truly reliable CT is in finding fraudulent certificates because, even by scanning a variety of logs, it is never guaranteed that the union of their results will return all

the certificates for the domains they investigated.

Many other transparency or decentralised systems exist which aim to solve problems similar to certificate monitoring. Sovereign keys [71], Revocation transparency [138], PoliCert [207], AKI [123], ARPKI [19], DTKI [222], AAD [210], CertCoin [83], IKP [151], BlockPKI [69] and DomainPKI [214] all deal with transparency in public key infrastructures. CIRT [184], CONIKS [155], EthIKS [29], KAS [75], DECIM [223] and Google's key transparency project [88] focus on providing a way of authenticating public keys for peer-to-peer services and message encryption. Catena [211] and Contour [4] both use the bitcoin blockchain (see the bitcoin paper [162]) to provide binary transparency for distributed software packages. Attestation transparency [20] looks at how to ensure that Internet services are running correctly for all clients and are not running a malicious version of the service to targeted victims. Google is also experimenting with running general transparency logs as part of their Trillian project [89].

Very recently, Szalachowski [206] introduced a radical redesign of the certificate ecosystem in the form of SmartCert, combining both the distribution, maintenance and revocation processes for certificates using smart contracts. In this system, SmartCert treats certificates as dynamic objects being periodically updated by CAs using the Ethereum blockchain [220] or any other smart contract platform, allowing for the addition of any information needed in certificates so clients can properly validate them without the need to contact a third-party. Using a *policy contract* which defines the conditions for a domain's certificate to be deemed valid, coupled with a *SmartCert contract* updated by authorised CAs which records the validation state of the key pair being used by the domain, SmartCert minimises the trust needed in CAs and gives domains more power to enforce their policies for certificates since any newer SmartCert contracts (and their corresponding certificates) cannot be

created without their consent. However, implementing SmartCert, even incrementally as the author suggested, is an enormous challenge to get right. In addition the systems places more responsibility on the CA by making them instigate periodic validation checks for potentially millions of domain names.

2.2 Gossip and Auditing for CT

Gossip protocols, also known as epidemic protocols, were first conceived in the late 1980s as a way to maintain consistency between databases replicated and stored at different sites [62]. Generally speaking, a gossip protocol makes nodes in distributed systems form peer-to-peer connections with other adjacent nodes and disseminate specific data, the goal of this being to ensure that a majority of the nodes have common knowledge about the state of an entity belonging to that same system. Much of the work in this thesis is based on the gossip protocols devised by Chuat et al. [48] where they make clients gossip digests of CT logs and update their local states when newer data arrives, having servers act as staging posts. Chuat et al. [215] have also recently introduced a secure consensus protocol which replicates the state of the CT log among independent entities so that they share responsibility of maintaining the log, using proof assistants to show that the protocol satisfies certain security properties and practical experiments to see how well it scales in a CT-like system as the number of replicating nodes increase. Nordberg et al. [166] have also developed gossiping mechanisms where auditors collect and validate log digests from servers whilst treating the data as sensitive so that they cannot be linked back to clients. Dahlberg et al. [53] have proposed aggregation-based gossiping by having routers or an intermediary collect log data as it passes through them and consistency checks are done via an off-path; this may be more suitable towards internal corporate networks where administrators have more control over what enters

and leaves.

We should note that KAS [75] and CONIKs [155] use a form of server-side gossiping in their schemes as opposed to client-side gossiping to minimise the burden on users and not pass on the responsibility of constantly checking for unusual behaviour from logs when they get updated. As an alternative (or supplement) to retroactive gossiping, Syta et al. [203] introduce witness co-signing where certificate authority activity is monitored by a small set of decentralised witnesses; an attacker cannot make a client use their certificate unless they can control the witnesses but this is an unlikely scenario if the witnesses are made diverse enough.

Services on the Internet already exist that users and businesses can access to check the current status of active logs or see which certificates have been submitted for particular domains [72, 78, 41, 188]. Dahlberg and Pulls [52] have also developed light-weight monitoring tools to reduce the trust placed in a small set of capable third party auditors.

2.3 Probabilistic Model Checking

Probabilistic model checking combines both model checking, originating independently from the works of Clarke and Emerson [49] and Queille and Sifakis [179], and Markovian processes which have been studied throughout the twentieth century [84, 106]. Numerous model checking tools exist that can perform automated verification on Markov models; the one used for this thesis is the PRISM symbolic model checker [129]. Other tools which may be of interest include STORM [58] and PAT [143].

2.3.1 Model Checking for Security Protocols

There have been well-documented cases of (non-probabilistic) model checking being used to find vulnerabilities within security protocols. In 1996, Lowe [146] was able to prove and provide examples of weaknesses in the Needham-Schroeder public-key protocol [163] using the Failures Divergence Refinements Checker (FDR). He demonstrated it in the case of an attack consisting of two interleaving runs of the protocol and listed a sequence of events which led to a breach of security. It was analogous to a man-in-the-middle attack where an initiator established a session with an intruder pretending to be the responder. Using FDR, Lowe was able to prove that, by amending the protocol in a certain way, no attacks could be found on the protocol when used in a small system.

In the case of PRISM, there are examples of it being applied to security problems. In 2006, Norman and Shmatikov [167] used PRISM to investigate EGL (Even, Goldreich, and Lempel), a contract signing protocol where clients who do not trust each other exchange commitments to a contract to share data [77], showing that the protocol did not satisfy a crucial property. They also suggested ways of how to minimise the chance of this particular property being violated by amending the original protocol. In that same year, Steel [198] was able to demonstrate using PRISM that, by exploiting APIs in tamper-proof hardware security modules which were being used in ATM networks, PIN digits could be revealed via a reformatting attack. Markov chains were used to represent both the probabilistic and non-deterministic choices an attacker can make when trying to leak data about the PIN number. Mirto et al. [159] have expressed an interest in using PRISM to analyse the probabilistic nature of consensus protocols used in blockchains, such as Proof-of-Work and Proof-of-Stake, and determine how robust they are in the presence of malicious actors and unequal computing power distribution amongst the nodes. Additionally, Marinković et al. [150] have introduced extensions of existing

temporal logics to reason about blockchain consensus protocols probabilistically.

2.3.2 Analysing Gossip Protocols via Model Checking

There have been studies of applying probabilistic model checking techniques to analyse gossip protocols during the late 2000s [118, 14]. For early examples which used PRISM, Fehnker and Gao [81] analysed the performance of gossiping and flooding protocols in a small network using while considering cases where network collisions and data loss occur, combining this with large-scale simulations to see if their verification results were consistent for more realistic network models. Kwiatkowska et al. [128] provided quantitative analysis of a gossip protocol which used random peer sampling in small networks [114]. The models included a scheduler which determined the order in which nodes executed the protocol, keeping track of the ones which have already exchanged data. In 2018, Webster et al. [216] studied a gossip protocol used in wireless sensor networks to analyse the synchronisation of nodes and the dissemination of information, showing how slight changes in the temperature of the sensor affects the quantitative results. Unlike the models used in these previous works, the ones used in this thesis are based on a client-server network model where clients are unable to communicate with each other and must gossip information through servers.

2.3.3 Verification Approaches for Uncertainty

The idea to extend DTMCs with interval probabilities was first proposed by Jonsson and Larsen [117] and was extended upon by Kozine and Utkin [127], Škulj [193] and Sen et al. [190]. Additionally, Delahaye et al. [60] introduced a framework which refined a set of interval Markov chains which modelled different aspects of a process to enable compositional modelling and Sproston, Chakraborty and Katoen [44, 196] studied Markov models which used open intervals instead of closed ones. In the more

complex case where any of the probabilistic bounds are unknown, Delahaye et al. [59, 61, 16] provide a comprehensive study of parametric interval Markov chains. Chonev [47] recently introduced an augmentation of Interval Markov chains to allow for any dependencies between different transition probabilities, studying how this affected the complexity of the reachability problem.

For performing automated verification on models with unknown parameters, Calinescu et al. [37, 38] introduced the FACT (formal verification with confidence intervals) probabilistic model checker. FACT computes confidence intervals for both probabilistic and reward-based properties for parametric DTMCs whose probabilities for certain transitions are unknown but recorded observations of them are available [113, 97]. FACT uses an extension of PRISM's functionalities to express parametric Markov chains and obtain an algebraic expression for PCTL properties. As part of the process of synthesising confidence intervals for quantitative properties, external convex optimisers can be used which are compatible with FACT, such as MATLAB's YALMIP [145], GNU Octave [70, 91] and the Gurobi optimiser [94].

2.3.4 Verification Approaches for Scalability

A common problem in verification is how to model processes appropriately without running into state space explosion which makes verification for complex specifications infeasible [51]. Traditional techniques to alleviate this problem include efficient symbolic algorithms which use binary decision trees [102], partial-order reduction [93] and bi-simulation methods [134, 12, 39, 131]. This thesis focuses on abstraction methods which aim to produce upper and lower bounds for specifications; in particular, we use the lumping technique which partitions the states in a large model to produce smaller ones which simulates the former [35]. Regarding abstracting labelled DTMCs, much work has been done in this field by Huth et al. [92, 107], Katoen et al. [119] and Fecher

et al. [80]. Abstraction techniques have also been developed for CTMCs [63, 120, 124, 101] and MDPs [54, 160, 173, 56, 103, 183].

In 2019, Ouadjaout and Miné [170] performed static analysis of wireless communication protocols using possibly infinitely-sized Markov models, assessing the performance of such protocols by computing the *stationary distribution*, which gives the overall proportion of time spent in every reachable state over all possible runs of the model [199]. By analysing the protocol source code, they were able to formulate finite abstract models to approximate the properties of their original counterparts, producing parametric bounds on properties of interest and extended the work to analyse non-deterministic models. Whereas Ouadjaout and Miné only focus on the analysis of one node in the network and modelled protocols at a low level down to their programming code, our work looks at the general behaviour of particular protocols and analyses their performance of them over a fixed period of time in multi-node networks.

2.3.5 Model Parameter Optimisation

In the context of CTMCs modelling biological systems, researchers have recently tried to bridge the disciplines of model checking and optimal parameter searching. Bortolussi et al. [18, 32, 30] addressed the *system design problem* to optimise parameters for models via Bayesian inference to maximise, for example, the robustness of desired specifications i.e. the probability of satisfying a property remains relatively unaffected by any perturbations in the model. To address the *parameter synthesis problem* which tries to find parameter values for models leading to the satisfiability of properties or matching the observed real-time behaviour of a process, automated tools have been developed which utilise the PRISM language, including PROPhESY [57] which focuses on parametric Markov chains and U-check [31] which employs Gaussian

processes, both of which include user-friendly GUI interfaces. Borrowing ideas from machine learning, statistical techniques have been devised to find, with a degree of confidence, the likelihood of satisfying a property or a range of possible parameters to optimise it given that the knowledge of a model is incomplete [175, 96, 17, 33].

In contrast to previous work, we use an existing optimisation method applied to machine learning problems to attempt to solve a specific problem involving DTMCs, where the underlying model structure is fixed yet we search over both probabilities and parameters related to which client types are present. Furthermore, we are not interested in which parameters of a model lead to the satisfiability of a property or match observed behaviour but rather ones which produce a worst-case scenario for important quantitative aspects of a protocol.

Chapter 3

Background

In this chapter, we cover the fundamental material needed to understand the later chapters of this thesis, including certificate transparency (CT), gossiping in CT, probabilistic model checking and sequential model-based optimisation (SMBO).

3.1 Certificate Transparency

In this section, we talk about certificate transparency, a technique proposed in the early 2010s which allows anyone to audit public key certificates to deter misbehaviour from certificate distributors.

3.1.1 Motivation

When a user's web browser (a *client*) connects to a website (hosted by a *server*), establishing secure connections by public key cryptography is insufficient to ensure that sensitive information is protected. There is the chance that the client is connected to a spoof website that is trying to steal their sensitive information, and the public key being used to encrypt the connection does not belong to the company hosting the

genuine website.

To resolve this issue, *public-key infrastructures*, or PKIs [36], are used to manage the creation and revocation of public-key certificates, digitally signed documents that bind a public key to its owner. PKIs are usually composed of the following [2]:

- *Certificate Authority* (CA): A trusted third party which authenticates the identities of individuals or companies by signing digital certificates.
- *Registration Authority* (RA): Issues any certificates to customers on behalf of the root CA that signed them.
- *Certificate Database*: A database to store or revoke issued certificates.

The weakest link in PKIs is that users must trust CAs to distribute certificates properly, otherwise it can lead to disastrous outcomes. An infamous example is the defunct CA Diginotar where, in 2011, it was revealed that at least 500 fraudulent certificates were issued in their name and they failed to keep any record of them. When the incident was reported, this halted many financial and governmental services from taking place over the Internet, and affected websites had to quickly obtain newer certificates. It was also later found out that Diginotar had poor security management in place, using out-of-date software and inadequate server-side protections [219, 141].

Due to this incident and other ones involving different CAs such as Comodo (now Sectigo) and Globalsign [218, 182], researchers began to investigate ways to allow anyone to see and audit the distribution of certificates, thereby reducing the need to trust CAs. We describe a few of these proposals below [135]:

- *Certificate pinning* - Websites advertise which public keys are suitable for them by associating (or 'pinning') them with their corresponding certificates and the web browser will reject any certificates which have not been pinned. However, these

pins expire after a fixed period of time and if a website loses its public key before this time expires, no-one will be able to access the site until a newer certificate is issued [76].

- *Notaries* - The Internet is scanned from multiple areas, inserting all known certificates into a notary log which is called by users when they receive unknown certificates. However, this mechanism is likely to generate a lot of false positive results because, for example, a certificate which is deemed suspicious may only have just been issued and be replacing a known one for a particular website [137].

- DNSSEC - Using DNS-based solutions, in particular the mechanisms of DNS-based authentication of named entities (DANE) and certification authority authorization (CAA), this extension of DNSSEC associates DNS records for hosts with particular certificates or CAs. Clients check DANE records when they connect to websites, whilst CAs check CAA records when issuing certificates to hosts, refusing to do so if the host is not included in any record. However, they introduce more trusted third parties such as DNS registries and, due to the distributive nature of DNS, it is near impossible to preserve the integrity of DNS records when being viewed by different parties, a situation which attackers can easily exploit [187, 98].

3.1.2 Introduction

Certificate transparency, or CT, is a system which uses a public, append-only log which anybody can interact with and monitor. Certificates get added to the CT log in a special way so that the log can efficiently prove that i) a certificate has been appended properly and ii) the latest state of the log is an extension of a previously seen one. Since

the log is append-only, everyone will be alerted if an attacker tries to tamper with its contents and replace genuine certificates with fraudulent ones. CT was first developed by Laurie, Langley and Kasper, submitting the first RFC for it in 2013 [139]. A newer experimental RFC, titled *Certificate Transparency Version* 2.0, was later released in 2014 and will obsolete the previous RFC once it has been finalised and approved. CT is currently being pioneered by Google and the code for it can be found online [87].

Merkle Hash Trees

A Merkle hash tree, first introduced by Merkle [156] in his PhD thesis in 1979, is a rooted binary tree where each leaf contains a block of cryptographically hashed data. For a non-leaf node n with child nodes L and R, if h_L and h_R are the hashed data stored at L and R respectively, then n contains the hashed data $h(h_L \,||\, h_R)$, where $||$ denotes string concatenation and h is a hash function. The *size* of the Merkle tree is defined to be the number of leaves it has; if a Merkle tree has size zero, it is by default a single node representing the hash of an empty string. In the context of CT, the leaves of a Merkle tree represent the hashed data of certificates appended to the corresponding log and are always arranged in the chronological order they get added. The *root hash* at the root node is the encapsulation of all the data present in the Merkle tree.

For an example, Fig. 3.1(a) depicts a Merkle tree with six leaves, meaning that this corresponds to a log containing six certificates $cert_1, \ldots, cert_6$. The leaves are given by $h_i = h(cert_i)$, where $i = 1, \ldots, 6$. By concatenating the hashes h_1 and h_2, for example, we obtain the hash for the parent node given by $h(h_1 \,||\, h_2) = h_{1||2}$. The root hash of

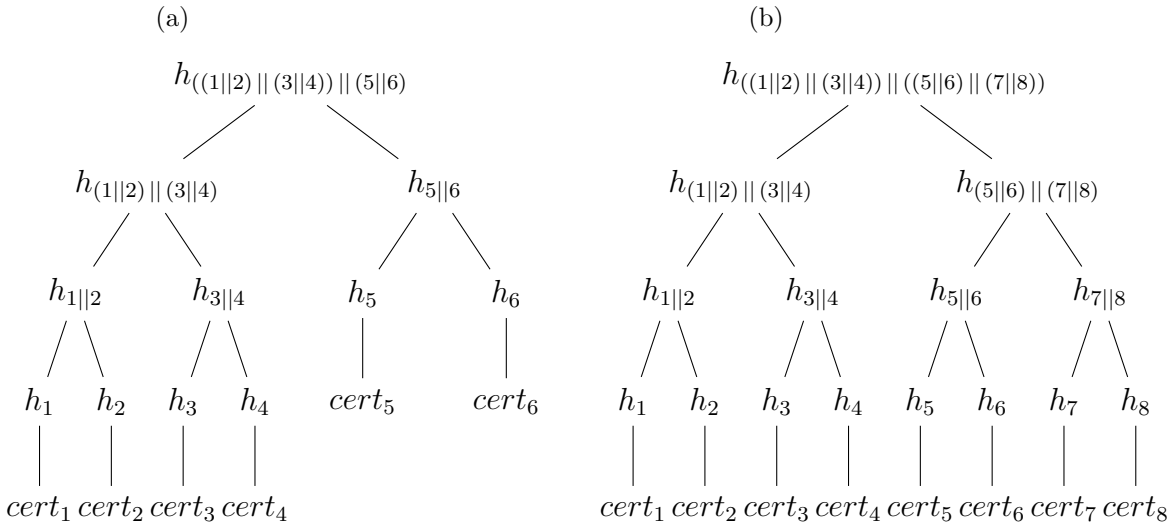


Figure 3.1: A pair of Merkle hash trees with (a) six leaves and (b) eight leaves. Adapted from [42].

the tree is given by

$$
\begin{aligned}
h\Big(h\Big(h(h_1\,||\,h_2)\,\Big|\Big|\,h(h_3\,||\,h_4)\Big)\,\Big|\Big|\,h(h_5\,||\,h_6)\Big) &= h\Big(h\Big(h_{1||2}\,\Big|\Big|\,h_{3||4}\Big)\,\Big|\Big|\,h_{5||6}\Big) \\
&= h(h_{(1||2)\,||\,(3||4)}\,||\,h_{5||6}) \\
&= h_{((1||2)\,||\,(3||4))\,||\,(5||6)}
\end{aligned}
$$

Audit Proofs

In the context of CT, given a hash h^*, a *proof* is fundamentally the minimal set of hashes needed already present that are present in the Merkle tree to reproduce the root hash of the tree using h^*; if no such proof is possible to produce, then it is by default an empty set. Given a certificate, an *audit proof* verifies that it is included in the log by listing the hashes needed to reproduce the current root hash using the hash of the certificate itself. It can be shown [156] that the size of such a list is at most logarithmic to the number of leaves in the Merkle tree, in other words, it is proportional to the number of layers the Merkle tree has. For example, in Fig. 3.1(a), to show that

certificate data $cert_4$ is in the log, observe that:

$$h_{3||4} = h\left(\underline{\boldsymbol{h_3}} \,||\, h_4\right);$$
$$h_{(1||2)\,||\,(3||4)} = h\left(\underline{\boldsymbol{h_{1||2}}} \,||\, h_{3||4}\right);$$
$$h_{((1||2)\,||\,(3||4))\,||\,(5||6)} = h\left(h_{(1||2)\,||\,(3||4)} \,||\, \underline{\boldsymbol{h_{5||6}}}\right).$$

Therefore, the audit proof for $cert_4$ is $\{h_3, h_{1||2}, h_{5||6}\}$.

Consistency Proofs

To make sure that the log respects the append-only property, *consistency proofs* (or *extension proofs*) are required and are generated similarly to audit proofs. Using the root hash from a previously seen snapshot of a Merkle tree, a consistency proof verifies that a more recent snapshot of the tree naturally extends the former. For a Merkle tree t_n with root hash h_n, suppose that t_m is a previously seen state of the tree with root hash h_m. To show that t_n extends t_m, the consistency proof needs to provide the necessary hashes to show that the following is true [42]:

i) h_m can be reproduced from hashes in t_n, and

ii) h_n can be reproduced from h_m plus any of the node hashes in t_n which are not present in t_m.

In Fig. 3.1, to show that tree (b) is an extension of tree (a), note that hashes $h_{(1||2)\,||\,(3||4)}$ and $h_{5||6}$ in (b) are necessary to reproduce $h_{((1||2)\,||\,(3||4))\,||\,(5||6)}$ and

$$h_{(5||6)\,||\,(7||8)} = h\left(\underline{\boldsymbol{h_{5||6}}} \,||\, \underline{\boldsymbol{h_{7||8}}}\right);$$
$$h_{((1||2)\,||\,(3||4))\,||\,((5||6)\,||\,(7||8))} = h\left(\underline{\boldsymbol{h_{(1||2)\,||\,(3||4)}}} \,||\, h_{(5||6)\,||\,(7||8)}\right).$$

Therefore, the consistency proof between trees (a) and (b) is $\{h_{5||6}, h_{7||8}, h_{(1||2)\,||\,(3||4)}\}$.

3.1.3 Log Format

This subsection describes how a CT log formats certificate data and the types of data it produces for auditing purposes.

Log Entries

Certificates which are given to a log to be appended can take the form of a *leaf certificate*, which is a newly issued certificate to be used for validating Internet connections, or a *precertificate* that acts as a 'proof of submission' for the final certificate soon to be issued by a CA. To distinguish precertificates from leaf certificates, a 'poison extension' is applied so that they are immediately deemed invalid by a client [147]. For any given log entry, it will contain the certificate chain up to the certificate root, the entry index and a Boolean flag which indicates the certificate type.

Anyone with access to a web browser can obtain a set of entries from a log using HTTP GET commands [139]. For example, suppose we want contact a log with the URL url to check a certificate with entry index index. We also want an audit proof for the certificate when the log was of size size. This can be done by typing the following in the URL address bar:

$$\langle \mathit{url} \rangle \texttt{/ct/v1/get-entry-and-proof?leaf\_index=} \langle \mathit{index} \rangle \texttt{\&tree\_size=} \langle \mathit{size} \rangle \tag{3.1}$$

The output of (3.1) gives the following:

- A Merkle tree leaf input,
- a log entry showing the encoded certificate data,
- the list of Merkle tree nodes which proves the inclusion of the entry.

Signed Certificate Timestamp

Whenever someone submits to a log, they are given a digitally signed receipt called a *signed certificate timestamp* (SCT) to indicate that the log will 'promise' to append the given certificate within a predetermined period of time called the *maximum merge delay* (MMD). In practice, the MMD is usually twenty-four hours. SCTs act as insurance in case the log does not append the corresponding certificates in the allotted time and can be immediately used to prove such misbehaviour. An SCT contains the following:

- A hash of the log's public key which acts as the log ID,
- A timestamp indicating when the certificate was accepted by the log (note that it is not the time when it was appended by the log),
- A digest of the corresponding certificate.

Signed Tree Head

A *signed tree head* (STH) is created whenever the CT log digitally signs the current Merkle root hash with its own public key, and is updated each time a new certificate gets appended (and hence a new root hash is created). The current STH of a log must not be older than an MMD; if the log receives no new submissions within an MMD period, then the log signs the same STH with a fresh timestamp. An STH contains the following:

- A timestamp indicating when the STH was updated,
- The size of the Merkle tree when the STH was updated,
- The root hash of the Merkle tree.

STHs are useful records whenever one needs to prove log consistency or certificate inclusion and anyone can request the current STH from the log at any time. If we want

the latest STH generated from a log with URL *url*, we can type the following in a URL address bar:

⟨*url*⟩/ct/v1/get-sth

3.1.4 CT in Practice

This subsection gives a basic explanation of how CT is used during normal Internet operations and the mechanisms that can be used to deliver SCTs with certificates. We also describe how logs can be audited and provide a list of available tools which can extract information from CT logs. Lastly, we briefly discuss how CT is integrated into the Google Chrome browser.

Basic Procedure

Fig. 3.2 illustrates how certificates get added to a CT log and passed onto clients with the corresponding SCT. First, before anyone can accept incoming connections from a website, the host must request a certificate from a CA which it then creates, submitting it to a log server on the host's behalf if desired. When the log server accepts the certificate, it issues an SCT for which the host must somehow obtain and bundle with the newly issued certificate, using both objects in future TLS connections so clients can validate the website.

To deliver SCTs to clients along with the certificates, three options are available which each involve different levels of responsibility from both the host and the CA [105, 42]:

- *X.509 extensions*: The SCTs are embedded into the certificate itself which clients can readily view. These SCTs are acquired by submitting precertificates to different logs (see subsubsection 3.1.3). This does not include any server-side modifications and responsibility for obtaining the SCT falls solely on the CA,

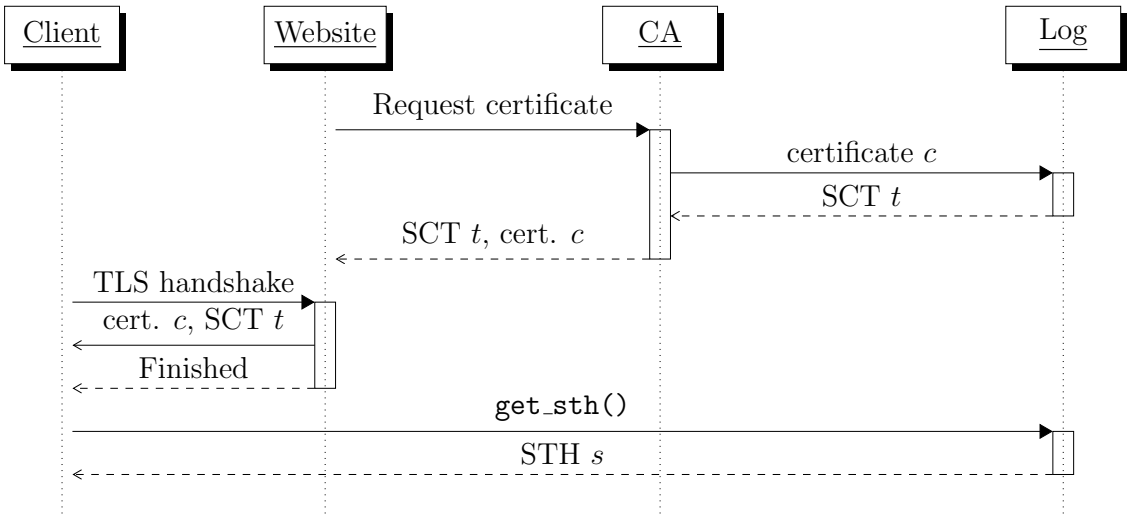


Figure 3.2: Communication flow of CT between a website, certificate authority (CA), log server and client. Anyone is capable of asking for a signed tree head (STH) at any time which contains up-to-date information about the log. In this figure, we assume that X.509 extensions are being used to deliver signed certificate timestamps (SCTs) with the certificates.

making this method the most popular choice for domain owners. Due to this, the CA *Let's Encrypt* automatically embeds SCTs into their newly issued certificates [104].

- *TLS extensions*: For this method, the domain owner is responsible for sending a request to the log server for their newly issued certificate to be appended, obtaining an SCT in the process which they must show to clients with the certificate. This does not alter the way CAs issue certificates and server operators need to accommodate for such extensions which can be risky and error-prone.

- *Online certificate status protocol (OCSP) stapling* : This is where the CA sends the certificate to both the website and log at the same time. The host then makes an OCSP query [185] to the CA to get the SCTs which they include in TLS handshake extensions. OCSP stapling reduces the delay in issuing certificates and both the CA and host share the responsibility of following the correct procedure.

However, server operators still need to accommodate for OCSP stapling which can be difficult to get right.

Log Auditing

The aim of *monitors* (or *auditors*) in CT is to continuously inspect logs and check for misbehaviour by collating and analysing log-generated data from multiple Internet sources. Monitors can exist for a variety of purposes, such as working on behalf of organisations by flagging up suspicious certificates in their name or working independently for research purposes. Some of the things monitors look out for when inspecting a log include [140]:

- Making sure the log respects the MMD and does not violate any issued SCTs.
- The STH frequency count i.e. the number of STHs generated by the log in an MMD period. This is to ensure that the log does not produce too many unique STHs that could mark individual clients when they request them.
- Checking that the log maintains its consistency across different parts of the Internet by verifying for consistency between multiple STHs.

Anyone is allowed to set up and maintain an auditor and there are already existing tools which fulfil this role. Cert Spotter [41] is a commercial monitor which scans all known logs which customers can query using a JSON API. There are also monitors which are free-to-view such as Facebook's monitoring tool [78, 43] and another managed by Edgecombe on his website [72]. Sectigo and Censys also host their own certificate databases which anyone can inspect [188, 40]. Note that, as Li et al. has shown [142], third party monitors of CT logs tend to be unreliable when consulting them for the complete sets of certificates for domains, mainly due to the increasing volume of certificates that gets submitted to logs on a daily basis and the

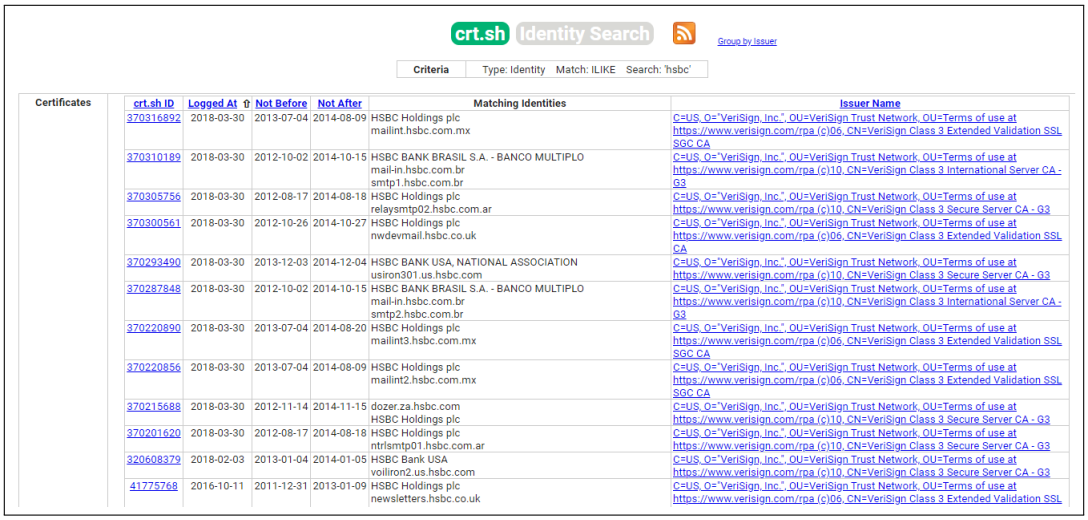


Figure 3.3: The result of a query when we search the domain name 'hsbc' on the certificate database website *crt.sh*. This is just one of many methods to inspect CT logs for any suspicious certificates. Accessed on 20^{th} January, 2020.

self-limitations monitors impose upon themselves e.g. limiting the results returned after each query. There is also the issue of having to place a lot of trust in a small set of well-resourced third parties which can inspect logs continuously; a proposed solution to this is called *verifiable light-weight monitoring*, developed by Dahlberg and Pulls [52].

Google Chrome Operations

Since January 2015, it has been a requirement that extended validation certificates be added to logs so that Internet connections can be accepted by the Google Chrome browser [169]; since April 2018, this requirement has been revised to include all newly issued TLS certificates [168]. Regardless of the method used to deliver SCTs, for a certificate to be accepted, it must be presented to clients with at least one SCT from a Google-operated log and another from a third-party log (Apple's CT policy states something similar [6]). Depending on the lifetime of the certificate, more SCTs from different logs may be required. Furthermore, newly created logs are not immediately

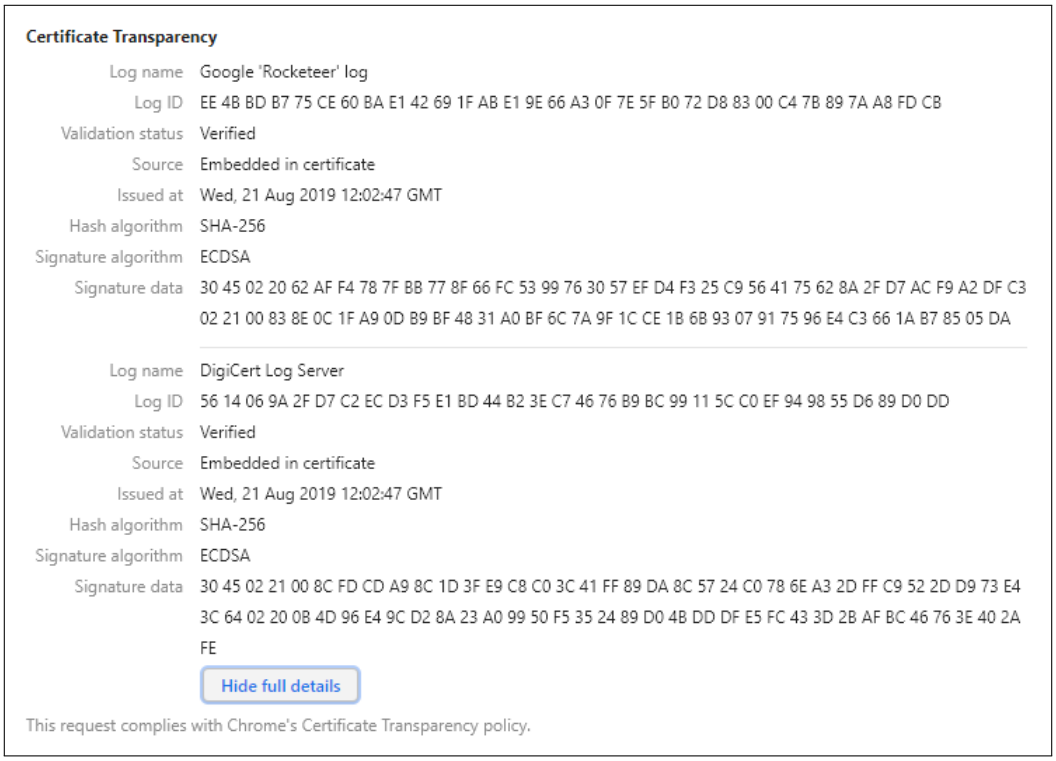


Figure 3.4: The CT information for a certificate when inspecting the index page of *hsbc.co.uk* via the Google Chrome browser. This certificate complies with Chrome's CT policy as it has been submitted to one Google-operated log and a third party log. The site was accessed on 20^{th} January, 2020.

seen as trustworthy by Google and must meet a strict set of criteria to be added to a list of trusted logs built into Chrome [125].

At the time of writing, there is no implementation in Chrome which asks for audit/consistency proofs when it sees new certificates or STHs due to privacy concerns: if a client contacts the log with an SCT, this risks leaking their browsing history which a log maintainer can exploit. One workaround called 'CT-over-DNS' uses special DNS records to obfuscate who is asking for proofs from the perspective of the log, with clients looking up CT information only through their existing DNS resolver. However, this is still in the experimental phase and carries some risks if deployed [194, 1, 157, 136].

3.2 CT Gossiping

This section describes the Chuat protocols [48], which makes a client gossip with servers after initialising encrypted connections with them and updates itself with log-based data, auditing SCTs if they have not seen them before. The purpose of the these protocols is to detect and report on an attack that involves hosting a fraudulent version of a log to victims who are being prevented from viewing the genuine one, thereby allowing bogus certificates to be accepted in the victims' TLS connections without question.

3.2.1 Split-world Attacks

In distributed systems, a *Byzantine fault* (inspired by the *Byzantine general's problem* [132]) occurs if independent processes fail to reach a consensus on the state of a system, resulting in service failure or network segmentation. As a consequence of this, different views of the network can be presented to different parties, causing confusion about which of these views is the correct one [3, 67, 9]. For example, in the case of cryptocurrencies such as bitcoin, it is vital for participating nodes in the bitcoin network to agree on a chronological list of approved transactions to prevent double spending from malicious nodes. One way that attackers can try and cause a Byzantine fault is to hijack the incoming and outgoing connections of nodes so that they only see attacker-controlled information (see *eclipse attacks* [192, 100, 149]).

In CT, a variation of a Byzantine fault can occur where the attacker attempts to host their fraudulent version of a log containing illegitimate certificates to a set of Internet users whose connections are being redirected so that they are made to contact it instead (e.g. via man-in-the-middle attacks). This attack we described is called a *split-world attack* [48, 153]. The attacker would have to be skilled and resourceful to carry out such

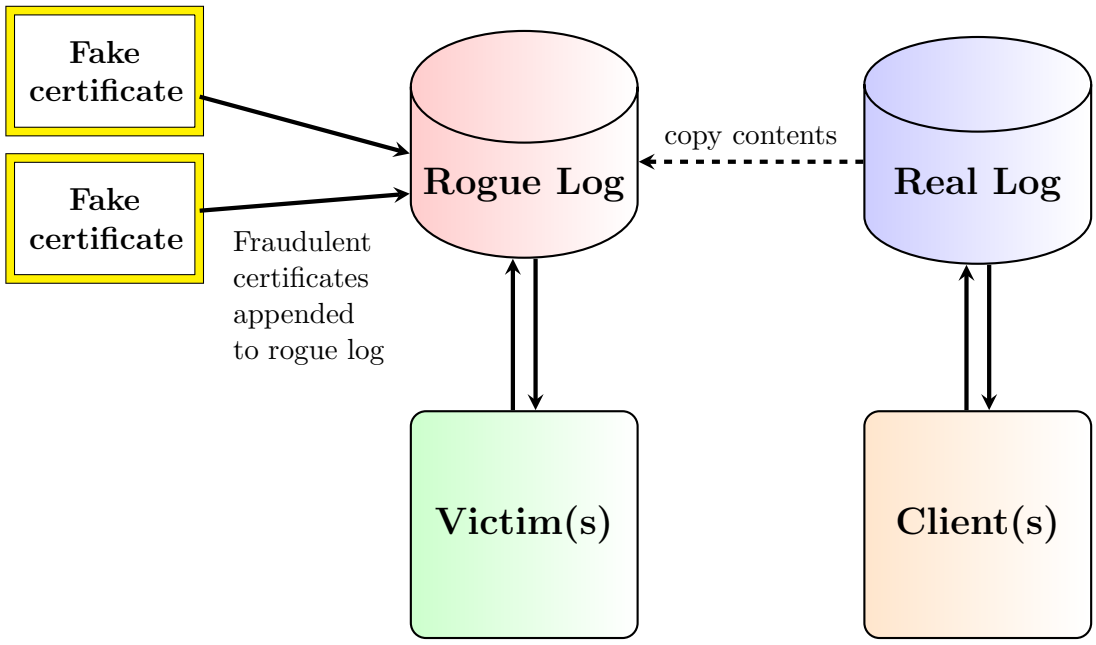


Figure 3.5: An Illustration of a split-world attack. The attacker copies the entire contents of a log and modifies it to include fraudulent certificates, presenting this rogue log to a set of isolated victims who have no means to verify consistency.

an attack for a prolonged period of time; potential candidates include an authoritarian government which is able to control its population's access to the Internet, or a log maintainer in collusion with criminal gangs operating online scams. Since users do not have a way to verify amongst themselves that their views of the log are identical, they can never be confident that the log maintainer is behaving honestly.

3.2.2 Chuat Protocols

In 2015, Chuat et al. described and tested a solution involving gossip protocols which required no extra architecture to implement on top of HTTPS and gossip messages were piggybacked on encrypted connections clients first make when they attempt to visit a website [48]. The protocols designed by Chuat et al. made clients and servers exchange log-sourced data, with servers acting as proxies for clients to exchange messages across different areas of the Internet.

We will briefly explain how these protocols work. Firstly, after a client connects to a HTTPS-enabled server and finishes the negotiation phase of the TLS handshake, it calls *getClientMessage()* to generate the gossip message m_1 which is piggy-backed on a HTTPS request (step (1) in Fig. 3.6). The server receives m_1 and checks to see if it is valid, meaning that all the STHs included in the message are correctly signed by a known log and contains the root hash of a Merkle tree with at least one certificate appended. Then, the server replies in kind by calling *getServerMessage()* to generate the gossip message m_2 which is sent back to the client using an HTTPS response (step (2) in Fig. 3.6). With log data from both parties now successfully exchanged, after performing the necessary consistency checks, both the client and server must refresh their knowledge of the log if the messages they retrieved have newer information. Since the gossiping was done via HTTPS, no-one can know whether any gossip happened without breaking the encryption used.

The two variants of this protocol presented by Chuat et al. are called *STH-only* gossiping, where both m_1 and m_2 consist of only one STH, and *STH-and-consistency-proof* gossiping, where messages contain a pair of STHs with a consistency proof between their respective tree sizes; for convenience, we will refer to the latter version as *STH-and-proof*. The idea behind gossiping proofs as well as messages is to reduce the amount of connections an entity needs to make to a log when requesting proofs as part of the protocol execution, thereby reducing the overhead of the log server to maintain operations. The updating procedure for the client is similar in both versions: using the server's message, it requests a consistency proof from the log whenever the tree size in m_2 is distinct from what the client already knows, updating itself when necessary using the contents of m_2 (step (3) in Fig. 3.6). Next, if the certificate's SCT has not been audited yet, then the client will request both inclusion and consistency proofs from the log to check if the certificate exists in

the log and finally updates its local state by retrieving the latest STH from the log.

Whenever a party receives an STH either through gossiping or by directly contacting the log, the protocol uses the `checkSTH()` method to check for consistency between multiple STHs when they are passed as parameters. To explain how this method works, suppose that we have two STHs sth_a and sth_b sourced from the same log, which are associated with tree sizes $t_a, t_b \in \mathbb{N}$. `checkSTH(`sth_a`,`sth_b`)` will verify two things:

- If $t_a = t_b$, check the root hashes of sth_a and sth_b are the same;
- If the timestamp of sth_a is older than the timestamp of sth_b, check that $t_a \leq t_b$.

If either of these conditions are not met, then the protocol treats this as a security incident and the log cannot be trusted. Otherwise, the protocol can be executed normally.

When a node does detect log inconsistency, either by `checkSTH()` failing or the log providing an erroneous proof upon request, the node will start to propagate a warning message which explains the issue discovered instead of STHs or proofs. A recipient of the warning message will pass it onto a log monitor for them to investigate further and determine if the log is conducting an attack.

The update procedures for the server in the two versions of the protocol work differently. In *STH-only* gossiping, the server will be required to only store one STH, whereas in *STH-and-proof* the server will record the messages it receives using a map and will choose to gossip one of them depending on the message it receives from a client.

To investigate how the gossip protocols would perform in a realistic setting, Chuat et al. collected traces of global Internet HTTP/HTTPS connections and the traffic data of different countries to model a global network. A small number of the clients used in their simulations use the gossiping software while all HTTPS servers hosting major

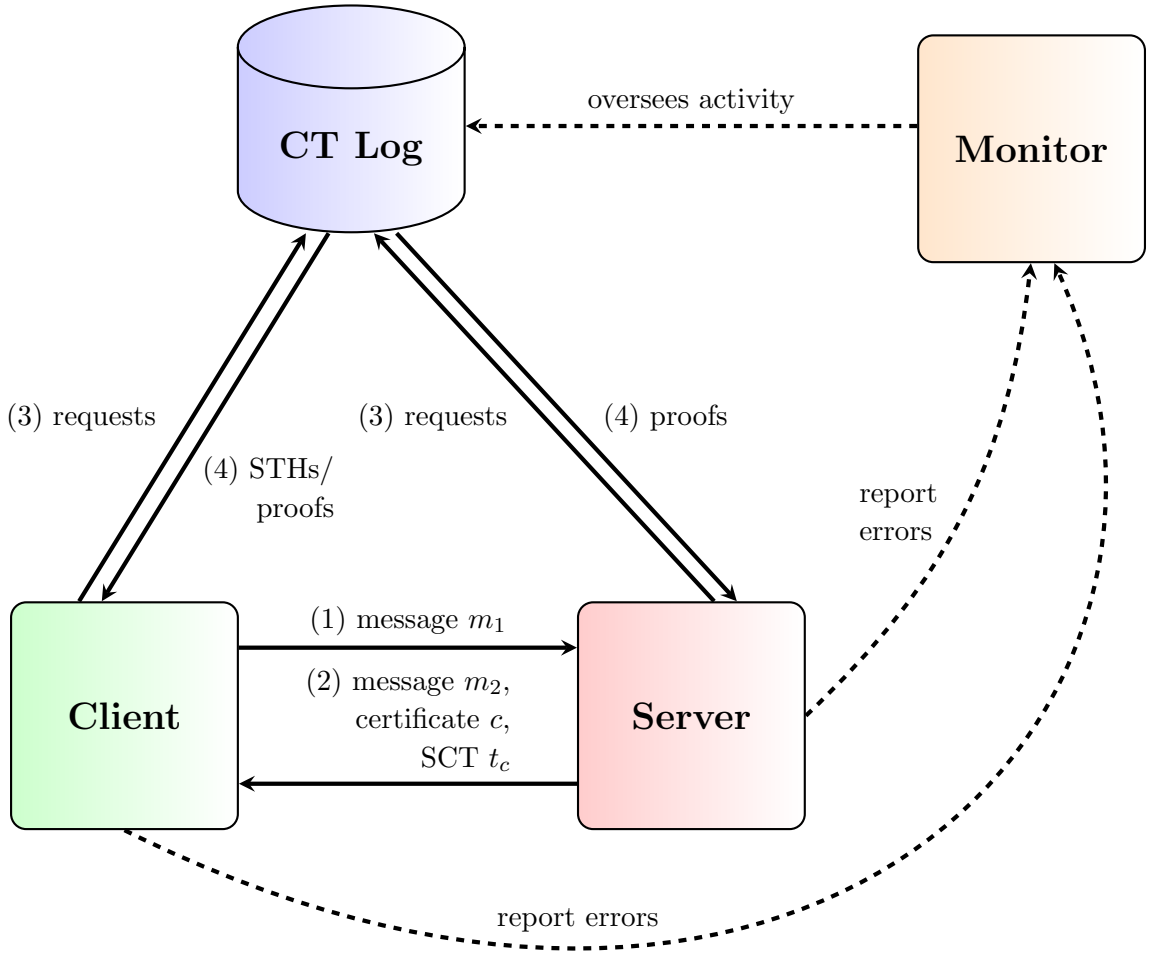


Figure 3.6: A basic illustration of the ChuatCT gossip protocols. First, a connected client and server exchange gossip messages using encrypted channels, with the server giving its SCT and the corresponding certificate to the client (steps (1) and (2)). Next, both entities update their states using the STHs received in the messages whilst checking that log consistency is not being violated (steps (3) and (4)). Anyone may request proofs or STHs from the log as part of the protocol execution. Clients may report to log monitors if they receive warnings from someone else stating that an anomaly has been found. Adapted from [48].

websites were assumed to be CT-enabled and had the ability to gossip, the reason for this decision was being that companies were more likely to adopt the latest security updates for their systems.

The investigations showed that the protocols were efficient in distributing log-based information among clients, greatly reducing the need for them to contact the log directly.

In particular, the use of gossiping consistency proofs with STHs was shown to be more scalable as it added little overhead (defined as the number of log connections strictly generated by the gossip protocol) compared to *STH-only* gossiping. Lastly, storing these gossip messages for both the client and server only needed several megabytes of data.

However, Chuat et al. acknowledged their investigations were not helpful in analysing how successful the protocols were in detecting attacks. In particular, if a split-world was taking place in the network, it is unclear whether detection could be achieved in a reasonable amount of time. Furthermore, the issue of privacy is still an issue whenever clients try to audit an SCT as it requires direct contact with the log.

3.3 Probabilistic Model Checking

Probabilistic model checking encompasses a broad set of formal techniques used to model and analyse phenomena which exhibit stochastic behaviour, such as communication protocols or biological processes. By mathematically describing every possible type of behaviour, model checking gives us a way to identify the possible configurations a system can be in and the random actions it may take.

In this section, to understand the techniques we shall use to analyse the performance of CT gossip protocols, we cover the basics of probabilistic model checking and discuss some advanced concepts involving uncertainty (when some of the transition probabilities are unknown) and unscalability (when the model becomes too large that model checking becomes computationally expensive). Lastly, we give a brief overview of the probabilistic model checker PRISM. Some of the content in this section is adapted from Baier and Katoen [13] and Forejt et al. [82].

3.3.1 Discrete Time Markov Chains

A discrete time Markov chain (DTMC) is a labelled transition system which encompasses a set of states, a probability transition function i.e. a real-valued function which determines if a transition has a positive probability or not, and a labelling function to indicate which atomic propositions are true for certain states. Furthermore, the set of outgoing transitions for each state is augmented with a probability distribution.

Definition 3.3.1. (DTMC) *A DTMC is a tuple* $\mathcal{M} = (S, S_i, P, \iota_{init}, AP, L)$*, where*

- S *is a countable set of states,*
- $S_i \subseteq S$ *is the set of initial states,*
- $P : S \times S \to [0, 1]$ *is the probability transition function such that, for each state* $s \in S$*,* $\sum_{s' \in S} P(s, s') = 1$*,*
- $\iota_{init} : S_i \to [0, 1]$ *is the initial distribution such that* $\sum_{s \in S_i} \iota_{init}(s) = 1$*,*
- AP *is a set of atomic propositions,*
- $L : S \to 2^{AP}$ *is a state labelling function.*

Fig. 3.7 gives an example of a DTMC which models a naive protocol design. We see, for example, $P(s_1, s_2) = P(s_1, s_3) = \frac{1}{2}$ and $L(s_1) = \{receive\}$. The state s_0 is the unique initial state in the model which we always start from i.e. $\iota_{init}(s_0) = 1$.

A *path* in a DTMC $\mathcal{M}$ is a (finite or infinite) sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 \in S_i$ and $P(s_i, s_{i+1}) > 0$ for every $i \geq 0$, representing a possible run of $\mathcal{M}$. We let $\pi(i)$ denote the $(i + 1)^{th}$ state in π. For a finite path $\pi_{fin} = s_0 s_1 \ldots s_n$, the

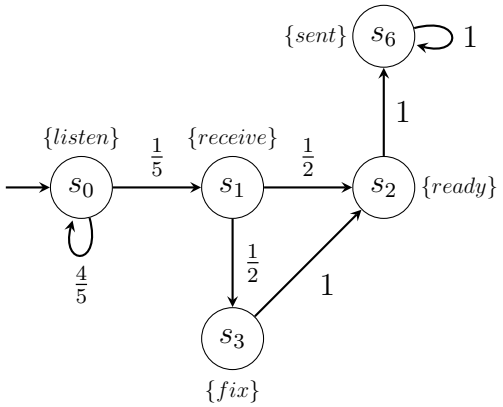


Figure 3.7: Example of a DTMC with labelling e.g. $L(s_2) = \{ready\}$.

probability of this path in $\mathcal{M}$ is given by:

$$Prob_{\mathcal{M}}(\pi_{fin}) = \iota_{init}(s_0) \prod_{i=0}^{n-1} P(s_i, s_{i+1}). \tag{3.2}$$

Going back to Fig. 3.7, a possible finite path in the model is $s_0 s_0 s_1 s_3 s_2 s_6$ which occurs with probability $\frac{4}{5} \cdot \frac{1}{5} \cdot \frac{1}{2} = \frac{2}{25}$. Unless otherwise stated, we assume all paths in $\mathcal{M}$ are infinite.

For a (finite or infinite) path π in $\mathcal{M}$, the *trace* of π, or $tr(\pi)$, is defined as the sequence of state labellings when taking π. For example, going back to our finite path $s_0 s_0 s_1 s_3 s_2 s_6$, its trace is equal to:

$$tr(s_0 s_0 s_1 s_3 s_2 s_6) = \{request\}\{request\}\{receive\}\{fix\}\{ready\}\{sent\}.$$

For a state s, we let $FPath_{\mathcal{M},s}$ and $IPath_{\mathcal{M},s}$ denote the set of finite and infinite paths in $\mathcal{M}$ starting from state s, respectively. The set of all finite and infinite paths

in $\mathcal{M}$, $FPath_{\mathcal{M}}$ and $IPath_{\mathcal{M}}$, are defined as:

$$FPath_{\mathcal{M}} = \bigcup_{s \in S} FPath_{\mathcal{M},s},$$
$$IPath_{\mathcal{M}} = \bigcup_{s \in S} IPath_{\mathcal{M},s}.$$

To reason about the likelihood of traversing certain paths in the model, a probability measure needs to be constructed over the paths in $\mathcal{M}$. For a finite path $\pi_{fin} = s_0 s_1 \dots s_n$ in $\mathcal{M}$, the cylinder set $Cyl(\pi_{fin})$ is the set of all infinite paths in $\mathcal{M}$ prefixed by π_{fin}:

$$Cyl(\pi_{fin}) = \{\pi \in IPath_{\mathcal{M}} \mid \pi_{fin} \text{ prefixes } \pi\}.$$

Using measure theory [7], it can be shown that for the smallest set of cylinder sets $Cyl(\pi_{fin})$ where π_{fin} ranges over all elements in $FPath_{\mathcal{M}}$, a unique probability measure $Prob_{\mathcal{M}}$ exists such that the probability for a cylinder set $Cyl(s_0 s_1 \dots s_n)$ is given by (3.2) (see Kemeny et al. for details [121]). For path fragments of length zero, we let the probability of such paths be equal to one. As path probabilities are well-defined, an important consequence of this fact is that we are able to reason about the probabilistic aspects of DTMCs such as the reachability of a set of states (see subsection 3.3.4 for details).

3.3.2 Markov Decision Processes

An extension of DTMCs are Markov decision processes (MDPs) which combine non-determinism and probabilistic choice. MDPs are useful in modelling choices over a set of actions, allowing us to determine which sets of choices produce best- and worst-outcomes. They are also useful in modelling a system if we have parts of it that are underspecified or too complex to model stochastically.

Definition 3.3.2. (MDP) *An MDP is a tuple* $\mathcal{M}_{MDP} = (S, S_i, \iota_{init}, Act, P, AP, L)$, *where*

- S, S_i, ι_{init}, AP *and* L *are the same as in Definition 3.3.1.*
- Act *is a set of actions,*
- $P : S \times Act \times S \to [0, 1]$ *is a probability transition function such for every state* s *and every action* α, *either* $\sum_{s' \in S} P(s, \alpha, s') = 1$ *or* $\sum_{s' \in S} P(s, \alpha, s') = 0$.

DTMCs are a special case of MDPs with only one action. From a state $s \in S$, transitions in the MDP occur by first choosing an action $\alpha \in Act$ non-deterministically (we assume that the set of actions for each state is non-empty) and then randomly choosing a successor state using the probability distribution given associated with α. Paths in an MDP are written as $\pi = s_0\alpha_0 s_1\alpha_1 s_2 \ldots$, where $s_0 \in S_i$, α_i is an action available in state s_i and $P(s_i, \alpha_i, s_{i+1}) > 0$ for every $i \geq 0$, with $\pi(i)$ indicating the $(i+1)^{th}$ state in π. The subsequent definitions for $FPath_{\mathcal{M},s}$, $IPath_{\mathcal{M},s}$, $FPath_{\mathcal{M}}$ and $IPath_{\mathcal{M}}$ easily carry over from Section 3.3.1.

To resolve the non-determinism in MDPs, functions called *adversaries* (or *policies*) can induce a DTMC depending on the path taken through the MDP. The main idea is that an adversary chooses from the list of permitted actions available at the last state of each finite path.

Definition 3.3.3. (Adversaries) *Given an MDP* $\mathcal{M} = (S, S_i, Act, P, \iota_{init}, AP, L)$, *an* adversary *is a function* $\sigma : FPath_{\mathcal{M}} \to Dist(Act)$, $Dist(Act)$ *is the set of probability distributions on* Act, *such that the probability* $\sigma(\pi)(\alpha) > 0$ *only if* $\alpha \in Act$ *is an allowed action in the last state of path* π.

We let $Adv_{\mathcal{M}}$ be the set of all adversaries for $\mathcal{M}$, or just Adv if $\mathcal{M}$ is clear from the context. We call an adversary *memoryless* if the probability distribution $\sigma(\pi)$ *only*

depends on the last state in π, not on π itself. In other words, if we have two different finite paths π and π' in $\mathcal{M}$ where their last states are identical, then $\sigma(\pi) = \sigma(\pi')$. When traversing MDPs under an adversary σ, this induces a DTMC $\mathcal{M}_\sigma$, with each state representing a finite path taken in the MDP.

3.3.3 Probabilistic Computational Tree Logic (PCTL)

The objective of model checking is to determine whether our models meet specified properties when starting from certain states. For example, we may like to know whether a property has a high probability of being true, or it will be true after taking a certain number of steps in the model. To do this, we need to be able to express these properties using a form of temporal logic which is either true or false for a state in a Markov model. For this thesis, we will use probabilistic computational tree logic (PCTL) [99]:

Definition 3.3.4. (PCTL syntax) *The syntax of* probabilistic computational tree logic *(PCTL) is as follows:*

$$\phi ::= \texttt{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{P}_{\bowtie p}[\Phi]$$
$$\Phi ::= \texttt{X}\,\phi \mid \phi_1\,\texttt{U}^{\leq l}\,\phi_2 \mid \phi_1\,\texttt{U}\,\phi_2$$

Here, a is an atomic proposition, ϕ is a state formula, Φ is a path formula, $p \in [0,1]$, $l \in \mathbb{N}$ and $\bowtie \in \{<, \leq, >, \geq\}$

The *probabilistic path operator* $\mathbf{P}_{\bowtie p}[\Phi]$ is read as follows: "the probability of satisfying the path formula Φ is within in the range specified by $\bowtie$ p". $\texttt{X}$ means "next" and therefore $\texttt{X}\,\phi$ is read as "ϕ will be true in the next state". For state formulae ϕ_1 and ϕ_2, $\phi_1\,\texttt{U}^{\leq l}\,\phi_2$ means "ϕ_2 is true within l steps in the model and ϕ_1 holds true up until that point". $\phi_1\,\texttt{U}\,\phi_2$ has a similar meaning but with ϕ_2 being satisfied sometime in the future.

Given a set of adversaries Adv for a model, a state s and PCTL formula ϕ, we say that "s satisfies ϕ", or $s \models_{Adv} \phi$, if ϕ is satisfied in s under all adversaries in Adv. In the sequel, let $Prob^{\sigma}_{\mathcal{M},s}(E)$ be the probability of event E occurring from state s in Markov model $\mathcal{M}$ under an adversary σ.

Definition 3.3.5. (PCTL semantics) *Let $\mathcal{M} = (S, S_i, Act, P, \iota_{init}, AP, L)$ be an MDP, Adv a set of adversaries for $\mathcal{M}$, $s \in S$ and $\pi \in IPath_{\mathcal{M}}$. The satisfaction relation $\models_{Adv}$ of PCTL is defined inductively by:*

- $s \models_{Adv} True$ *is always true,*
- $s \models_{Adv} a \Leftrightarrow a \in L(s)$,
- $s \models_{Adv} \phi_1 \wedge \phi_2 \Leftrightarrow s \models_{Adv} \phi_1$ and $s \models_{Adv} \phi_2$,
- $s \models_{Adv} \neg\phi \Leftrightarrow s \not\models_{Adv} \phi$,
- $s \models_{Adv} \mathbf{P}_{\bowtie p}[\Phi] \Leftrightarrow Prob^{\sigma}_{\mathcal{M},s}(\{\pi' \in IPath_{\mathcal{M},s} \mid \pi' \models_{Adv} \Phi\}) \bowtie p$, $\forall \sigma \in Adv$,
- $\pi \models_{Adv} \mathtt{X}\,\phi \Leftrightarrow \pi(1) \models_{Adv} \phi$,
- $\pi \models_{Adv} \phi_1\, \mathtt{U}^{\leq l}\, \phi_2 \Leftrightarrow \exists i \leq l$ *s.t.* $\pi(i) \models_{Adv} \phi_2$ *and* $\pi(j) \models_{Adv} \phi_1$, $\forall j < i$,
- $\pi \models_{Adv} \phi_1\, \mathtt{U}\, \phi_2 \Leftrightarrow \exists l \geq 0$ *s.t.* $\pi \models_{Adv} \phi_1\, \mathtt{U}^{\leq l}\, \phi_2$.

We can define something similar in the case of DTMCs; note that the set of adversaries for such models has a cardinality of one. If Adv is clear from the context, we shall simply write $\models$ instead of $\models_{Adv}$.

There are several other useful operators which can be derived from the syntax in Definition 3.3.4 but for our purposes we give only one example. For a state PCTL property ϕ, the *future operator* $\mathbf{F}\,\phi$, sometimes written as $\Diamond\phi$, means that ϕ is eventually

satisfied and its bounded variant $\mathbf{F}^{\leq l}\phi$ means that ϕ will be true within $l \geq 0$ steps. It is easy to show the following equivalences:

$$\mathbf{F}^{\leq l}\,\phi \equiv \mathtt{true}\,\mathtt{U}^{\leq l}\,\phi,$$

$$\mathbf{F}\,\phi \equiv \mathtt{true}\,\mathtt{U}\,\phi.$$

3.3.4 PCTL Model Checking

Since this is outside the scope of this thesis, we only briefly describe the model checking problem and the general mechanics used to solve it. Given a model $\mathcal{M}$ with state space S, a set of adversaries Adv and a PCTL formula ϕ, the set $Sat_{\mathcal{M}}(\phi) \subseteq S$ is defined to be the set of states in $\mathcal{M}$ which satisfies ϕ, in other words, $Sat_{\mathcal{M}}(\phi) = \{s \in S \mid s \models_{Adv} \phi\}$. The main problem of model checking is to determine if $S_i \subseteq Sat(\phi)$. If this is the case, we say that "$\mathcal{M}$ satisfies ϕ", written as $\mathcal{M} \models \phi$.

Verifying PCTL formulae is usually solved by using backward breadth-first search algorithms [27, 55] on the underlying graph of the Markov model [50]. To solve for probabilistic-type properties, this can be reduced to solving the *probabilistic reachability problem*, which computes the probability of reaching a set of target states from an initial state, solved using a system of linear equations. Model checking software uses efficient, iterative methods to find such probabilities to a suitable degree of accuracy; see the *Jacobi/Gauss-Seidel method* [200]. In the case of MDPs, *value iteration* [178, 11] can be applied to maximise or minimise a particular sum by finding adversaries that assign as much or as little probability to certain states (see the *Bellman equations* [21]).

3.3.5 Reward Structures and Reward-based Properties

In addition to PCTL, *reward structures* on a model allow us to reason about other quantitative properties we may be interested in, for example the number of times a

certain process is executed before a program terminates or the amount of power used by a device to execute an instruction. This involves augmenting the model so that the states or transitions are assigned a value which are incremented when a path goes through them.

Definition 3.3.6. (reward structures) *A* reward structure *for a DTMC* $\mathcal{M} = (S, S_i, P, \iota_{init}, AP, L)$ *is a pair of functions* $r = (r_{state}, r_{action})$*, where* $r_{state} : S \to \mathbb{R}_{\geq 0}$ *is a state reward function and* $r_{action} : S \times S \to \mathbb{R}_{\geq 0}$ *is an action reward function.*

We shall focus on three types of rewards for this thesis: *instantaneous*, *cumulative* and *reachability* rewards.

Instantaneous Rewards

Instantaneous rewards measure the state reward after taking exactly $k \in \mathbb{N}$ steps in the model. Given a path $\pi \in IPath_{\mathcal{M}}$, the instantaneous reward function $\mathbf{I}^{=k} : IPath_{\mathcal{M}} \to \mathbb{R}_{\geq 0}$ is defined as $\mathbf{I}^{=k}(\pi) = r_{state}(\pi(k))$. Calculating the expected value can easily be done using repeated matrix multiplication.

Cumulative Rewards

The cumulative reward measures the total reward accumulated along a path of length $k \in \mathbb{N}$ and thus considers both state and transition rewards. For $k \in \mathbb{N}$ and a Markov model $\mathcal{M}$, the cumulative reward function $\mathbf{C}^{\leq k} : IPath_{\mathcal{M}} \to \mathbb{R}_{\geq 0}$ is defined as the total amount of state and transition reward accumulated along a path π of length k. Like with instantaneous rewards, the expected cumulative reward can be computed iteratively.

Reachability Rewards

A special class of cumulative reward involves finding the reward until a set of target states is reached in the model. Given a Markov model $\mathcal{M}$, a target set T, the reachability reward function $R^T : IPath_{\mathcal{M}} \rightarrow \mathbb{R}_{\geq 0}$ is defined similarly to the function $\mathbf{C}^{\leq k}$ as seen previously but with "k" in the summations replaced with "k^*", where $k^* = \min\{k \mid \pi(k) \in T\}$. Computing the expected reachability award can be done by solving a system of linear equations.

Extending PCTL with Rewards

We extend our PCTL syntax given in Definition 3.3.1 to include reward-based properties based on the three types we have already covered [82]. Letting r be a reward structure and ϕ_r be a reward-based state formula, the syntax is defined as follows:

$$\phi_r ::= \mathbf{R}^r_{\bowtie x}[\mathbf{I}^{=k}] \mid \mathbf{R}^r_{\bowtie x}[\mathbf{C}^{\leq k}] \mid \mathbf{R}^r_{\bowtie x}[\mathbf{F}\,\phi] \tag{3.3}$$

Here, $x \in \mathbb{R}_{\geq 0}$, $k \in \mathbb{N}$, ϕ is a PCTL expression and $\bowtie \in \{<, \leq, >, \geq\}$. Next, we give the semantics for each operator in (3.3) using the notation from Definition 3.3.5. For a reward function f_r, let $\mathbb{E}^{\sigma}_{\mathcal{M},s}(f_r)$ be the expected value of f_r from state s in Markov model $\mathcal{M}$ under an adversary σ. Then:

- $s \models_{Adv} \mathbf{R}^r_{\bowtie x}[\mathbf{I}^{=k}] \Leftrightarrow \mathbb{E}^{\sigma}_{\mathcal{M},s}(\mathbf{I}^{=k}) \bowtie x$, for every $\sigma \in Adv$,
- $s \models_{Adv} \mathbf{R}^r_{\bowtie x}[\mathbf{C}^{\leq k}] \Leftrightarrow \mathbb{E}^{\sigma}_{\mathcal{M},s}(\mathbf{C}^{\leq k}) \bowtie x$, for every $\sigma \in Adv$,
- $s \models_{Adv} \mathbf{R}^r_{\bowtie x}[\mathbf{F}\,\phi] \Leftrightarrow \mathbb{E}^{\sigma}_{\mathcal{M},s}(R^{Sat_{\mathcal{M}}(\phi)}) \bowtie x$, for every $\sigma \in Adv$.

Here, $Sat_{\mathcal{M}}(\phi)$ is defined as in Section 3.3.4. Examples of such reward-based properties include:

- $\mathbf{R}^{queue}_{>4}[\mathbf{I}^{=t}]$: the expected size of a queue after exactly t steps is greater than four (instantaneous reward).

- $\mathbf{R}^{power}_{\leq 3}[\mathbf{C}^{\leq t}]$: the expected cumulative amount of power used within t steps is at most three units (cumulative reward).

- $\mathbf{R}^{requests}_{<5}[\mathbf{F}\ \texttt{end}]$: the expected number of requests before a protocol terminates is less than five (reachability reward).

- $\mathbf{R}^{energy}_{=?}[\mathbf{F}\ \texttt{target}]$: the expected amount of energy used before an automated system reaches the target destination (reachability reward).

The first three properties given above are examples of *qualitative-based* properties which evaluate to either true or false; the last example is a *quantitative-based* property which evaluates to a real value.

3.3.6 Interval Discrete Time Markov Chains

When modelling processes which exhibit fluctuating behaviour, it is hard to capture exact transition probabilities simply because we lack experimental evidence. One way to overcome this is to use bounded intervals to show the range of possible probabilities from a given state. The resulting model takes the form of an *interval discrete Markov chain* (IDTMC) [190, 117, 127]. In the sequel, we let $P^u, P^l : S \times S \to [0, 1]$ be functions which give the maximal and minimal probability of transitioning between two states respectively.

Definition 3.3.7. (IDTMCs) *An IDTMC is a tuple* $\mathcal{M}_I = (S, s_\iota, P^u, P^l, AP, L)$*, where*

- S, AP *and* L *are the same as in Definition 3.3.1,*

- $s_\iota \in S$ *is the initial state in the model,*

- $P^u, P^l : S \times S \to [0,1]$ *are functions such that, for every* $s \in S$,

$$\sum_{s' \in S} P^l(s,s') \leq 1 \leq \sum_{s' \in S} P^u(s,s') \text{ and } P^l(s,s') \leq P^u(s,s'), \text{ for all } s' \in S. \quad (3.4)$$

The interval $[P^l(s,s'), P^u(s,s')]$ denotes the range of possible probabilities of transitioning from s to s', allowing for a possibly infinite set of probability distributions from s. We need the condition stated in (3.4) for probability distributions to exist given any state in $\mathcal{M}_I$.

We can interpret an IDTMC is in two different ways. The first one is an *uncertain Markov chain* (UMC), where the IDTMC $\mathcal{M}_I$ represents an infinite set of DTMCs $[\mathcal{M}_I]$, where for each DTMC $\mathcal{M} \in [\mathcal{M}_I]$, the following is true:

$$P^l(s,s') \leq P(s,s') \leq P^u(s,s'), \text{ for every } (s,s') \in S \times S.$$

A DTMC is non-deterministically chosen from $[\mathcal{M}_I]$, determining all the transitions and their respective probabilities for the chosen DTMC.

The second interpretation is that an IDTMC is an MDP with an infinite number of actions, where the environment non-deterministically chooses a probability distribution at *each state*. For a state s, we define $Dist_s$ to be the set of all probability distributions which can be chosen from s:

$$Dist_s = \left\{ P_s : S \to [0,1] \;\middle|\; \sum_{s' \in S} P_s(s') = 1 \text{ and } P^l(s,s') \leq P_s(s') \leq P^u(s,s') \right\}.$$

Like with regular MDPs, we can similarly define paths as an alternating sequence of states and probability distributions, and adversaries which choose probability distributions in $Dist_s$ to induce a DTMC.

Regarding PCTL semantics, given a UMC $\mathcal{M}_I$ and a PCTL formula ϕ, we say

$\mathcal{M}_I \models \phi$ if and only if $\mathcal{M} \models \phi$ for every $\mathcal{M} \in [\mathcal{M}_I]$. In the case of the MDP interpretation, the PCTL semantics are the same as in Definition 3.3.5.

Care must be taken when using either the UMC or MDP interpretations for IDTMC model checking as they can give completely different results. For example, there are cases of IDTMCs where for a given PCTL formula ϕ, the UMC form satisfies ϕ but the MDP form does not; see Sen et al. for details [190]. For this thesis, we use the MDP interpretation as it allows us to adapt well-known algorithms used for MDP model checking which reduces the complexity of the probabilistic reachability problem (see Section 3.3.8 for details).

3.3.7 Abstract Markov Chains

A special class of IDTMCs involve collapsing larger Markov models by partitioning the states of the original model which we call the *concrete states*, giving each subset a representative state called the *abstract state*, with transitions between such states describing the minimal and maximal probabilities of moving from one state partition to another [80].

Definition 3.3.8. (ADTMCs) *An* Abstract discrete time Markov chain *(ADTMC) is a tuple* $\mathcal{M}_A = (S, s_\iota, P^u, P^l, AP, L)$*, where:*

- $S, s_\iota, \iota_{init}, AP, P^l, P^u$ *are the same as in Definition 3.3.7,*
- $L : S \times AP \to \{\top, \bot, ?\}$ *is a labelling function.*

In *three-valued abstraction* techniques [119, 108], L is a labelling function which determines the truthfulness of an atomic proposition in an abstract state depending on the states from the original model it represents; for a given atomic proposition $a \in AP$ and abstract state $s \in S$, $L(s, a)$ evaluates to either true ($\top$) if a is true in every state represented by s, false ($\bot$) if a is false every state represented by s, or indefinite (?)

otherwise. In this thesis, we assume that our ADTMC models are unlabelled as we are not concerned with determining if a qualitative PCTL property evaluates to either true or false, thereby preserving the truthfulness of that same property in the original concrete model.

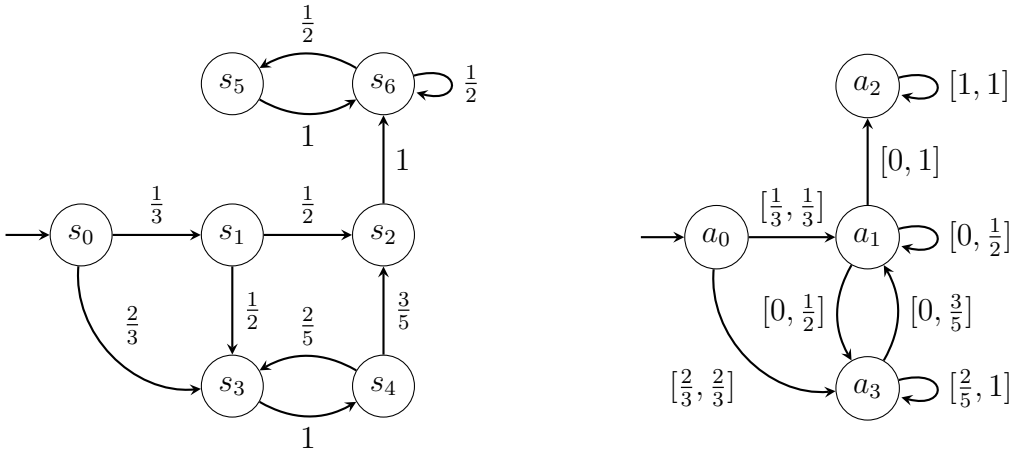
Fig. 3.8 shows how models can be abstracted using a small DTMC. We have (a) an unlabelled concrete model with seven states which we can transform into (b) an ADTMC by partitioning the state space as follows: $a_0 = \{s_0\}, a_1 = \{s_1, s_2\}, a_2 = \{s_5, s_6\}, a_3 = \{s_3, s_4\}$. To find bounds $P^l(a_1, a_3)$ and $P^u(a_1, a_3)$, for example, we observe the following:

$$P^l(a_1, a_3) = \min(P(s_1, s_3), P(s_1, s_4), P(s_2, s_3), P(s_2, s_4)) = 0,$$
$$P^u(a_1, a_3) = \max(P(s_1, s_3), P(s_1, s_4), P(s_2, s_3), P(s_2, s_4)) = \tfrac{1}{2}$$

It is also possible to abstract IDTMCs in a similar fashion. For an IDTMC $\mathcal{M}_I = (S, s_\iota, P^u, P^l, AP, L)$ and a finite partitioning $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ of S, an abstraction of $\mathcal{M}_I$ with respect to $\mathcal{A}$ is the ADTMC $\mathcal{M}_I^{\mathcal{A}} = (\mathcal{A}, A_\iota, \widetilde{P}^u, \widetilde{P}^l, AP, \widetilde{L})$, where:

- For any $A_i, A_j \in \mathcal{A}$, we have $\widetilde{P}^l(A_i, A_j) = \inf_{s \in A_i} P^l(s, A_j)$ and $\widetilde{P}^u(A_i, A_j) = \min\{1, \sup_{s \in A_i} P^u(s, A_j)\}$.
- $A_\iota \in \mathcal{A}$ is the abstract state such that $s_\iota \in A_\iota$,
- $L : S \times AP \to \{\top, \bot, ?\}$ is labelling function as seen in Definition 3.3.8.

When abstracting IDTMCs, a caveat is that it will make our probability intervals less meaningful because, for some $s, s' \in S$, there may exist a value $p \in [P^l(s, s'), P^u(s, s')]$ such that there is no probability distribution μ_s with $\mu_s(s') = p$; if this is not the case, then we say that our model is *delimited*. Abstracting a DTMC model always yields a delimited ADTMC. To obtain delimited models, we can perform a special cut on the probability intervals which shrinks them appropriately [119]:



(a) concrete model (b) abstract model

Figure 3.8: An example abstraction of a concrete DTMC (a) into an ADTMC (b). The abstract states represent the following subsets of the concrete state space: $a_0 = \{s_0\}, a_1 = \{s_1, s_2\}, a_2 = \{s_5, s_6\}, a_3 = \{s_3, s_4\}$.

Definition 3.3.9. (Normalisation) *For an IDTMC $\mathcal{M} = (S, s_\iota, P^l, P^u, AP, L)$, the normalisation of $\mathcal{M}_A$ yields an IDTMC $\widetilde{\mathcal{M}} = (S, s_\iota, \widetilde{P}^u, \widetilde{P}^l, AP, L)$, where for all $s, s' \in S$:*

$$\widetilde{P}^l(s, s') = \max\left(P^l(s, s'), 1 - P^u(s, S\backslash\{s'\})\right),$$
$$\widetilde{P}^u(s, s') = \min\left(P^u(s, s'), 1 - P^l(s, S\backslash\{s'\})\right).$$

It can be shown that normalisation will always yield a delimited model and only needs to be applied once. In Chapter 6, we adapt Definition 3.3.9 to derive probability distributions given a set of intervals.

3.3.8 Model Checking IDTMCs

Seminal work on model checking IDTMCs has been done by Sen et al. [190, 45] who describe a set of techniques to solve both the MDP and UMC semantics, showing their respective computational complexity. They conclude that model checking UMCs are less efficient to analyse in practice since the algorithms are related to checking the

feasibility of bilinear matrix inequalities which is known to be NP-hard [209]. However, under the MDP semantics, it is more efficient to reduce the problem of model checking IDTMCs to model checking MDPs whose state space size is exponential in the state space size of the former. Chen et al. [46] extends on the work of Sen et al. by showing the reachability problem (i.e. "from a given state, what is the probability of reaching a set of target states?") coincides for both the UMC and MDP semantics whilst improving the complexity bounds of model checking IDTMCs, proving that PCTL model checking IDTMCs under the MDP semantics is P-complete.

Benedikt et al. [23] also present algorithms for model checking interval Markov chains with respect to basic linear-time properties. In the context of ADTMCs, Katoen et al. [119] describe in detail the notion of *extreme schedulers* which choose a class of distributions where each probability is either maximised or minimised while respecting the interval bounds, called *extreme distributions*. They show that, since ADTMCs mimic the step-wise behaviour of their concrete counterparts (either a DTMC or an IDTMC), extreme schedulers are sufficient to find under- and over-approximations of satisfying PCTL formulae for abstraction-based model checking. Later on in Chapter 5, when we describe the probability intervals we use for our IDTMCs, we assume that they are closed; Sproston, Chakraborty and Katoen [44, 196] show how open interval sets can also be accommodated into models.

Model checking IDTMCs as MDPs can involve computing an infeasible amount of extreme distributions at each state. In a similar fashion to Sen et al., to efficiently compute the minimal (maximal) reachability probabilities over these potentially large MDPs, a variant of value iteration is applied to efficiently compute extreme distributions which minimise (maximise) a certain sum, which avoids having to find the set of extreme distributions at every state [80].

3.3.9 PRISM

PRISM [129] is an open-source probabilistic model checker which specialises in modelling and verifying systems exhibiting random behaviour. Using the PRISM modelling language, it allows us to formally describe models with probabilistic, non-deterministic or real-time characteristics at a high level. Properties of interest can be evaluated with a wide range of temporal logics including PCTL and its reward-based extension. By having the ability to randomly sample thousands of paths in models, the discrete-event simulator engine also supports statistical model checking methods which can approximate quantitative properties if they are infeasible to derive exactly. The version of PRISM used for this thesis is v4.6. We give a brief overview of basic PRISM model design and some of its key functionalities we will use in this thesis; see the PRISM website for more details [177].

Basic PRISM Syntax

Describing the details of a PRISM model using the PRISM language is a straightforward process. For each independent process in the model, we can encapsulate their behaviour using a *module*. A module M is made up of a set of *variables* $V = \{v_1, v_2, \ldots, v_k\}$ describing the possible states the module can be in, with each v_i defined by a range of values $R(v_i)$, where either $R(v_i) = \{\bot, \top\}$ or $R(v_i) \subset \mathbb{Z}$, and a set of *commands* which describe the behaviour of the module depending on the current state of its variables. Generally speaking, a command is usually of the form:

$$[action]\ \mathcal{G}\ \rightarrow\ p_1 : u_1 + \ldots + p_n : u_n; \tag{3.5}$$

In line (3.5), *action* is the action label for the command which is commonly used to synchronise the behaviour with other commands in the model (otherwise, we just use

[ ]), $\mathcal{G}$ is the guard which must evaluate to true for the command to be executed and p_i is the probability of the corresponding variable update function $u_i : R(v_1) \times \ldots \times R(v_k) \rightarrow R(v_1) \times \ldots \times R(v_k)$ occurring, where $\sum_{i=1}^{n} p_i = 1$. Note that variable updates can only change variables defined within the module it resides in and it is possible for the guards of two distinct commands to be true at the same time; in the case of DTMCs, it is randomly decided which command get executed. An example of a command is:

$$[\,]\ (s = 1 \wedge t > 1)\ \rightarrow\ \tfrac{1}{2} : (s' = 2) + \tfrac{1}{2} : (s' = 3) \wedge (t' = 0);$$

where s and t are integer variables (we use the s' and t' notation to indicate the new value of s and t respectively when the particular update function is applied). Additionally, we can define reward structures for our models using separate constructs and assigning different values to states or transitions if they satisfy certain conditions.

Symmetry Reduction

Briefly speaking, given two Markov models $\mathcal{M}$ and $\mathcal{M}'$, we say that $\mathcal{M}$ *probabilistically bisimulates* $\mathcal{M}'$ if a relation can be defined between the sets of states for each model, showing that $\mathcal{M}$ can mimic the step-wise behaviour of the $\mathcal{M}'$ in a way that they are indistinguishable [134]. Probabilistic bisimulation is useful when we want to find more compact, smaller versions of large models where model checking may be too expensive to perform, preserving both quantitative- and qualitative-based properties under certain conditions [10, 189].

To automate the process of finding and working with bisimulations of PRISM models with large state spaces, PRISM allows us to compress the model information during the building process using *symmetry reduction* [131, 64, 65], producing an equivalent model which retains the same properties of the original model but uses significantly

fewer states and thus takes less time to perform model checking on. We consider the case of component symmetry, where for N modules $m_1, m_2, \ldots, m_N$ in the model, their behaviour is indistinguishable and any pair can be permuted without affecting the whole model, allowing us to uniquely represent all N of these modules with a single meta-module. While symmetry reduction is a powerful technique, it is not applicable in most circumstances where there are many processes behaving in different ways and compacting the model in this way does produce an overhead during the building process.

Statistical Model Checking

If the models are too large, then it is sometimes infeasible for PRISM to compute exact values for quantitative properties. A low-cost approach to overcome this problem is to instead approximate the results by generating a large number of sample paths through the model and evaluating a specified property for each of them. This technique is often called *statistical model checking* [191, 165].

Using the discrete-event simulator, PRISM has multiple ways to perform statistical model checking depending on the property we want to analyse:

- Confidence Interval (CI),
- Asymptotic Confidence Interval (ACI),
- Approximate probabilistic Model Checking (APMC),
- Sequential probability Ratio Test (SPRT).

For this thesis, we focus only on the CI method due to being applicable for the formal specifications we want to measure. Given a confidence level γ, it produces an estimation of a quantitative property y and an interval $[y-w, y+w]$ where the true value will lie with probability γ, w being the half-width of the interval which is calculated

using the desired confidence level and the number of path samples we want to produce. The default confidence level used in PRISM is 99%. At the time of writing, PRISM does not support statistical model checking with multiple initial states. Furthermore, the CI method is not suited to the use of bounded properties (i.e. of the form $\mathbf{P}_{\bowtie p}[\ldots]$ or $\mathbf{R}_{\bowtie r}[\ldots]$); the SPRT method is more appropriate to use in such cases.

3.4 Derivative-free Optimisation

In this section, we begin describing a technique that, given a function which is expensive to compute, iterates between the construction of a series of models and the choosing of an input for the function to best optimise it. Afterwards, we list a couple of Python libraries which implement a variation of this technique and briefly demonstrate their usage. In Chapter 6, we will also use these libraries to search over a space of probabilistic models to minimise particular protocol properties.

3.4.1 Sequential Model-based Optimisation

Suppose that we want to optimise a particular function (or an algorithm) which requires a long period of time to produce an output and is often erratic, thus making it hard to discern any patterns of behaviour. To add another layer of complexity to the issue, the analytical form of this function is unknown, meaning we cannot apply any numerical methods which use calculus to find global (or local) maxima or minima of the function [28]. This means that we can only rely on a finite sequence of inputs and outputs to investigate the function due to external constraints such as cost and time [8].

Problem 3.4.1. (The black-box optimisation problem) *Consider χ the search space and $F : \chi \to \mathbb{R}$ a real-valued deterministic function where the analytic form of F is*

unknown. Suggest an input $x^* \in \chi$ *which best minimises* F*.* F *can only be called up to* $\mathcal{T} \in \mathbb{N}$ *times to obtain a set of observations* $\{(x_i, F(x_i))\}_{i=1}^{\mathcal{T}}$*, where each* $x_i \in \chi$*.*

Problem 3.4.1 allows for the existence of cases whereby F is multi-dimensional and can be taken as parameters for both continuous and discrete values. We make F deterministic to ensure the same inputs F always produce the same output. We shall call F the *objective function*.

An efficient technique used to address the black-box problem is called *sequential model-based optimisation* (SMBO), also known as *Bayesian optimisation*. SMBO is a popular tool used within the machine learning community for the hyper-parameter tuning of classifier models [109, 195, 15, 110, 111]. SMBO is a derivative-free optimisation method which iteratively constructs a cheaper *surrogate model* which mimics F. For each iteration of the surrogate model, an *acquisition function* is used to recommend where to next evaluate F, making a trade-off between the *exploration* (search areas where the behaviour of the function is unknown) and *exploitation* (conservatively stay in areas where the best minimal value of F was found) of the search space. The key idea behind SMBO uses *Bayes' theorem*, which states that the *posterior* belief on a hypothesis (or model) H being true given the collection of evidence E gathered is proportional to the product of the *prior* belief of H being true and the probability of obtaining E given that H is true [34]:

$$Prob(H \mid E) \propto Prob(E \mid H) \cdot Prob(H). \tag{3.6}$$

Equation (3.6) suggests a way to construct successive surrogate models for F: suppose we have a set of data points $\mathcal{H} = \{(x_i, F(x_i))\}_{i=0}^{N}$ and a prior belief of the objective function $Prob(F)$. Combining this with the likelihood of obtaining $\mathcal{H}$ with our prior belief, $Prob(\mathcal{H} \mid F)$, we can find the posterior belief which updates what we

Algorithm 1 Generic sequential model-based optimisation (SMBO).

Require: Objective function F, initial surrogate model $\mathcal{N}_0$, maximum number of trials $\mathcal{T}$, acquisition function A

Ensure: Observation history $\mathcal{H}$

```
1: procedure SMBO(F, N_0, T, A)
2:     H ← ∅
3:     for t = 1, ..., T do
4:         x* ← Argmin_x A(x, N_{t-1})      ▷ Suggest the next point to evaluate at
5:         Evaluate F(x*)                    ▷ Expensive step
6:         H ← H ∪ (x*, F(x*))               ▷ Update observation history
7:         Fit newer model N_t to H          ▷ Fit the newer model accordingly
8:     end for
9:     return H
10: end procedure
```

know about F:

$$Prob(F \mid \mathcal{H}) \propto Prob(\mathcal{H} \mid F) \cdot Prob(F).$$

A template of SMBO is given in Algorithm 1. We initially have a cheaper surrogate function $\mathcal{N}_0$ which we continuously update with each received output of F, an empty observation history set $\mathcal{H}$ and the maximum number of trials permitted $\mathcal{T} \in \mathbb{N}$. At trial number $t \leq \mathcal{T}$, we use the acquisition function A to choose the next input x^* to sample, given the prior belief over F modelled by $\mathcal{N}_{t-1}$. After evaluating $F(x^*)$, we update $\mathcal{H}$ with $(x^*, F(x^*))$ and fit a newer model $\mathcal{N}_t$ to F with this newer information. We iterate this process until we reach the maximum number of permitted trials.

While we will not cover the surrogate models or acquisition functions suitable for SMBO, the standard for modelling objective functions is the *Gaussian process* approach, which uses a combination of Gaussian distributions induced by $\mathcal{H}$ to derive a distribution over the space of models for F, defined by a mean and covariance function which can be analytically derived. If our prior belief of F is a Gaussian process, the posterior belief after drawing a newer sample is also a Gaussian process

[181]. For the acquisition function, a commonly used method is the *expected improvement function* $EI(x, \mathcal{N})$ which measures, given a surrogate model $\mathcal{N}$, the expectation of F exceeding the best minimal value found so far at a point $x \in \chi$ [116]. Jones [115], Villemonteix et al. [213] and Srinivas et al. [197] also discuss alternatives.

3.4.2 Tree-structured Parzen Estimator

Whilst the Gaussian process approach provides an elegant way to both optimise and fit a model to F, the time complexity of each iteration scales cubically in $|\mathcal{H}|$ due to the matrix multiplications involved, meaning an alternative is needed to accommodate for more lengthy SMBO runs. The tree-structured Parzen estimator (TPE) transforms the surrogate model constructed into two non-parametric densities formed by $\mathcal{H}$ when the output of F is above or below a threshold chosen by TPE. We assume the search space comprises of uniform (continuous), log-uniform (continuous), quantised log-uniform (discrete), and categorical (discrete) variables.

For a pair $(x, F(x))$, if the surrogate models are supposed to represent $Prob(F(x) \mid x)$, then TPE tries to model both $Prob(x \mid F(x))$ and $Prob(F(x))$, defining the former as:

$$Prob(x \mid F(x)) = \begin{cases} l(x), & \text{if } F(x) < y^* \\ g(x), & \text{otherwise} \end{cases} \tag{3.7}$$

In (3.7), l is a Gaussian mixture model (GMM) fitted to the set of observations $L \subseteq \mathcal{H}$ associated with the function values under a threshold y^* chosen by TPE (the 'good' points which sufficiently minimises F), and g is another GMM fitted to the rest of the observations. To choose the next input of F, TPE evaluates the ratio $\frac{g(x)}{l(x)}$, choosing points which have a higher probability in l and lower probability under g. By maintaining a sorted $\mathcal{H}$, the runtime of each iteration of TPE scales linearly with $|\mathcal{H}|$ and linearly in the number of optimised variables [26, 24].

3.4.3 Python Libraries

In this thesis, we use two Python libraries which apply TPE and provides ways to describe the parameters which are to be investigated:

- **Hyperopt** [90, 26, 25] provides algorithms and parallelisation infrastructure for performing efficient parameter searching. It requires users to explicitly describe the search space to take advantage of the its capabilities, allowing for flexibility of the SMBO algorithm so that it can be applied to most search problems. It has been successfully used in areas which require the training of neural networks or other forms of scientific modelling [217, 180, 68].

- **Benderopt** [86, 22] is a lightweight black-box optimiser created by Valentin Thorey, which takes inspiration from Bergstra et al. [24]. Although it has less functionality than Hyperopt, configuration spaces in Benderopt can be described in a more intuitive manner. The in-built Parzen estimator algorithm works similarly to how TPE is implemented in Hyperopt.

At a basic level, these packages can be used as follows: firstly, we define the space we wish the optimiser to search over, using a list of variables of differing types, quantised or otherwise. Next, we run a minimising function on the objective function we wish to investigate, taking as arguments the parameter space, the optimiser to use (in this case, TPE) and the number of trials before the algorithm gives a final suggestion. Example Python code using the two packages is given in Fig. 3.9 and 3.10.

After finishing an experiment, Hyperopt includes the option of collecting and saving trial data should one wish to use it in future experiments. This is done by storing the trial data into an object and modifying the result. The objective function returns into a Python dictionary which stores the inputs used and the corresponding values obtained. We can save the trial data into another file using the object serialisation library pickle,

```
# Hyperopt demonstration
from hyperopt import hp, fmin, tpe, space_eval
import numpy as np

# The function we want to minimise
# Actual optimal co-ordinates => (3*pi/2, pi)
def q(args):
    x, y = args
    return np.sin(x) + np.cos(y)

# Define the search space
space = [hp.uniform("x", 0, 2*np.pi), hp.uniform("y", 0, 2*np.pi)]

# Let TPE suggest the best set of parameters which minimises q
best = fmin(q, space, max_evals=500, algo=tpe.suggest)

print (space_eval(space, best))
# output => (4.7083977675276705, 3.1231335362403763)
# output of q with best sample => -1.99982167
```

Figure 3.9: Demonstration of the Hyperopt library

for example [174]. At the time of writing, Benderopt has no equivalent functionality but it can suggest a sample to input into F given a set of trial data; in Chapter 6, we shall use this to create a function based on Benderopt's minimise function, which takes trial data as an argument and uses it to suggest the next input at which to evaluate F before starting the experiment.

```python
from benderopt import minimize
import numpy as np

# The function we want to minimise
# Actual optimal co-ordinates => (3*pi/2, pi)
def q(x, y):
    return np.sin(x) + np.cos(y)

# Define the search space
space = [{
            "name": "x",
            "category": "uniform",
            "search_space": {
                "low" : 0,
                "high" : 2*np.pi,
            }
        },
        {
            "name": "y",
            "category": "uniform",
            "search_space": {
                "low" : 0,
                "high" : 2*np.pi,
            }
        }]

# Find the input which minimises q. Uses the "parzen_estimator"
   optimiser as default
best_sample = minimize(q, space, number_of_evaluation=500)

print(best_sample)
# output => {'x': 4.778215974445023, 'y': 3.1834439790242306}
# output of q with best sample => -1.996958547
```

Figure 3.10: Demonstration of the Benderopt library

Chapter 4

Modelling and Verification of Gossip Protocols

This chapter details the process of modelling clients and servers gossiping with each other in a network using the Chuat gossip protocols. We achieve this by abstracting the logic of the protocols and assuming that the clients behave stochastically. We distinguish between the cases of whether a split-world attack is in process or not, specify what properties we are interested in analysing and suggest an intuitive way to improve the performance of the protocols. Lastly we provide verification results for our models.

We use DTMCs to model the execution of the protocols as they provide a simple but effective way of recording the state of each gossiping node and the random behaviour of the clients do not depend on how the other entities are behaving or any previous events that occurred i.e. the transition probabilities in the model are independent. We can exploit efficient numerical techniques already implemented in PRISM, such as the Gauss-Seidel method [200], when verifying DTMCs (see Chapter 3), allowing us to find exact values for PCTL- and rewards-based specifications which allows us to evaluate

the protocol performance. Lastly, since we assume that clients of the same type behave in the exact same way, we can exploit this to perform symmetry reduction on our DTMCs and perform verification on smaller models that bisimulate the original model and preserves the results of our quantitative properties.

4.1 Network Topology

Recall in Chapter 3 that in the Chuat protocols, clients gossip STHs with each other by using servers as staging posts which distribute the data whenever they receive TLS connections. There also exist two versions of the protocol which we have called *STH-only* and *STH-and-proof*, based on the form the gossip messages take and the necessary checks clients need to perform to update themselves with any data they receive. For the purposes of modelling and verification, we define an abstraction of the network comprising of clients and the servers they regularly connect with, categorising the clients into different types based on their observed behaviour.

In our models, gossiping occurs via a sequence of rounds, each of which can be broken down into a series of discrete steps which have clients randomly connecting with servers and updating themselves afterwards. We assume that all the clients are capable of gossiping and that they are allowed to connect with at most one entity during a round. The rate at which the clients gossip with servers, which we call the *gossiping rate*, is defined as the average proportion of outgoing gossiping connections to the servers included in the network. The rates at which at which the clients connect to each domain in the network given that it will gossip with them, called the *client profile*, is the average proportion of outgoing connections to each server where gossip is being used out of the total number of outgoing gossiping connections to these servers.

Definition 4.1.1. (Network topology) *A* network topology *is a tuple*

$NT = (\mathcal{C}, \mathcal{S}, \mathcal{P}, \mathcal{G})$, *where:*

- $\mathcal{C}$ *is the set of client types,*
- $\mathcal{S}$ *is the set of server types,*
- $\mathcal{P} : \mathcal{C} \times \mathcal{S} \to [0,1]$ *is the client profile function where, for each* $C \in \mathcal{C}$, $\sum_{S \in \mathcal{S}} \mathcal{P}(C,S) = 1$,
- $\mathcal{G} : \mathcal{C} \to [0,1]$ *is the gossiping rate function.*

Our framework of having different client types in the network is useful for two main reasons. The first is that, by having clients categorised by their different features and then statistically deriving their behaviour, we can get a more realistic view of how the protocols will perform compared to just using randomised values which would not produce any meaningful results. The second reason is from a model checking perspective because, as we will show later, determining quantitative properties is expensive (increasing exponentially with the size of the network). By declaring client types, we can apply symmetry reduction (see Chapter 3) which significantly reduces the number of states we need by creating one 'meta-client' which records the states of the clients that are identical in their behaviour and initial state before the gossiping begins.

To give a small example, suppose we have the following network topology NT which includes three unique client types and two server types described as follows:

- $\mathcal{C} = \{C^1, C^2, C^3\}$,
- $\mathcal{S} = \{S^1, S^2\}$,
- $\mathcal{P}(C^1, S^1) = 0.1$; $\mathcal{P}(C^1, S^2) = 0.9$,
- $\mathcal{P}(C^2, S^1) = 0.2$; $\mathcal{P}(C^2, S^2) = 0.8$,

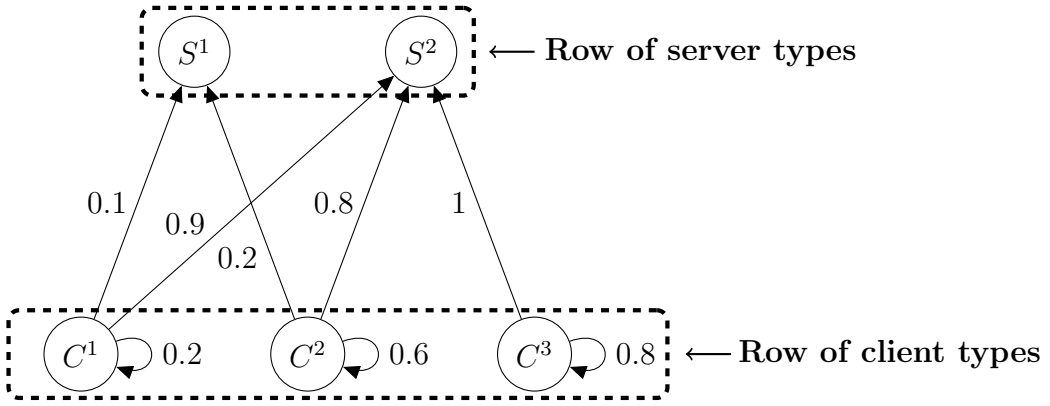


Figure 4.1: A visual representation of our example network topology NT using directed graphs. Each C^i represents an abstract client whose behaviour is based on the browsing habits statistically derived from a sample of users. Each S^j is a collection of websites or domains controlled by a single entity which we represent as a server. Each directed edge going to a client to a server has a label denoting the likelihood of a client connecting to the neighbouring server according to their profile. The gossip rate for a client type is indicated by the labels of each looped edge.

- $\mathcal{P}(C^3, S^1) = 0$; $\mathcal{P}(C^3, S^2) = 1$,
- $\mathcal{G}(C^1) = 0.2$; $\mathcal{G}(C^2) = 0.6$; $\mathcal{G}(C^3) = 0.8$.

Using directed graphs, we can depict what this network topology looks like; see Fig. 4.1 for details. Even though we did not make it explicit in Definition 4.1.1, we assume that there is a log server which entities in the network topology can contact and is operated by a maintainer, which chooses to behave either honestly or maliciously depending on the scenario we want to model; see subsection 4.2.1 for details.

4.1.1 How to Build Client Type Profiles

We list a few ways on how one could obtain client data and construct profiles for them by parsing and analysing Internet connections. To determine how users may perform in a simulation experiment, Chuat et al. [48] have clients randomly make connections based on discrete probability distributions. They also consult Amazon's Alexa service to collect data on the top domains for different countries to build profiles of how users

are likely to browse in different parts of the world. Tools and algorithms do exist to analyse user behaviour from raw datasets; Xie et al. [221] were able to reconstruct user behaviour with high accuracy by collating unencrypted web traffic, and Feng et al. [79] applied stream algorithms on data recorded from cellular networks to achieve something similar.

4.1.2 Deriving Server Types

To derive different server types, whilst a simplistic approach would be to have one server represent a single domain or host, we could also have these servers represent the services available through the Internet (e.g. banking, shopping) which our client types have access to. Using this method, to derive any client profiles, it would be necessary to categorise the connections clients make; Moore and Zuev [161] describe a way of how this can be done using naive Bayesian techniques to develop a classifier which determine the types of network traffic by application, illustrating how their classifier performs with high accuracy when refinements were applied to the manually classified training data. Jeffery et al. [73] demonstrate another way to classify network traffic using unsupervised clustering algorithms, which can be useful when peer-to-peer services obfuscate any port numbers they use to avoid detection. For more information, Nguyen and Armitage [164] provide a survey of machine learning techniques being applied to classify Internet traffic.

4.2 Modelling the Protocol

Using a network topology as a template, we consider probabilistic models which include gossip-enabled nodes where each one is of a type defined in the topology which determines their identity and behaviour; we shall call them *network models*. The basic procedure used in these models can be described as follows: first, clients

randomly decide whether to participate in a round of gossiping depending on their pre-defined gossip rate. If they decide not to, they do nothing until the next round of gossiping commences. Otherwise, they choose which server to connect with using their client profile and 'gossip' their STH. Afterwards, every client in the round updates their local state by comparing against the state variables representing the entity they are connected with and changes their own internal state accordingly. After the round is complete, the clients reset themselves by disconnecting but remember their stored messages before the next round begins.

We can initialise the states of our clients and servers in many ways because in a real network they are certain to have different STHs and SCTs stored, bringing more complexity to our model. Therefore, to avoid these problems we impose the following design restrictions:

i) Every client and server in the network will contact only one log; clients that are victims of a split-world attack are redirected to a forked version of the log.

ii) Clients and servers already have valid and non-empty data (i.e. the tree sizes of stored STHs are non-zero) before gossiping begins.

iii) All clients and servers are gossip-enabled.

iv) All the clients have previously audited the SCTs for the servers they can contact.

The state of each client in the model comprises several variables. First, c_g indicates which stage they are at in the protocol execution. A connection state c_s records the server they are currently connected with ($c_s = 0$ means that the client is not connected with anyone) and c_{skip} denotes whether or not a client decides to skip a round. If c_{skip} is true, then all the client variables do not get altered for the rest of the round apart from c_g. A variable c_{sth} keeps track of the STH the client has stored, where its usage

depends on the scenario we are looking at. Each server has one variable s_{sth} which has an equivalent purpose to c_{sth}.

We give examples of PRISM commands to show how a single gossip round is executed: first, a probabilistic choice determines whether a node will participate in the current gossip round:

$$[connect]\, c_g = 0 \;\rightarrow\; p : (c'_g = 1) \;+\; (1-p) : (c'_g = 1) \;\wedge\; (c'_{skip} = \top);$$

Next, if the client does decide to gossip (i.e c_{skip} is false), it randomly chooses exactly one of the M possible server types to gossip with, otherwise it moves on to the next phase of the round:

$$\begin{aligned}[choose]\, c_{skip} = \bot \wedge c_g = 1 \rightarrow\; & p_1 : (c'_g = 2) \wedge (c'_s = 1) + \\ & p_2 : (c'_g = 2) \wedge (c'_s = 2) + \\ & \vdots \\ & p_M : (c'_g = 2) \wedge (c'_s = 5); \\ [choose]\, c_{skip} = \top \wedge c_g = 1 \;\rightarrow\; & (c'_g = 2);\end{aligned}$$

The clients then go into the update phase. To put it simply, if the connected server has newer STH data, the client 'updates' itself by changing the value of c_{sth} (the update command server-side is defined by considering all current connections it has from clients and choose the newest STH data to update with):

$$[update]\, c_g = 2 \wedge c_{skip} = \bot \wedge server\_has\_sth \rightarrow\; (c'_g = 3) \wedge (c'_{sth} = \top);$$

$$[update]\, c_g = 2 \wedge ((c_{skip} = \bot \wedge \neg\, server\_has\_sth) \,|\, c_{skip} = \top) \rightarrow\; (c'_g = 3)$$

Lastly, when the round ends, the clients check to see if all of the clients are now

updated. If they are, then the clients progress to a further state (from which they do not exit). Otherwise, the clients reset some of their own variables and another round of gossiping commences:

$$[round\_complete]\, c_g = 3 \wedge \neg\, clients\_all\_updated \rightarrow (c'_g = 0) \wedge (c'_s = 0) \wedge (c'_{skip} = \bot);$$
$$[round\_complete]\, c_g = 3 \wedge clients\_all\_updated \rightarrow (c'_g = 4)$$

To see the PRISM models in full, consult the supporting material for this thesis [202]. In Chapter 5, when defining IDTMC models, we modify some of the above commands by replacing exact probability values with real-valued intervals with non-zero lower bounds.

4.2.1 Normal and Split-World Scenarios

We want to analyse how well the protocols adapt under normal conditions and during a split-world attack; the log maintainer is honest in the former situation, while in the latter it attempts to fork the log for malicious purposes. In the following, we denote s_i to be the i^{th} STH generated from the CT log with Merkle tree size $t_i \in \mathbb{N}$.

For the normal scenario, the nodes in the network start with an old STH s_l (with tree size t_l) which was generated before the rounds of gossiping begin. We let one server and one client have the newest STH s_m initially, with $t_l < t_m$. This is to reflect the fact that entities may choose to regularly contact the log to request newer STHs outside of gossiping most likely because they act as log auditors.

Our second scenario looks at the situation where the log maintainer decides to target a single client in the network by forking the log as part of a split-world attack. We assume that the attacker, i.e. the log maintainer, wants to sustain the attack for as long as possible until they achieve their objective of stealing sensitive information from a

victim, regardless if the attack gets discovered afterwards (a 'smash-and-grab' attack).

We describe the attack as follows: before gossiping begins, the log has already commenced a split-world attack by maintaining two separate ledgers forked from a previous version of a log whose corresponding Merkle tree had tree size t_l - the genuine log that the attacker wants non-targeted entities to see, and a rogue log created to target a victim whose Internet connections are intentionally being re-routed whenever it attempts to contact the genuine log. This rogue log includes fraudulent certificates which attacker-controlled servers use to host spoof websites. To further impersonate genuine domains, we also assume that these servers support CT gossiping.

The newer STHs corresponding to the genuine and rogue logs are s_m (the *real* data) and s'_n (the *fake* data) respectively, where $t_m, t'_n \in \mathbb{N}$ are their respective tree sizes, with $t_m < t'_n$ and $timestamp(s_m) < timestamp(s'_n)$ i.e. s'_n is the newly generated STH after s_m. This is done to bypass the `checkSTH()` function used in the Chuat protocols which checks for inconsistency (see Chapter 3). Up to tree size t_l, both logs are identical in their contents and the structure of the Merkle tree at that time, implying that consistency between STH pairs s_l/s_m and s_l/s'_n can be proven by the log server when necessary.

As previously mentioned, the goal of the attacker is to steal information from a particular client when they connect and gossip with servers under their control. To do this, it tricks or forces the victim client with sufficiently older knowledge of the log into gossiping with an attacker-controlled server that already possesses s'_n (either through phishing, man-in-the-middle attacks etc.), redirecting any of their connections so that proof/STH requests they make are given to the rogue log. Afterwards, when the fake STH data has been verified and the fraudulent certificate has been audited using its SCT, the victim updates itself by caching s'_n.

In our models, we have one updated server with s_m that is not controlled by the

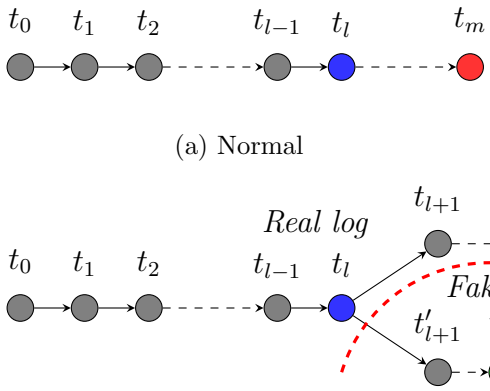
attacker, and one updated client with s'_n which we assume to be the victim. The rest of the entities in the network have an old STH s_l such that $t_l < t_m < t'_n$. We assume that no spoof servers are ever contacted again (or the probability of them receiving any connections in the future by clients is negligible) so that they are not part of the model.

In our models, detection of the split-world attack occurs when either at least one client retrieves both s_m and s'_n through gossiping and receives an erroneous response to its request for an extension proof by the log server (since the log cannot provide a valid proof between the real and fake STH data), or at least one client receives a warning message from a server which has detected an inconsistency with the log.

The variables c_{sth} and s_{sth} are integers which represent the data type each entity possesses. The rounds of gossiping end when at least one client detects something is wrong for the reasons mentioned previously. In addition, we use Boolean variables c_d and s_d as 'detection flags' which are set to true when a client or server have detected something, respectively. The changes we make to our models is that during the *update* stage, clients and servers can either update themselves with newer data or go into detection mode when an inconsistency is spotted and, during the *round complete* stage, the model checks if at least one client has c_d set to true.

4.2.2 Protocol Variations

As mentioned in Chapter 3, the format of the gossip messages is different between the *STH-only* and *STH-and-proof* versions of the Chuat protocols. Each client stores only one message in both versions, but in *STH-and-proof* a server stores multiple messages and gossips them depending on what messages it receives. In our models, by providing all the entities with a baseline knowledge of the log, in both cases we make servers gossip only the messages represented by s_{sth} in the model. Table 4.1 shows which messages correspond to the values that c_{sth}/s_{sth} can take in both our normal and split-world



(a) Normal

(b) Split-world

Figure 4.2: An abstract representation of the growth of the log in (a) a normal scenario with no attack occurring and (b) during a split-world attack. Each node in the directed graphs indicate a state of the log when the Merkle tree size was t_i, where $i \geq 0$ and $t_i \in \mathbb{N}$. For an older state of the log with tree size t_i, it is succeeded by a new state of the log indicated by an outgoing directed edge, and the tree size gets updated to t_{i+1} in the process. In the normal scenario, the current state of the log has tree size t_m and two entities in the network start gossiping with the corresponding STH s_m, while the rest of the entities have the old STH data s_l with tree size t_l, with $t_l < t_m$. In the split-world attack scenario, the log is forked into two different versions when it had tree size t_l, creating a genuine log with STH s_m and tree size t_m, and a rogue log with STH s'_n and tree size t'_n, where $t_m < t'_n$. In our network models, one server has cached s_m sourced from the genuine log while a designated victim client has s'_n obtained by gossiping with an attacker-controlled server . The rest of the entities in the network initially have cached s_l.

models.

We are interested in how often the protocols make clients contact the log during their execution. To measure this in our models we use reward structures to count how many times a consistency proof is requested from clients. Each variation of the protocol has different requirements for the clients to call the log depending what gossip messages they receive:

- *STH-only* - the tree size of the received message is different from what the client already knows. Regarding our models, this means that between a connected client/server pair the log is 'called' when the value of c_{sth} is not equal to s_{sth}.

Normal case		Gossip message format	
c_{sth}/s_{sth}	Data Type	*STH-only*	*STH-and-proof*
`False`	Old data	s_l	$(s_a, s_l, p_{a,l})$
`True`	New data	s_m	$(s_l, s_m, p_{l,m})$

Split-world case		Gossip message format	
c_{sth}/s_{sth}	Data Type	*STH-only*	*STH-and-proof*
0	Old data	s_l	$(s_a, s_l, p_{a,l})$
1	Real data	s_m	$(s_l, s_m, p_{l,m})$
2	Fake data	s'_n	$(s_l, s'_n, p'_{l,n})$

Table 4.1: The tables above describe what the c_{sth}/s_{sth} variables represent in the models for each value in both the normal and split-world cases. Here, s_a is an STH with tree size $t_a > 0$, where $t_a < t_l$, and $p_{a,b}$ be a consistency proof between STHs s_a and s_b. We let $p'_{l,n}$ be a 'fake' consistency proof between s_l and s'_n.

- *STH-and-proof* - if the client's message is $(s_a, s_b, p_{a,b})$ and the obtained message $(s_c, s_d, p_{c,d})$, where $t_a, t_b, t_c, t_d \in \mathbb{N}$ are tree sizes, then we must have that $t_b \neq t_c$ and $t_b \neq t_d$. Due to our model design, this condition can occur if a client is already updated but the server it is gossiping with is not. The client will also request a proof when it obtains both the real and fake data which will trigger a detection.

4.3 Specification of Protocol Properties

We want to evaluate certain quantitative properties which will help us determine if the protocols provide good security and do not place a burden on bandwidth. To be more precise, we look at three properties:

i) *Data dissemination*: How effectively do the protocols spread the new data to each client in the network?

ii) *Protocol efficiency*: How much demand does the protocol put on the log?

iii) *Rate of detection*: How quickly can the protocols detect a split-world attack occurring in the network?

To evaluate property i) using our models, we will measure the expected proportion of clients to possess the newer STH per round, computed using reward structures which track the current proportion after round $r \in \mathbb{N}$. In PRISM, this is written as:

$$\mathbf{R}_{=?}^{client\_proportion}[\mathbf{I}^{=r}],$$

where $\mathbf{I}^{=r}$ is the instantaneous reward function that outputs the state reward for a finite path in the model after r rounds (see Chapter 3). The reward structure *client_proportion* simply calculates the fraction of clients that possess the newer STH at every state in the model. Alternatively, we can find the expected amount of gossiping rounds it takes until every client in the network has the latest data, i.e., c_{sth} is set to true every client:

$$\mathbf{R}_{=?}^{rounds}[\mathbf{F}\ \ clients\_all\_updated\ ].$$

In the case of the split-world scenario, since we now have three types of STHs being gossiped including the fake data generated from the rogue log, we measure the expected proportion of clients to possess each type of STH per gossiping round:

$$\mathbf{R}_{=?}^{data\_type}[\mathbf{I}^{=r}].$$

Here, the reward structure *data_type* works similarly to *client_proportion* but counts the proportion of clients that have a specific type of STH.

We also measure ii) using reward structures, distinguishing between the conditions for each variation to request a proof and find the cumulative amount of log connections made from clients per round. We use the property:

$$\mathbf{R}_{=?}^{log\_connections}[\mathbf{C}^{\leq r}],$$

where $\mathbf{C}^{\leq r}$ is the cumulative reward operator that outputs the accumulated state and transition rewards for a finite path in the model after r rounds. The reward structure *log_connections* counts the number of log connections made during the *update* phase of the round. Property iii) is exclusive to the split-world model and can be written using the **P** operator which is used to compute the likelihood of the occurrence of a specified event:

$$\mathbf{P}_{=?}[\mathbf{F}^{\leq r}\ \mathit{detect}].$$

Lastly, to find the expected number of rounds which will take place before the *detect* event occurs, we use the property $\mathbf{R}^{\mathit{rounds}}_{=?}[\mathbf{F}\ \mathit{detect}]$.

4.4 Server-to-server Gossip

One of the key benefits of our approach using probabilistic verification is that we can investigate the effect of modifying a protocol in terms of performance. This can help in the future development of the protocol. To demonstrate this, we define a variant of the gossip protocols in which the servers gossip directly with each other instead of behaving as static entities. Server-to-server gossiping can help spread STHs to different regions of the Internet quickly. This forgoes the need for users to venture outside their typical activity to obtain fake data from potentially malicious sources, which we rely on in the case of client-to-server gossip.

Some design decisions arise when deciding how to make the servers gossip. We need to find the correct balance between data sharing and servicing clients and decide how many servers a server should contact in each session; having too many servers gossip with each other at the same time will place a strain on bandwidth and will result in dissatisfied customers. We suggest some ways in which servers can discover peers to gossip with, inspired by existing mechanisms used by the Tor anonymity network and

BitTorrent:

- **Directory authority** - Tor clients contact a directory authority to discover any relay points which can be used to create secure channels. The relay points are randomly selected periodically to prevent linking a circuit and a user's activity [204]. To apply this to our extended protocols, we can establish similar authorities which keep track of a list of selected servers for gossiping due to their popularity with clients or availability.

- **Distributed Hash Table (DHT)** - BitTorrent uses a DHT mechanism which removes the need for a point of authority (a 'tracker') which keeps track of all the clients in the network. Clients store a DHT, which in practice acts as a routing table, for a small number of nodes that have responded to previous requests successfully [144]. Each node in the network is given a unique identifier and uses a distance metric to measure how close it is to other nodes; the higher this value is for certain nodes, the more detailed their entries are in the table (see the *Kademlia protocol* [152]). To find peers for a specific torrent, a client would use the DHT to seek out neighbouring nodes and ask them for information on other peers that are downloading the same torrent.

We extend our models with server-server gossip using a simple abstraction: when a server has the latest log data it gossips this outside the update phase, i.e. before clients connect with them. To limit the additional burden on each server we fix the probability of a server choosing another server to gossip with (including itself) to be $\frac{1}{5}$. To reduce the complexity in our models, gossiping is conducted using unidirectional channels i.e. servers send messages but do not receive replies to any of them.

Client type frequency	C^1: 1, C^2: 3, C^3: 1
Normal scenario initial set-up	Client of type C^3 and server of type S^1 have the newest data
Split-world scenario initial set-up	Client of type C^3 has fake data and server of type S^5 has real data
C^1 gossip rate	0.8
C^1 distribution	$[0.02, 0.28, 0.7, 0.0, 0.0]$
C^2 gossip rate	0.6
C^2 distribution	$[0.02, 0.28, 0.0, 0.7, 0.0]$
C^3 gossip rate	0.2
C^3 distribution	$[0.02, 0.28, 0.0, 0.0, 0.7]$

Table 4.2: Initial modelling setup for both model types. For each client type, they connect with server types S^1 and S^2 with probabilities 0.02 and 0.28, respectively. They also connect with one other unique server with probability 0.7 e.g. the client type C^1 connects with server type S^3 with probability 0.7.

4.5 Experimental Results

4.5.1 Protocol Designs Analysed

Now we demonstrate the use of our models using artificial network data, starting with the presumption that our clients are grouped into three abstract types with differing behaviours, which we will call C^1, C^2 and C^3; we also use five server types $S^1, \ldots, S^5$. Suppose we have a network with five clients categorised into these three client types and five servers which are of a unique type. We fix a representative set of probabilities for the client profile probabilities, the initial setup we use for each model type and the number of clients of each abstract type, as shown in Table 4.2. For each client type, the probability of connecting to servers of type S^1 and S^2 are 0.02 and 0.28, respectively. They also connect with one other unique server with probability 0.7.

In total, we study four protocol variants:

- *STH-only* protocol without servers gossiping,
- *STH-only* protocol with servers gossiping,

- *STH-and-proof* protocol without servers gossiping,
- *STH-and-proof* protocol with servers gossiping.

The first and third variants are the original designs presented by Chuat et al. [48], which we will refer to as the *initial* designs. The other two are our modified versions which include server-to-server unidirectional gossip, which we call the *extended* designs. We model both types of design under normal conditions and when a split-world attack is taking place, both described previously in section 4.2.1.

To validate our verification results, We use PRISM's discrete event simulator to perform statistical model checking using the CI method (see Chapter 3). In the following graphs where we analyse the performance of the protocols over a sequence of rounds, we give the 99% confidence interval for each approximate value found using black error bars.

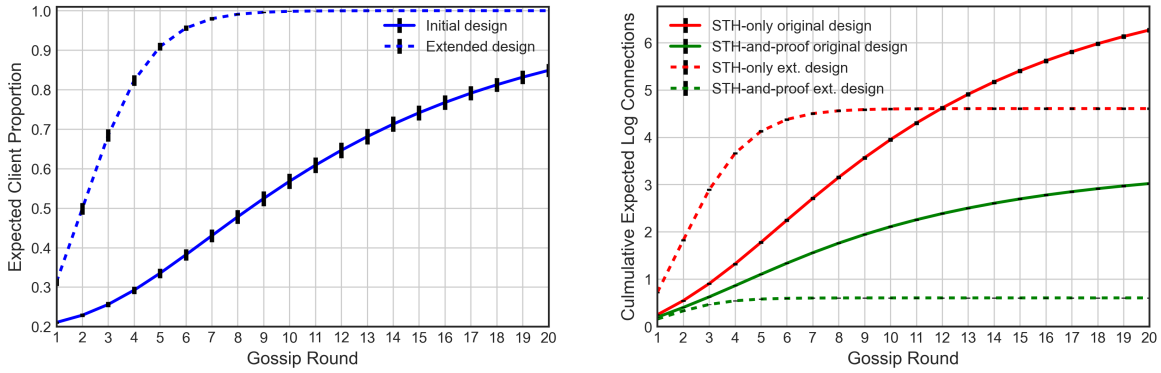
4.5.2 Verification Results

Model Statistics

We use PRISM's implementation of symmetry reduction (see Chapter 3) on our models to improve scalability when clients exhibit identical behaviour with the same initial state and compare model statistics before and after symmetry reduction is applied. Despite using a small network topology, it appears that our normal model types are relatively inexpensive to model check compared to our split-world model types, largely because there are fewer variables being used in the former. There is also the common feature in all our models where each client can only gossip with three distinct servers; clearly, the more possibilities a client can connect with, the more states needed to accommodate these choices and thus the more expensive model checking will be. Furthermore, implementing server gossip increases the number of

Scenario	States (before SR)	States (after SR)	Transitions (after SR)	Build time (s)
Normal conditions without server gossip	215,947	49,850	86,902	0.6
Normal conditions with server gossip	304,310	72,256	137,824	0.5
Split-world conditions without server gossip	1,427,618	304,515	525,583	223.7
Split-world conditions with server gossip	37,017,853	7,984,798	41,737,133	215.8

Table 4.3: Model statistics for each scenario investigated, before and after symmetry reduction (SR) was applied. The build time consists of both building the original model and applying SR to it.



(a) Client proportion

(b) Log connections

Figure 4.3: Plots of the model checking results for the normal scenario. Allowing servers to gossip improves the data spread and reduces the need for clients to request an extension proof from the log, regardless of whether STH-only or STH-and-proof is used. In each graph, we give the 99% confidence interval for each gossip round found via statistical model checking using black error bars.

states in the model than without so we must abstract this behaviour as much as possible to minimise state space explosion. We synchronise server gossip for each type of data with the actions that occur before or after the client update phase, computing the probability of a server obtaining a data type and updating themselves accordingly. Table 4.3 gives the complete set of statistics for the models we used for verification.

Analysis of Normal Scenario Models

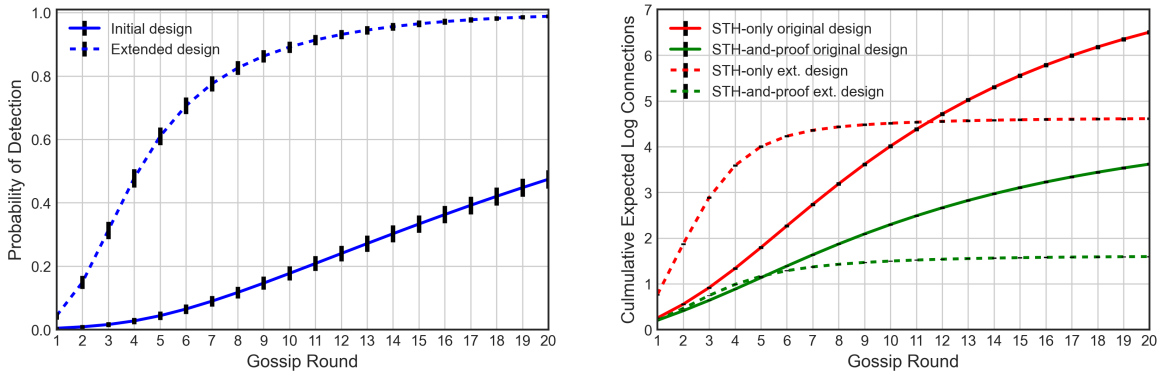
We measured the expected proportion of clients which have the latest log data (Fig. 4.3(a)). It is clear that having servers gossip helps to improve the data spread; as there are more chances of a server being updated per round, this will impact the client's chance of obtaining the latest data too. For both versions of the Chuat et al. protocol, we expect the dissemination rate to remain the same; the updating procedure for both of them are similar as the protocol prioritises the data the client will store by the tree sizes of the received STHs and the SCT validation procedure is exactly the same.

We also measured the number of client connections made to the log to determine the efficiency of the protocol. Fig. 4.3(b) shows that *STH-and-proof* requires fewer log connections from clients. For *STH-only*, the client must always contact the log whenever it sees a different STH to what it already has cached even if may have seen it before, making this version of the protocol very inefficient, whilst for *STH-and-proof* the conditions to request a proof are very specific so the chances of a client calling the log are small. Server-to-server gossip slightly improves efficiency for both versions in the long term, most likely because servers will update themselves sooner than clients.

Analysis of the Split-world Scenario Models

We expect the chances of detection per gossiping round to be the same for both versions of the protocol; in the context of our PRISM model design, the conditions for detection in *STH-only* and *STH-and-proof* are exactly the same.

From Fig. 4.4(a), we see that using the original protocol to detect attacks takes a while: after twenty rounds there there is only about a fifty per cent chance for a client to notice that something is wrong. However, making servers gossip significantly improves the detection rate as the legitimate data will be spread through the network quickly



(a) Detection rate

(b) Log connections

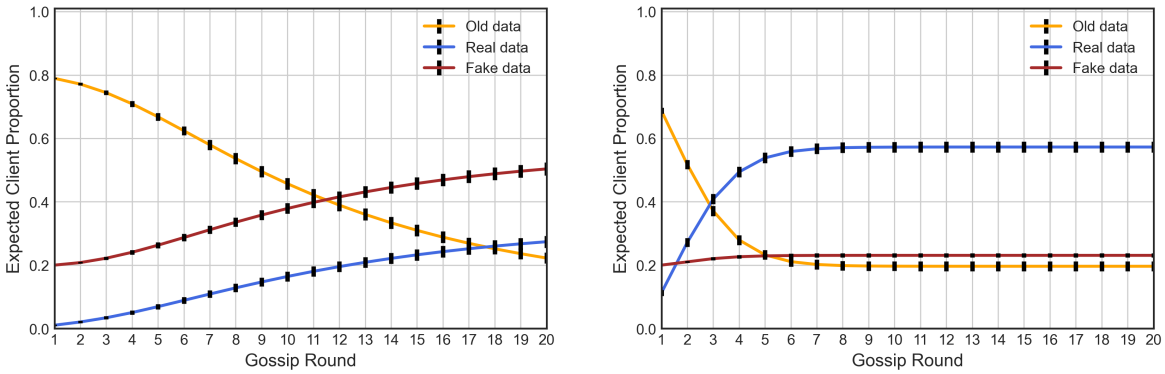
Figure 4.4: Plots of the model checking results for the split-world scenario. Similar to the normal scenario, the *STH-and-proof* protocol with servers gossiping can quickly detect attacks and is economical at the same time.

and result in an inconsistency being found when fake data is gossiped. We note that the results we obtained are influenced by the initial conditions in our network; if we let a client with a high connectivity rate be targeted, then it would be more likely for someone to detect an attack in only a few rounds. To address this issue, in Chapter 6 we employ techniques to intelligently search over all possible configurations of a network model to find a *bad one* which minimises an important quantitative property of the gossip protocols.

As for the expected cumulative amount of log connections made per round, the patterns in the results from Fig. 4.4(b) are similar to what we have observed previously in the normal scenario but slightly more log connections are made on average due to the addition of more types of data being passed around in the network.

Fig. 4.5(a) and (b) shows the expected proportion of clients to have each type of STH per round when the initial and extended design are implemented, respectively. Note that the data distribution among clients in the split-world scenario is identical for *STH-only* and *STH-and-proof* as the update procedures for both versions are identical.

From Fig. 4.5 we see that, by using the initial design, clients are more likely to



(a) STH distribution (initial design)

(b) STH distribution (extended design)

Figure 4.5: Plots showing the STH distribution for the split-world scenario. In the extended designs, the clients were more likely to have the genuine STH than the fake one, whereas the opposite result was seen using the initial designs.

possess data sourced from the rogue log than the legitimate data after twenty rounds. This is due to how rare it is for someone to gossip with the server which initially has the real STH while a fake STH with a larger tree size will spread relatively quickly. This is in contrast to what happens when the extended design is used where after five rounds more than half the clients are expected to have the real STH. We notice in Fig. 4.5(b) that the proportions remain constant after seven rounds; as detection happens sooner, the clients in the model retain the data they have at the moment when gossiping stops.

4.5.3 Improving Scalability Using Statistical Model Checking

We further investigate the gossip protocols when deployed in larger networks; this is hard to achieve through model checking so we rely on PRISM's statistical model checking capabilities to find approximate values at the expense of accuracy. We have extended our models to include 50, 100 and 200 clients while still fixing the number of of servers at five. We scale the number of each type with respect to the ratios we used in the original five-client network; while type 2 clients still make up a majority, there are also slightly more clients of type 1 than of type 3. Table 4.4 gives the frequency of

	Client type frequency		
# Clients	Type C^1	Type C^2	Type C^3
50	12	31	7
100	25	60	15
200	50	120	30

Table 4.4: The number of clients of each type for the differently sized networks we used.

each client type for our networks of different sizes.

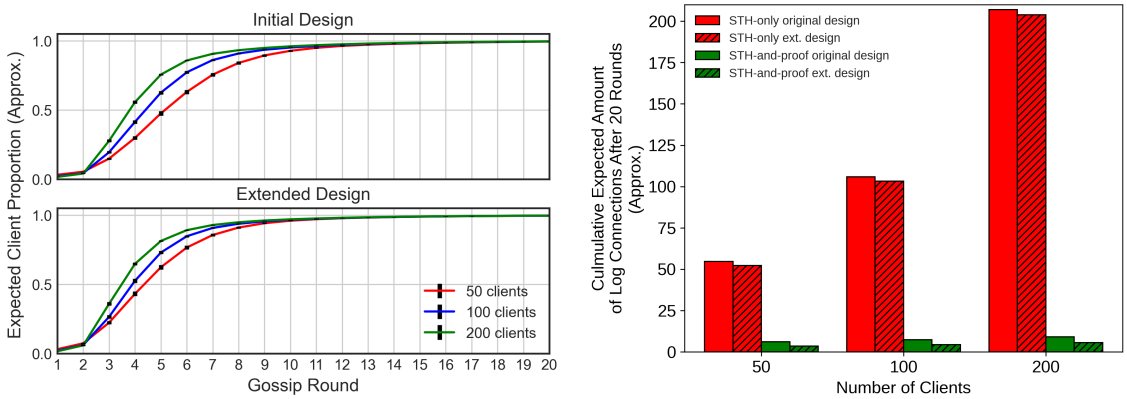
For all our results, we use the CI method to find our estimations with their respective 99% confidence interval by sampling 2000 random paths. The amount of time it took to output results did often fluctuate but in general it took longer to generate results for our split-world models. Estimating the expected STH distribution per round was the most time-consuming property to estimate.

Normal Models

For both the initial and extended designs, we approximate the data dissemination for the first twenty rounds of gossiping. In Fig. 4.6(a), we also provide the confidence intervals at each round represented by a black bar. We can clearly see by having our servers gossip the clients will get updated quickly. When measuring efficiency, in Fig. 4.6(b) we see a similar pattern emerging from when we looked at a smaller network, with the *STH-and-proof* version looking more scalable as the number of clients increases. However, in both versions of the protocol, server gossip does not appear to affect efficiency very much.

Split-world Models

The approximate detection rate for the first twenty rounds is shown in Fig. 4.7(a), suggesting that server gossip improves this property. One interesting observation is that after round nine, the detection rate for the extended design remains constant no



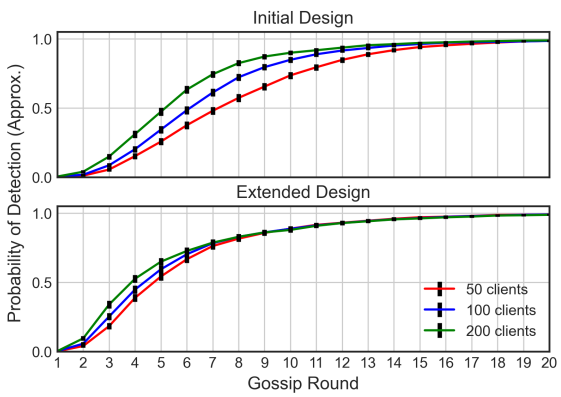
(a) Data dissemination

(b) Log connections

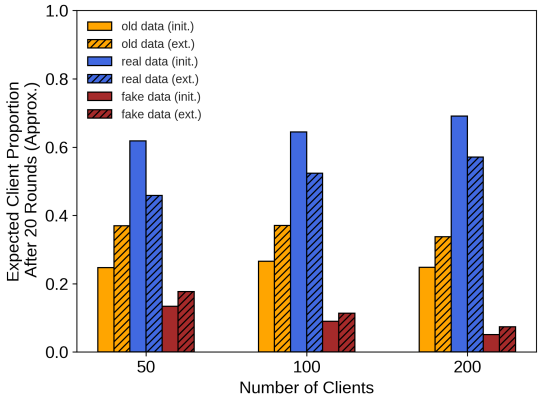
Figure 4.6: Statistical model checking results for the normal scenario for 50, 100 and 200 clients. The number of servers is five in all cases. We notice that the *STH-and-proof* version of the protocol scales well when the number of clients increases in the network in terms of efficiency, whilst this is not the case for the *STH-only* version even with server-to-server gossip enabled. Furthermore, clients tend to get updated earlier with server-to-server gossiping enabled.

matter how many clients there are in the network. This is probably due to the fact that as more servers go into the detection phase early, computing the probability of detection reduces to finding the chances of at least one client connecting to a server which has already started to gossip warning messages.

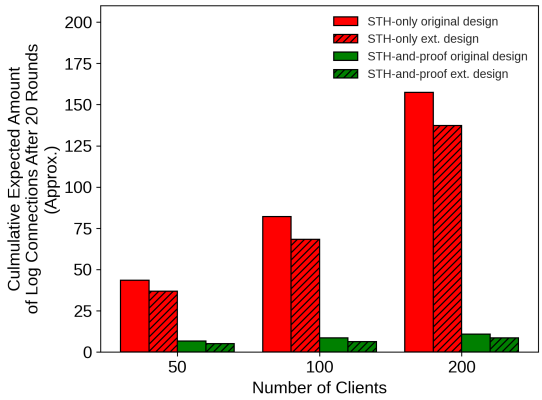
From Fig. 4.7(b), we see clients are more likely to have the real STH invariant on the amount of of clients present in the network. Fewer clients are expected to update themselves with the fake STH but we should note that the initial conditions on the model influence this outcome. In Fig. 4.7(c) we see the same outcomes in the expected cumulative total of log connections as in the normal scenario but *STH-and-proof* produces slightly more connections than usual while *STH-only* achieves the opposite.



(a) Detection rate



(b) STH-distribution



(c) Log connections

Figure 4.7: Statistical model checking results for the split-world scenario. The STH distribution remains relatively constant as the number of clients increases and making servers gossip help improve the detection rate.

4.5.4 Validating Results Against Simulations Using Randomly Sampled Data

In the models we previously analysed, the probabilities represented statistical averages for each client type, namely the average proportion of gossip connections made to servers in the network. To conclude this section, to test if the results obtained from verification using such averages give an accurate indicator of protocol performance, we compare them against approximations for the same quantitative properties using models induced from randomly sampled 'concrete' data from a batch of clients, provided these

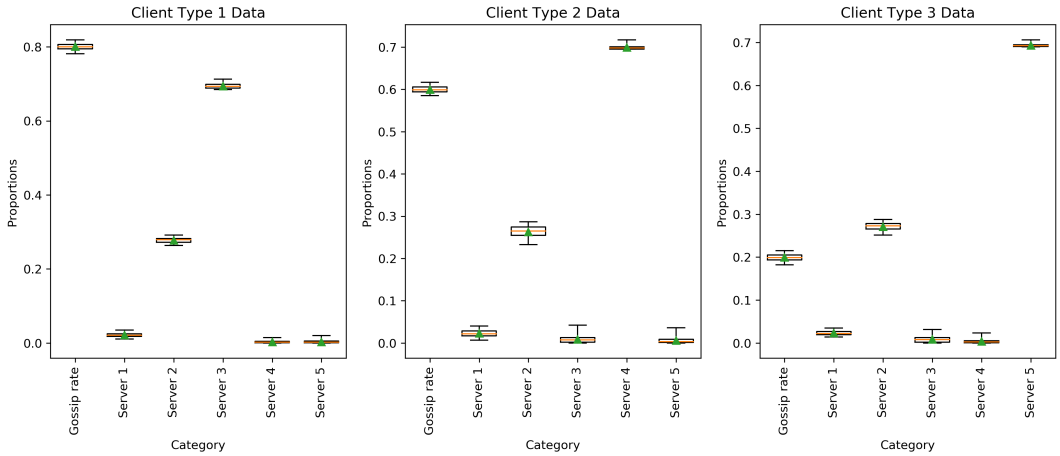


Figure 4.8: Box-and-whisker plots showing the overall range of values for each randomly generated proportion, broken down by client type. The tail ends represent the maximum/minimum values for that category, the orange line and green triangle giving the median and mean value for that data type, respectively.

clients have already been categorised (i.e. assigned a type) and clients belonging to the same category behave very similarly. We use the same modelling setup as described in Table 4.2 but this time we randomise the gossip rates and profiles for each client node in the network.

We produce artificial client data by randomly generating 100 samples for each client type (creating 300 individual samples in total), having the proportions for each sample that make up the gossip rate and the individual client profile randomly chosen within a prescribed range of values. When randomly choosing server connection proportions for each type, we adapt an algorithm used in this thesis so that the sum of these values equal one; see Appendix B for more details. We keep the range of possible values allowed for each proportion sufficiently small so that, when aggregating the data of all the client samples belonging to a single type, the average proportions are roughly equal to the probabilities used in our models (see Table 4.5 for details). We give a visual representation of the sampled data using box-and-whisker plots in Fig. 4.8.

To compare against the model checking results, we approximate the values for all

	Category					
Client type	Gossip Rate	Server 1	Server 2	Server 3	Server 4	Server 5
Type C^1	0.801	0.022	0.278	0.694	0.003	0.003
Type C^2	0.600	0.023	0.263	0.009	0.699	0.006
Type C^3	0.200	0.023	0.271	0.009	0.004	0.693

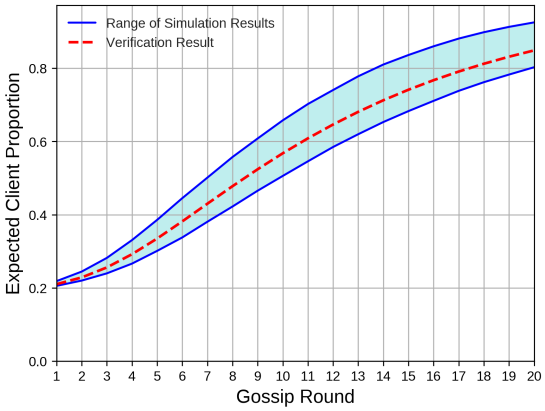
Table 4.5: Mean values for each proportion, rounded to three decimal places each.

the properties we previously analysed using statistical model checking. For a given scenario and protocol design, we perform 1000 trials where each time we randomly choose individual client samples whilst still respecting the client type frequency in the verification models (meaning that we randomly choose a single client of type C^1 and C^3 and three distinct clients of type C^2), construct the PRISM model based on the chosen samples and finally approximate results for the first twenty gossip rounds. To make the estimations as accurate as possible, we find the 99% confidence level and have PRISM generate 5000 path samples before approximating each result. Figures 4.9, 4.10 and 4.11 gives the full set of results; in each graph, the blue shaded areas give the range of all estimations found for a given property (we do not plot any confidence intervals), whilst the red dashed line gives the corresponding verification result.

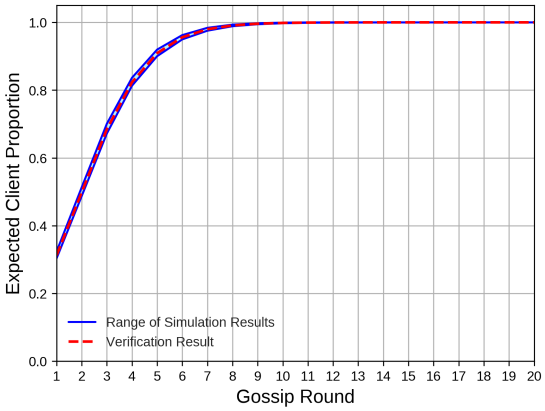
Looking at the data visualised in Fig. 4.9, the model checking results for the normal scenario fall within the shaded regions and there is less deviation in the estimations, indicating that the probabilities we used helped to give a good indicator of protocol performance, irrespective of the protocol design used. This is especially true for the extended design as the estimations are almost equal with the model checking results in some cases (see Fig. 4.9(b) and (d)).

As for the split-world scenario, when looking at the initial protocol design (Fig. 4.10), the verification results are a coarser indicator of performance over a set of concrete results. For example, while the verification results for the efficiency properties overlap with the statistical data quite well (Fig. 4.10(e) and (f)), it is hard to say the same when measuring the probability of detection (Fig. 4.10(a)) or the data type distribution

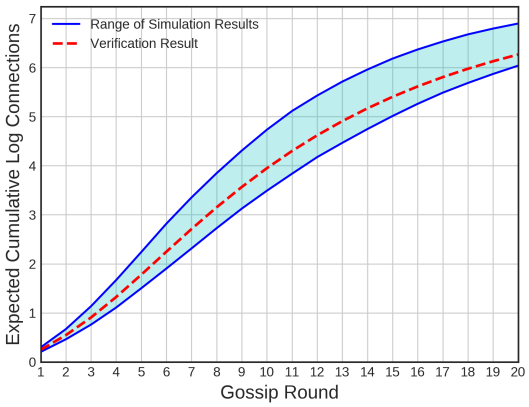
(Fig. 4.10(b)). On the other hand, when comparing the results for the extended design (Fig. 4.11), there is little variance between the two sets of results.



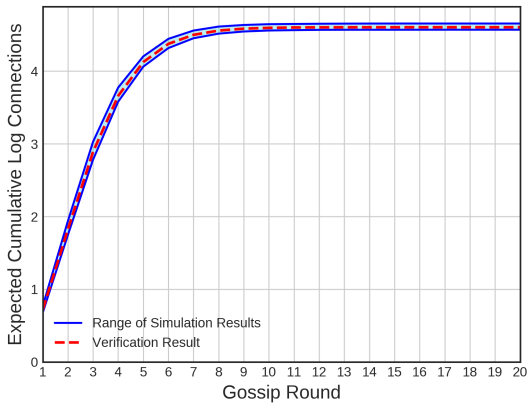
(a) Data dissemination, initial design



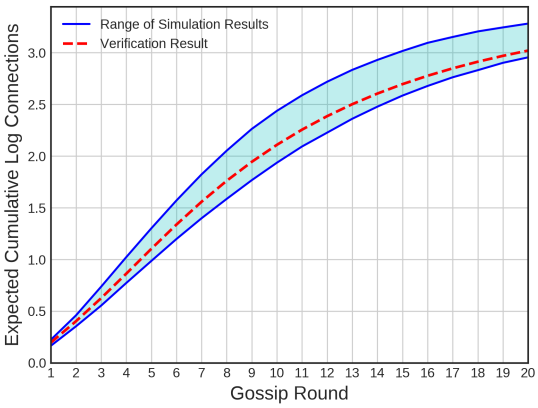
(b) Data dissemination, extended design



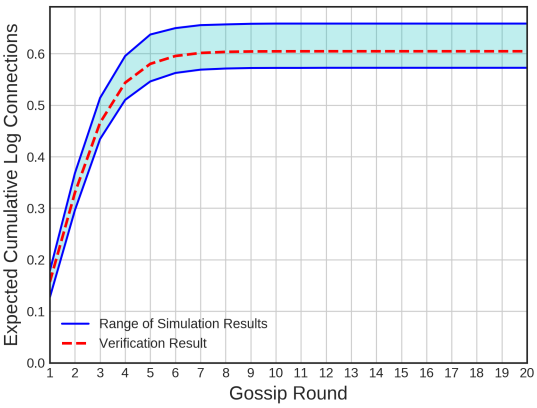
(c) *STH-only* log connections, initial design



(d) *STH-only* log connections, extended design

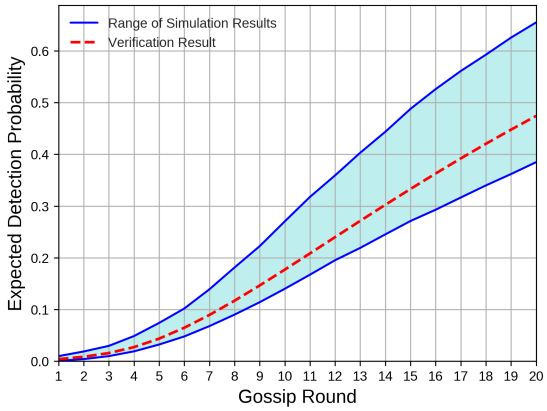


(e) *STH-and-proof* log connections, initial design

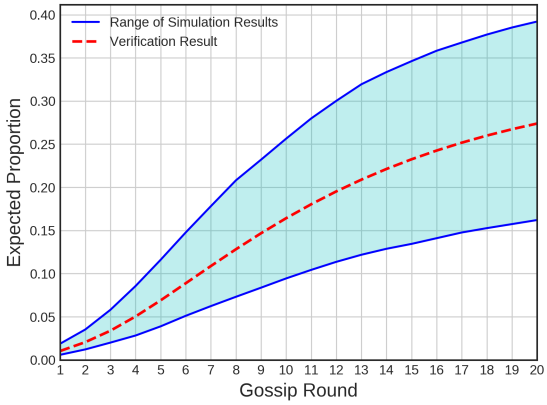


(f) *STH-and-proof* log connections, extended design

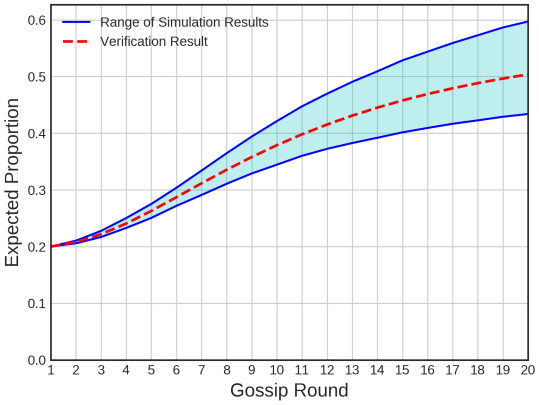
Figure 4.9: Comparing the range of approximate results using artificial sample data (shaded blue area) against the model checking results presented earlier in this chapter (red dashed line) in the normal scenario. The experiment performed 1,000 trials per property by randomly selecting sample data to initialise the client probabilities whilst still preserving other aspects of the model structure previously used for verification. To make the approximations as accurate as possible, PRISM's CI method is used with 99% confidence and 5,000 path samples.



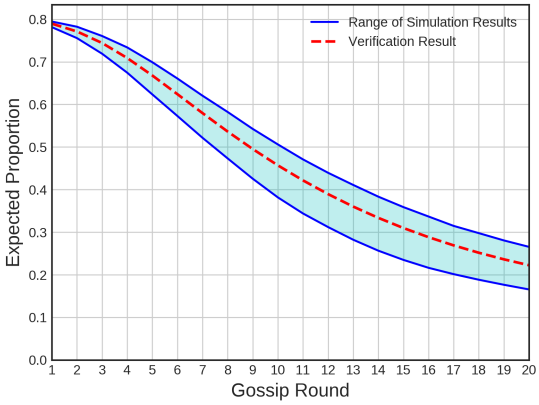
(a) Detection rate



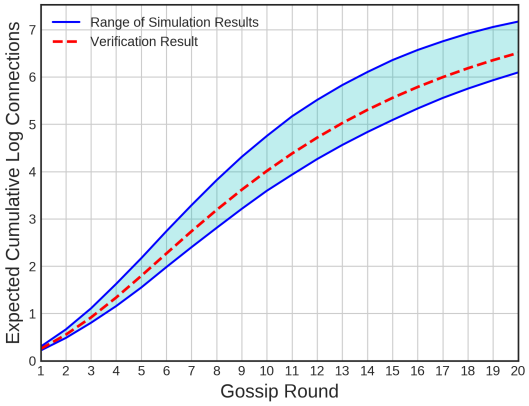
(b) Real data distribution



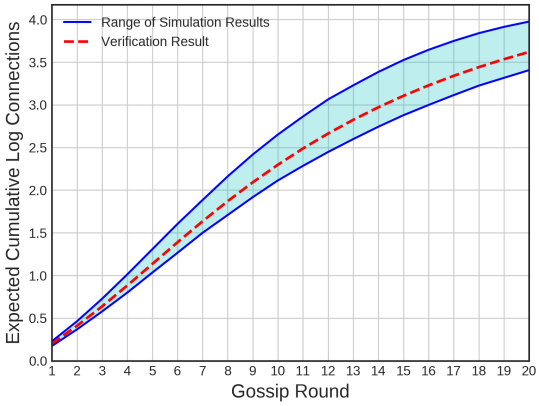
(c) Fake data distribution



(d) Old data distribution

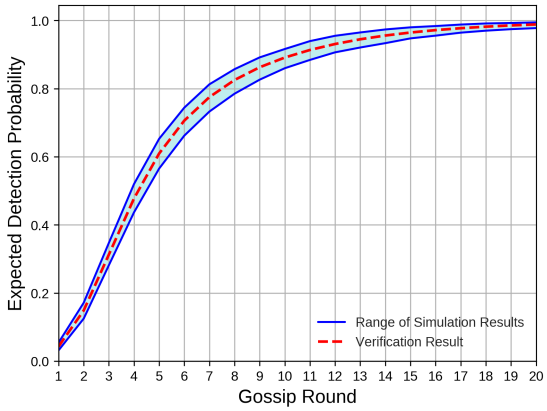


(e) *STH-only* log connections

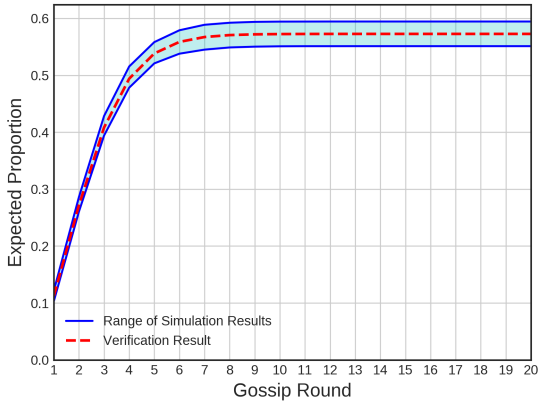


(f) *STH-and-proof* log connections

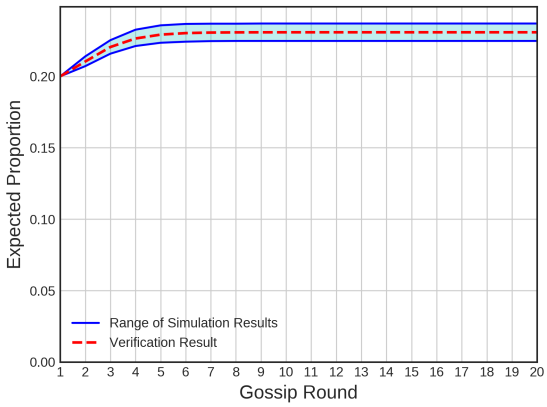
Figure 4.10: Comparing the range of approximate results using artificial sample data against the model checking results presented earlier in this chapter for the split-world scenario (initial protocol design only).



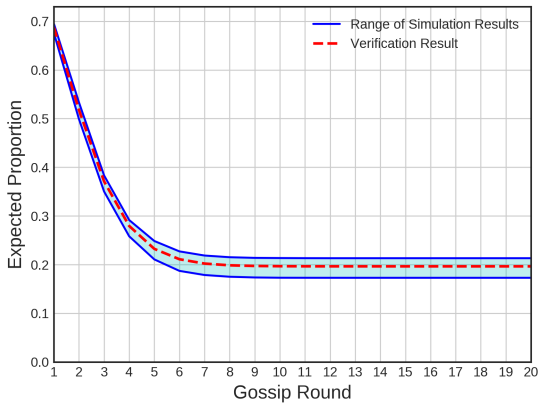
(a) Detection rate



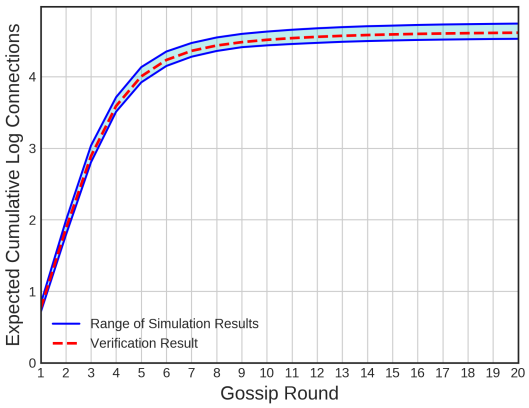
(b) Real data distribution



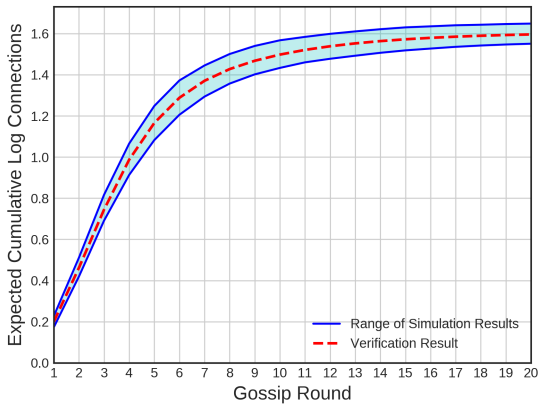
(c) Fake data distribution



(d) Old data distribution



(e) *STH-only* log connections



(f) *STH-and-proof* log connections

Figure 4.11: Comparing the range of approximate results using artificial sample data against the model checking results presented earlier in this chapter for the split-world scenario (extended protocol design only).

4.6 Summary

This section presented a methodology for formally evaluating quantitative aspects of the security and efficiency of the gossip protocols for certificate transparency using probabilistic model checking. We explained our abstraction of the network and the PCTL properties we used. We have also proposed an improvement to the original protocol in which servers gossip directly with each other (instead of just via clients) and our verification results show that this extension improves the security and efficiency aspects of the protocols. Lastly, we used PRISM's discrete event simulator to approximate protocol properties in networks with a large amount of clients and when comparing against the use of raw client data to build models to see if the protocol performance remains relatively consistent when we adjust some of the PRISM model parameters.

However, using exact and arbitrary values for probabilities clearly does not produce meaningful analysis and perturbations are likely to be present when trying to capture client type behaviour; in the next chapter, we will show how to introduce uncertainty into our models in the form of IDTMCs. To alleviate the scalability issues we are likely to encounter when analysing such models, we apply a technique to collapse them at the cost of approximating protocol properties.

Chapter 5

Tackling Uncertainty and Unscalability using IDTMCs

In the previous chapter, when we performed verification on a collection of DTMC models, we used exact values for probabilities which may not be justifiable in more rigorous analysis. A reason for this is that when collecting data from gossiping clients belonging to a particular category or type, there is a chance that the observed behaviour of the clients does not follow a common pattern of behaviour and thus it is not possible to find a representative client type profile. Another reason is that using exact values makes us only look at *one instance* of a possible gossiping scenario and not a *range* of possible scenarios that allows us to find best- and worst-case protocol performances. We also have issues of model unscalability when the state spaces for our models were large and at times relied on statistical methods to find approximate results.

This chapter shows how transforming our models into interval discrete Markov chains (IDTMCs) can resolve the issue of uncertainty present in client behaviour. Using prototype implementations, we perform verification on IDTMCs and show how we can abstract them to obtain smaller models which significantly reduces the time

needed to complete the model checking process. To see the models and implementations in full, please refer to the supporting material [202].

5.1 Using IDTMCs When Client Probabilities are Unknown

In Chapter 3, we discussed IDTMC models where exact probabilities are replaced with real-valued intervals, introducing non-determinism over the set of possible probability distributions to use when transitioning from a state. In this section, we transform our models from Chapter 4 into IDTMCs so that we have under- and over- approximations for client type behaviour, describing a prototype extension of PRISM which performs verification on these IDTMCs.

5.1.1 IDTMC Network Model Design and Properties

As noted in Chapter 4, a network topology NT is defined by: the set of client types, $\mathcal{C}$; the set of server types, $\mathcal{S}$; the client type profile function, $\mathcal{P}$; and the gossiping rate function $\mathcal{G}$. For a client type $C^i \in \mathcal{C}$, $\mathcal{G}(C^i)$ is defined to be the average proportion of outgoing gossiping connections clients of type C^i make to the server types in $\mathcal{S}$. For a pair $(C^i, S^j) \in \mathcal{C} \times \mathcal{S}$, $\mathcal{P}(C^i, S^j)$ is defined to be the average proportion of outgoing gossiping connections from C^i to S^j out of all the total gossiping connections made to the server types in $\mathcal{S}$.

As we have shown empirically in Chapter 4, this interpretation of the functions $\mathcal{P}$ and $\mathcal{G}$ suffices to analyse network models where type-equivalent clients rarely deviate in their behaviour. While the notion of client types is useful when aggregating batches of client data coming from a single source (e.g. a country, Internet trace etc.), a consequence of this is a diverse range of behaviour where no representative profile of

them is possible. To accommodate this, instead of having $\mathcal{P}$ and $\mathcal{G}$ map to exact values, they could map client types to *real-valued intervals* instead, implying this will transform our probabilistic models into IDTMCs. Mathematically speaking, for a client type $C^i \in \mathcal{C}$, the gossip rate function $\mathcal{G}$ is replaced by two real-valued functions $\mathcal{G}^u$ and $\mathcal{G}^l$ which give the upper and lower bound of the gossip rate for $\mathcal{G}(C^i)$, respectively. We can also define something similar for $\mathcal{P}$.

To efficiently model check IDTMCs in PRISM, we implement an extension that adapts value iteration [178, 11] used in MDP model checking to find the maximal or minimal probabilistic reachability given a set of target states T in a recursive manner. For an IDTMC $\mathcal{M} = (S, s_\iota, P^u, P^l, AP, L)$ and state $s \in S$, to compute the maximum probability of reaching T within $n \in \mathbb{N}$ steps, denoted as $p_{(s,n)}$, the algorithm uses, for every state $s' \in S$, the maximum probability of reaching T within $(n-1)$ steps from s', or $p_{(s',n-1)}$, deriving the probability distribution that assigns as much probability to outgoing states of s for which the reachability probability is maximal i.e. an *extreme distribution*, to maximise a certain sum [119]:

$$p_{(s,n)} = \max \left\{ \sum_{s' \in S} \delta(s)(s') \cdot p_{(t,n-1)} \;\middle|\; \delta(s) \in Dist_s \right\}.$$

By definition, we let $p_{(s,0)} = 1$ if $s \in T$ and zero otherwise. The algorithm for finding the minimal probabilistic reachability for a given state is derived similarly.

To demonstrate this process, three client types are used, C^1, C^2, C^3 and five server types $S^1, \ldots, S^5$ as used in the previous chapter, however, for testing purposes we model a smaller network to avoid building a large state space (the IDTMC verification tool uses the explicit engine in PRISM to build the model which typically scales slightly less well than the symbolic engines as the number of states in the model increases). Present in the network topology is one client of each distinct type and the five server types,

Server type \ Client type	C^1	C^2	C^3
S^1	$[0.01, 0.1]$	$[0.01, 0.1]$	$[0.01, 0.1]$
S^2	$[0.2, 0.4]$	$[0.2, 0.4]$	$[0.2, 0.4]$
S^3	$[0.2, 0.3]$	$[0.4, 0.5]$	$[0.15, 0.25]$
S^4	$[0.3, 0.4]$	$[0.2, 0.3]$	$[0.15, 0.25]$
S^5	$[0, 0.29]$	$[0, 0.19]$	$[0, 0.49]$

Table 5.1: Probability intervals used for each of the three client types, denoted as C^1, C^2 and C^3. We assume they can connect with five distinct server entities, labelled from S^1 to S^5, in the network.

meaning eight entities in total are present in the network model. For each client type, we assume the gossip rate falls within the range $[0.48, 0.52]$ and for their client profiles we specify the individual server connection probability intervals given in Table 5.1. For example, a client of type C^1 randomly connects with a server of type S^1 with a probability within the range $[0.01, 0.1]$. In these IDTMCs, we still respect our design assumptions where one client and server each possesses recently generated STH data before gossiping begins and that clients can only connect with at most one server per gossip round.

Specifying intervals in a PRISM model so that the verification tool can identify and parse them is straightforward; for a client module *Client* with connect rate probability p, the command for *connect* is written as:

$$\begin{aligned}[connect]\, c_g = 0 \rightarrow [p-\epsilon, p+\epsilon] : (c'_g = 1) + \\ [1-(p+\epsilon), 1-(p-\epsilon)] : (c'_g = 1) \wedge (c'_{skip} = \top)\end{aligned} \tag{5.1}$$

In (5.1), ϵ is the error which we fix at 0.02 to obtain our desired intervals. Next, we

write the *choose* command as:

$$\begin{aligned}
[\mathit{choose}]\, c_{skip} = \bot \wedge c_g = 1 \rightarrow\ & [p_1^L, p_1^H] : (c_g' = 2) \wedge (c_s' = 1) + \\
& [p_2^L, p_2^H] : (c_g' = 2) \wedge (c_s' = 2) + \\
& \vdots \\
& [p_5^L, p_5^H] : (c_g' = 2) \wedge (c_s' = 5); \\
[\mathit{choose}]\, c_{skip} = \top \wedge c_g = 1 \rightarrow\ & (c_g' = 2)
\end{aligned} \tag{5.2}$$

In (5.2), p_j^L/p_j^H are the lower/upper bounds of the server connection probability of the client choosing server S^j, while c_s specifies which server the client is currently connected with. For a set of probability intervals $[l_1, u_1]$, $[l_2, u_2], \ldots, [l_N, u_N]$ in a single command, PRISM checks the following conditions are met:

1) $l_i > 0$ for each $i = 1, \ldots, N$,

2) $l_i \leq u_i$ for each $i = 1, \ldots, N$,

3) $\sum_{i=1}^{N} l_i \leq 1 \leq \sum_{i=1}^{N} u_i$,

Condition 1 is needed to preserve the underlying graph structure of the Markov chain; if there are distributions where the probability of transitioning to certain states is zero, this will affect the qualitative results of the model. In Table 5.1, there are intervals where the lower bound is zero, so to fulfil condition 1 we use a lower bound of 10^{-14}. Condition 3 ensures that legal probability distributions exist during that particular action.

To specify the minimal/maximal reachability probabilities, we can use the **Pmin**/**Pmax** syntax which denotes the minimum/maximum probabilistic path

operators in PRISM respectively:

$$\begin{aligned} &\mathbf{Pmin}_{=?}\,[\,\mathbf{F}^{\leq r}\ \mathit{prop}\,], \\ &\mathbf{Pmax}_{=?}\,[\,\mathbf{F}^{\leq r}\ \mathit{prop}\,]. \end{aligned} \tag{5.3}$$

In (5.3), the queries denote the minimal and maximal probability to reach a set of states satisfying a proposition *prop* within r gossip rounds, respectively.

5.1.2 Methodology and Results

An in Chapter 4, we evaluate both the initial and extended designs of the protocol under both normal circumstances and when a split-world attack is in progress, with one victim client present in the network. The client of type C^3 and the server of type S^1 will have the latest STH data from the log with the former possessing fake data during the split-world scenario. For the normal scenario, we analysed the minimal/maximal likelihood of all clients receiving the latest data during the first twenty rounds. Similarly, for the split-world scenario, we analysed the minimal/maximal likelihood of inconsistency detection. At the time of writing, the tool extension does not support rewards-based IDTMC model checking, meaning protocol efficiency cannot be investigated. Using (5.3), the properties we analyse are:

$$\begin{aligned} &\mathbf{Pmin}_{=?}\,[\,\mathbf{F}^{\leq r}\ \mathit{clients\_all\_updated}\,], \\ &\mathbf{Pmax}_{=?}\,[\,\mathbf{F}^{\leq r}\ \mathit{clients\_all\_updated}\,], \end{aligned}$$

in the normal case and

$$\begin{aligned} &\mathbf{Pmin}_{=?}\,[\,\mathbf{F}^{\leq r}\ \mathit{detect}\,], \\ &\mathbf{Pmax}_{=?}\,[\,\mathbf{F}^{\leq r}\ \mathit{detect}\,], \end{aligned}$$

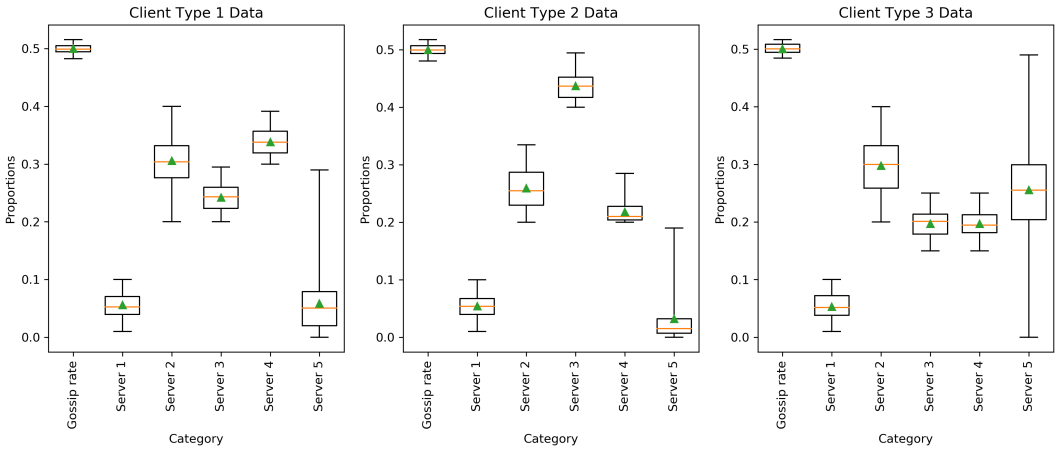


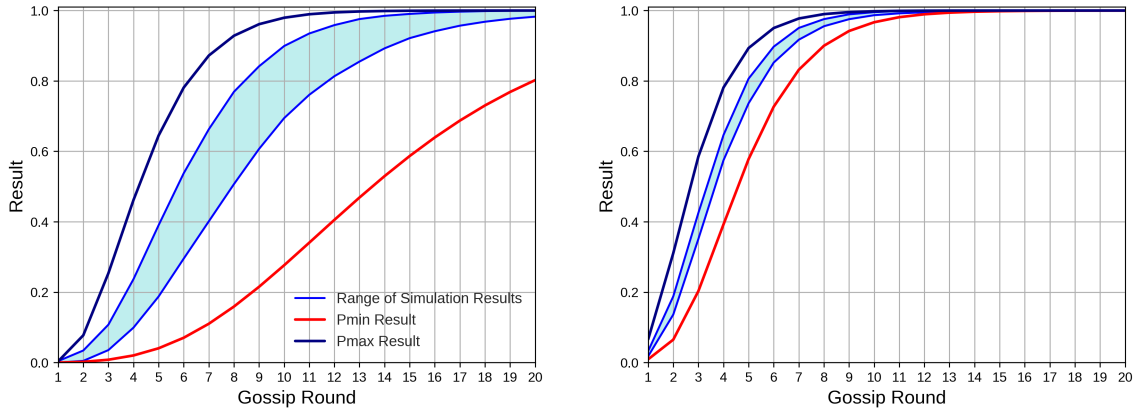
Figure 5.1: Box-and-whisker plots showing the overall range of values for each proportion randomly generated, broken down by client type. The tail ends represent the maximum/minimum values for that category, the orange line and green triangle give the median and the mean value for that data type, respectively. The spread of the data is designed to closely match the intervals used in model checking as much as possible.

	Category											
	Gossip Rate		Server 1		Server 2		Server 3		Server 4		Server 5	
Client Type	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Type C^1	0.482	0.516	0.01	0.1	0.2	0.4	0.2	0.295	0.3	0.392	0.0	0.29
Type C^2	0.48	0.518	0.01	0.1	0.2	0.335	0.4	0.495	0.2	0.285	0.0	0.19
Type C^3	0.484	0.516	0.01	0.1	0.2	0.4	0.15	0.25	0.15	0.25	0.0	0.49

Table 5.2: Maximal/minimal values for each proportion, rounded to three decimal places each.

in the split-world case.

Next, we generate artificial data in such a way that the range of proportions for each category is large enough to try to closely match the size of the intervals used for model checking. We present box-and-whisker plots for this collection of randomly sampled data in Fig. 5.1 and give the maximal/minimal values of each proportion in Table 5.2. We randomly chose samples from this data to construct models for statistical model checking and compared the range of approximated results against our verification results.



(a) Initial design

(b) Extended design

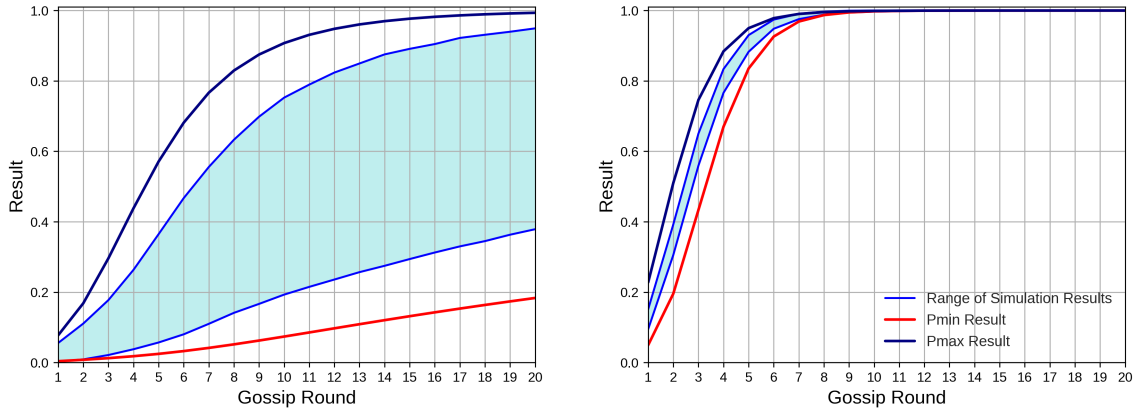
Figure 5.2: IDTMC model checking and simulation results showing the probability of all clients in the network being updated during the first twenty gossip rounds. We compare against the simulation results (blue shaded area), minimal verification result (red line) and maximal verification result (navy line).

Normal Scenario

In Fig. 5.2, for both the initial and extended versions of the protocol, we display the range of statistical results using the randomly sampled data (the shaded blue area) and the exact **Pmin** and **Pmax** reachability results (red and navy line, respectively). We see the statistical results lie neatly between the verification results. In particular, the **Pmin** result in Fig. 5.2(a) shows there are probability distributions which produce even worse results than the randomly sampled data provides, giving an example of IDTMC model checking having advantages over random simulations. In Fig. 5.2(b), the verification results produce tight bounds on the simulation results and show that, after fourteen rounds, it is guaranteed the newer data will be disseminated to all the clients.

Split-world Scenario

Fig. 5.3(a) shows that a wider range of possibilities can happen; in the worst-case, the probability of detection is just under twenty per cent given by the **Pmin** result. Using



(a) Initial design

(b) Extended design

Figure 5.3: IDTMC model checking and simulation results showing the probability of inconsistency detection amongst clients during the first twenty gossip rounds. We compared against the simulation results (blue shaded area), minimal verification result (red line) and maximal verification result (navy line).

the extended protocol design, detection is guaranteed after around twelve rounds. To explain why, since server gossip is independent of client behaviour and the probability of broadcasting to selected servers remains constant, the chances of the client detecting something rapidly increases after several rounds no matter their client profile. There would be little reason to put bounds on server behaviour as the more servers being selected for gossip per round the better the data spread, so the question here is how much liberty can be given to servers when selecting peers before it starts to negatively impact on performance.

5.2 IDTMC Abstraction

As our models become more complex and the IDTMCs increase in size, applying the IDTMC model checking implementation becomes a more expensive task, where an excessive amount of time is needed to complete the process. At the cost of accuracy, IDTMC abstraction can help derive relatively simpler models to perform model checking

on and obtain lower and upper bounds on probabilistic-type properties. This section explains how we abstract a model using a combination of the algorithms described in this section and existing functionalities within the PRISM codebase. To demonstrate our tool, we focus on only abstracting normal scenario models without server gossip.

5.2.1 Dynamic Abstraction using PRISM

To abstract models with PRISM, one possibility is to construct the original, *concrete* model first before then using the state/transition information, whether stored as a data structure or exported to other files, to construct a newer *abstracted* model. However, there are two problems with this approach:

- *Size of the intermediate model* - By trying to build the concrete model first so we can derive the desired ADTMC, there is a risk that the former is too big to build. Even if we build the concrete model, computer resource issues will still persist which will not allow us to carry out the abstraction process.

- *Usability* - Writing code to abstract models in a specific way does not create effective re-usable software which can be used for other purposes as certain variables and functions will have to be hard-coded to solve our problem.

To address the above issues, we developed an algorithm which scans concrete states individually as they are being explored and uses their respective transition information to build a newer probability distribution to construct the abstract model dynamically, by adding newer states/transitions or updating the bounds of existing probability intervals. To describe the procedure, suppose we have an ADTMC $\mathcal{M} = (S, s_\iota, P^l, P^u)$ constructed from the concrete IDTMC model $\mathcal{M}^c = (S^c, s^c_{iota}, \widetilde{P}^l, \widetilde{P}^u)$ after scanning states $s_1, s_2, \ldots, s_{(n-1)} \in S^c$. Assume that $\mathcal{M}^c$ has a unique initial state and has no deadlock states. We wish to update $\mathcal{M}$ with

newer information after exploring state $s_n \in S^c$ with the set of neighbouring states $Next(s_n)$ and the lower/upper probability bound function $\delta^l(s_n)/\delta^u(s_n)$.

Abstract states are supposed to represent a partition of states, meaning to find which partitions s_n and its neighbouring states belong to, we evaluate a set of expressions Exp_{abs}, with each one defining an abstract variable using concrete state information to find the integer or truth value, depending on the context. To put it more formally, $Exp_{abs} = \{F_1, F_2, \ldots, F_N\}$ is a collection of functions which maps a state $s \in S^c$ to a value and is one of two types:

$$F^{\text{Int}} : S^c \to \mathbb{Z}, \text{ or}$$
$$F^{\text{Bool}} : S^c \to \{\top, \bot\}.$$

Updating the ADTMC using s_n, $\delta^l(s_n)$ and $\delta^u(s_n)$ can be broken down into three stages, which we will denote as functions:

i) ***FindAbstractState*** - Using Exp_{abs}, find which abstract state represents s_n as a tuple $s_{abs} = (F_1(s_n), F_2(s_n), \ldots, F_N(s_n))$, where each $F_i \in Exp_{abs}$ for every $i = 1, \ldots, n$.

ii) ***ConstructComponent*** (Algorithm 2) - Next, using $Next(s_n)$, find the set of abstract states $Next_{abs}(s_n)$ (similar to step i)) which neighbour s_n and the lower/upper probability bound function $\delta^l_{abs}(s_n)/\delta^u_{abs}(s_n)$ describes the minimal/maximal probability of transitioning from s_n to each state in $Next_{abs}(s_n)$.

iii) ***UpdateADTMC*** (Algorithm 3) - Lastly, use s_{abs}, $Next_{abs}(s_n)$, $\delta^l_{abs}(s_n)$ and $\delta^u_{abs}(s_n)$ to update $\mathcal{M}$. This may include appending s_{abs} to S if it has not been seen before, adding newer transitions or updating existing bounds.

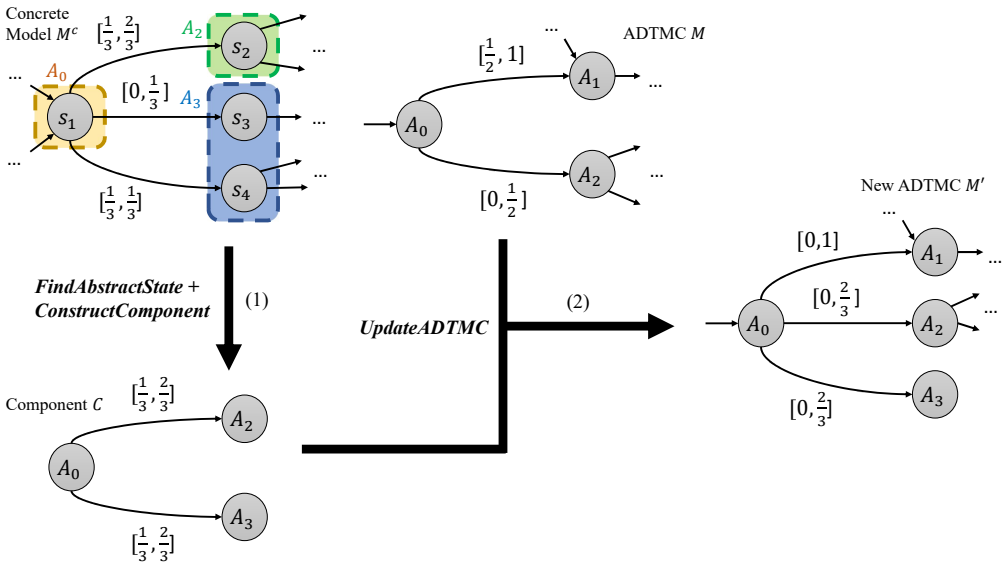


Figure 5.4: Illustration of the three-stage process of (1) parsing state/transition information from a concrete model (***FindAbstractState*** and ***ConstructComponent***), constructing the abstract component and (2) using it to update an ADTMC (***UpdateADTMC***.)

Method i) simply takes as input s_n and performs a loop over Exp_{abs} to evaluate each F_i. For methods ii) and iii), please refer to Appendix A.

Fig.5.4 gives an example of where we have an ADTMC $\mathcal{M}$ and the state and transition information from state s_1 is parsed from the concrete model $\mathcal{M}^c$ to derive a component C, which is then synthesised with $\mathcal{M}$ to produce a newer ADTMC $\mathcal{M}'$. In step (1), if we let $\delta^{abs}(s)(A_i) = [\delta^l_{abs}(s)(A_i), \delta^u_{abs}(s)(A_i)]$ we apply ***FindAbstractState*** and ***ConstructComponent*** to obtain the following:

- $s_1 \in A_0$, $s_2 \in A_2$, $s_3, s_4 \in A_3$, $Next_{abs}(s) = \{A_2, A_3\}$,
- $\delta^{abs}(s)(A_2) = [\frac{1}{3}, \frac{2}{3}]$,
- $\delta^{abs}(s)(A_3) = [0, \frac{1}{3}] + [\frac{1}{3}, \frac{1}{3}] = [0 + \frac{1}{3}, \frac{1}{3} + \frac{1}{3}] = [\frac{1}{3}, \frac{2}{3}]$

Lastly, in step (2), we apply ***UpdateADTMC*** to transform $\mathcal{M}$ by adding or updating information about the abstract model:

- A_3 gets added to $\mathcal{M}$, so $S' = S \cup \{A_3\}$,

Algorithm 2 Constructing the set of adjacent abstract states and the corresponding distribution function after exploring the state and transition information for a concrete state s.

Require: State s, set $Next(s)$ of outgoing states from s in IDTMC $\mathcal{M} = (S, s_\iota, \widetilde{P}^l, \widetilde{P}^u)$, lower/upper probability bound function $\delta^l(s)/\delta^u(s)$, set of expressions for abstract variables Exp_{abs}.

1: **procedure** ConstructComponent(s, $Next(s)$, $\delta^l(s)$, $\delta^u(s)$, Exp_{abs})
2: $Next_{abs}(s) \leftarrow \emptyset$
3: Initialise probability bound functions $\delta^l_{abs}(s)$, $\delta^u_{abs}(s)$
4: **for** $t \in Next(s)$ **do**
5: $t_{abs} \leftarrow$ FindAbstractState(t, Exp_{abs})
6: **if** $t_a \in Next_{abs}(s)$ **then**
7: $\delta^l_{abs}(s)(t_{abs}) \leftarrow \delta^l_{abs}(s)(t_{abs}) + \delta^l(s)(t)$
8: $\delta^u_{abs}(s)(t_{abs}) \leftarrow \min(1, \delta^u_{abs}(s)(t_{abs}) + \delta^u(s)(t))$
9: **else**
10: $Next_{abs}(s) \leftarrow Next_{abs}(s) \cup \{t_{abs}\}$
11: $\delta^l_{abs}(s)(t_{abs}) \leftarrow \widetilde{P}^l(s,t)$
12: $\delta^u_{abs}(s)(t_{abs}) \leftarrow \widetilde{P}^u(s,t)$
13: **end if**
14: **end for**
15: **Return** $Next_{abs}(s_{abs})$, $\delta^l_{abs}(s_{abs})$, $\delta^u_{abs}(s_{abs})$
16: **end procedure**

- $P^u(A_0, A_1) = 1$ (unchanged), $P^l(A_0, A_1) = 0$ (since $A_1 \notin Next_{abs}(s)$),
- $P^u(A_0, A_2) = \max\left(\frac{1}{2}, \frac{2}{3}\right) = \frac{2}{3}$, $P^l(A_0, A_2) = \min\left(0, \frac{1}{3}\right) = 0$,
- $P^u(A_0, A_3) = \delta^u_{abs}(s)(A_3) = \frac{2}{3}$, $P^l(A_0, A_3) = 0$ (since $A_3 \notin Sup(A_0)$),

5.3 Experimental Results

Our implementation of the process previously described involves a hybrid approach, applying the skeleton algorithm of one of PRISM's engines and combining it with original code to build an ADTMC instead, using labels and formulae written within the PRISM file to specify the abstract variables and the set of target states with which

Algorithm 3 Updating an abstract model $\mathcal{M} = (S, s_\iota, P^l, P^u)$ using a state s from concrete IDTMC model $\mathcal{M}^c = (S^c, s^c_\iota, \widetilde{P}^l, \widetilde{P}^u)$. Assume paths in $\mathcal{M}^c$ are all infinite and there is only one initial state.

Require: Concrete state $s \in S^c$, set of neighbouring states $Next(s)$, lower/upper probability bound function $\delta^l(s)/\delta^u(s)$, set of expressions for abstract variables Exp_{abs}.

procedure UPDATEADTMC($\mathcal{M}$, s, $Next(s)$, $\delta^l(s)$, $\delta^u(s)$, Exp_{abs})
  $s_{abs} \leftarrow$ FINDABSTRACTSTATE(s, Exp_{abs})
  $Next_{abs}(s), \delta^l_{abs}(s), \delta^u_{abs}(s) \leftarrow$ CONSTRUCTCOMPONENT(s, $Next(s)$, $\delta^l(s)$, $\delta^u(s)$, Exp_{abs})
  if $s^c_\iota = s$ **then**
    $s_\iota \leftarrow s_{abs}$
  end if
  if $s_{abs} \in S$ **then**
    $Sup(s_{abs}) \leftarrow$ GETSUPPORT($\mathcal{M}$, s_{abs}) ▷ Called via PRISM
    for $t \in Next_{abs}(s_{abs})$ **do**
      if $t \in Sup(s_{abs})$ **then**
        $P^l(s_{abs}, t) \leftarrow \min(P^l(s_{abs}, t), \delta^l_{abs}(s)(t))$
        $P^u(s_{abs}, t) \leftarrow \max(P^u(s_{abs}, t), \delta^u_{abs}(s)(t))$
      else
        $S \leftarrow S \cup \{t\}$
        if $Sup(s_{abs}) \neq \emptyset$ **then**
          $P^l(s_{abs}, t) \leftarrow 0$
        else
          $P^l(s_{abs}, t) \leftarrow \delta^l_{abs}(s)(t)$
        end if
        $P^u(s_{abs}, t) \leftarrow \delta^u_{abs}(s)(t)$
      end if
    end for
    for $u \in Sup(s_{abs}) \setminus Next_{abs}(s)$ **do**
      $P^l(s_{abs}, u) \leftarrow 0$
    end for
  else
    $S \leftarrow S \cup \{s_{abs}\}$
    for $t \in Next_{abs}(s)$ **do**
      $S \leftarrow S \cup \{t\}$
      $P^l(s_{abs}, t) \leftarrow \delta^l_{abs}(s)(t)$
      $P^u(s_{abs}, t) \leftarrow \delta^u_{abs}(s)(t)$
    end for
  end if
end procedure

to perform probabilistic reachability. By keeping track of the concrete states seen so far and the ones yet to be explored, state/transition information is parsed to gain newer information which is then added to the ADTMC model using the functionalities provided by the PRISM codebase.

For the code to recognise abstraction information provided by the PRISM file, we prefix formula names with the string "*abs_*", where one abstraction formula corresponds with one abstract variable. We reserve the formula name *abs_t* which is used to denote the satisfiability condition for an abstract state to be a target state (the code will raise an error if no such formula named *abs_t* exists). As with normal PRISM formulae, we can embed a formula into others using its name. We store the abstract variable names (after stripping the prefix "*abs_*") and their respective formulae separately in vector objects $absVarNames$ and $absVarExps$. When the ADTMC has been successfully constructed using the algorithm explained in the previous section, our code then performs bounded reachability model checking on it automatically; for experimental purposes, given a target set of states specified by *abs_t*, we compute the probabilistic reachability for the first twenty rounds.

5.3.1 Abstracting Normal Scenario Models

To test our implementation and evaluate the effectiveness of the abstraction, we compare the model statistics, build times and verification results between full-scale network models and their abstracted counterparts. Due to the difficulties of building split-world models in PRISM and the inherent limitations of PRISM's explicit engine, we focus on normal scenario network models where no attack is taking place (see Chapter 4). We devise one way that we could abstract our models based solely on remembering whether a client or server has been updated and the global state the clients are in. As we have explained in Chapter 4, each client c in the model has a

Boolean c_{sth}, which is true if the client has the newer STH, and each server s records a Boolean s_{sth} which has a similar function to c_{sth}.

To record the state of a client c in the abstract model, we allow $(c_{sth})^{abs} \in \{\top, \bot\}$ to be a Boolean variable which is false if the client is not updated or true otherwise. For the client to be updated, one of the two conditions must be met:

i) If $c_g \neq 2$, c_{sth} is also true,

ii) If $c_g = 2$ (the *connect* phase), we look at whether c_{sth} is true or, if not, we check whether the client will be updated in the next phase of the round i.e. if $c_s = j$, where $j > 0$, check that s^j_{sth} is true.

Therefore, letting M equal the number of server nodes in the network model, we can express $(c_{sth})^{abs}$ as:

$$(c_{sth})^{abs} = (c_g \neq 2 \wedge c_{sth} = \top) \vee \left(c_g = 2 \wedge \left(c_{sth} = \top \vee \bigvee_{j=1}^{M} \left(c_s = j \wedge s^j_{sth} = \top \right) \right) \right).$$

Next, for a server node s^j in the network model, letting $(s^j_{sth})^{abs} \in \{\bot, \top\}$ also be a Boolean variable in the abstract model which records the update state of s^j. It behaves similarly to $(c_{sth})^{abs}$, however, it can evaluate to true if an updated client chooses to gossip with it in the connect phase. Letting N equal the number of client nodes in the model, we have:

$$(s^j_{sth})^{abs} = \left(c^1_g \neq 2 \wedge s^j_{sth} = \top \right) \vee \left(c^1_g = 2 \wedge \left(s^j_{sth} = \top \vee \bigvee_{i=1}^{N} \left(c^i_s = j \wedge c^i_{sth} = \top \right) \right) \right).$$

Since all clients move synchronously in the original model, it does not matter which c^i_g is used in the previous expression. We also have an integer variable $abs_g \in \mathbb{N} \cup \{0\}$, which records the phase of the gossip round, having a similar purpose to the client's

global state in the concrete model, with the *choose* phase split into four different cases. We describe the values for abs_g and what they represent next:

0 - Start of the gossip round,

1 - *Choose* stage where at least one updated client gossip and all the non-updated ones decide to skip,

2 - *Choose* stage where all updated clients skip the round and at least one non-updated client gossip,

3 - *Choose* stage where at least one updated and one non-updated client gossips ,

4 - *Choose* stage where all clients decide to skip,

5 - *Connect* stage,

6 - *Round Complete* stage,

7 - *END* stage. This is reached when all clients have been updated.

Lastly, we consider an abstract state of the form

$$\big((c^1_{sth})^{abs}, \ldots, (c^M_{sth})^{abs}, (s^1_{sth})^{abs}, \ldots, (s^N_{sth})^{abs}, abs_g\big)$$

to be a target state if $abs_g \geq 6$ (i.e. when the gossip round is completed) and $\bigwedge_{i=1}^{M}(c^i_{sth})^{abs}$ evaluates to true.

5.3.2 Results

We looked at four network models with differing amounts of clients and servers, each with ten entities present. For the gossip rates and client profiles in each category, we generated random intervals with a maximum length of 0.2 and the lower bound being no

N	M	**States**		**Transitions**		**Build Time (s)**		**Total MC Time (s)**	
		original	abstract	original	abstract	original	abstract	original	abstract
4	6	666,864	1664	1,204,464	8458	5.239	7.37	198.438	1.263
5	5	2,185,808	1728	4,051,808	9758	16.979	25.531	733.861	1.454
6	4	4,446,768	1760	8,321,520	10,689	37.455	55.865	1420.524	1.964
7	3	4,747,942	1776	8,876,458	11,616	42.567	163.35	1568.989	1.790

Table 5.3: Model statistics and times for normal scenario models with differing amounts of clients, N, and servers, M, present. Whilst the building times for the abstract model took longer than their concrete counterparts, we also see huge gains in the model checking efficiency. All values are rounded to three decimal places where appropriate.

less than 0.01. We first evaluated the model statistics and build/model checking times of constructing the concrete models and their abstracted counterparts, and afterwards compared the experimental results.

Model Statistics and Times. For our concrete IDTMC models, their sizes ranged from about $667,000$ to 4.7 million states, with the largest number of transitions found nearly reaching 8.9 million. They typically take less time to build than their abstracted versions, yet the total time to perform model checking, the total time to compute the minimal *and* maximal reachability probabilities for the first twenty gossip rounds, took 25 minutes to finish. The number of states built for the abstract models did not go into the tens of thousands; this is expected as, there are at most $2^N \cdot 2^M \cdot 8 = 2^{10} \cdot 8 = 8192$ possible states. Due to these models being small, verification took at most two seconds to finish. In each of the four cases investigated, it was more efficient to analyse the abstracted models. Table 5.3 displays the full set of results.

Verification Results. Next, we compare the results obtained between the original, concrete IDTMCs and their ADTMC counterparts by finding the minimal and maximal reachability probabilities of reaching the set of targets states where all clients have newer data. Since we are performing verification on concrete models where the probability interval ranges were made to be small, we would expect there to be tight lower and

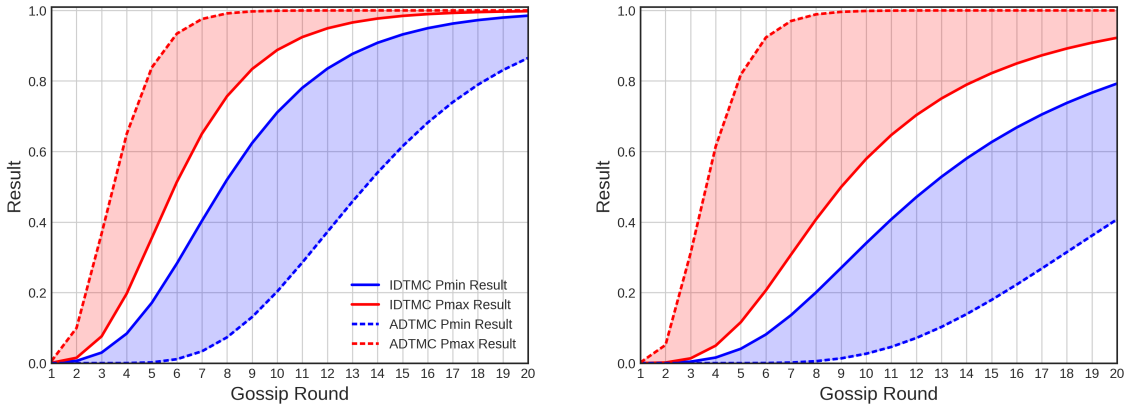
upper bounds. The ADTMCs will produce weaker bounds on the same properties, since this is the trade-off for obscuring information concerning certain concrete variables. The reliability of these bounds also depends on the difference in the sizes of the original and abstract models; since the latter models are small, the bounds deviate when compared against the results of concrete models with millions of states.

With four clients and six servers present in the model, the minimal (**Pmin**) and maximal (**Pmax**) results obtained from the ADTMC, as we can see in Fig.5.5(a), appear to be close to the IDTMC results, whereas with seven clients and three servers (Fig.5.5(d)) the bounds are coarser, yet not too much as to be deemed impractical. From an investigator's perspective, to judge the effectiveness of the gossip mechanism, the **Pmin** ADTMC results provide a reliable limit on the worst-case behaviour without deviating too much from the IDTMC results.

5.3.3 Limitations

We have shown how our code can abstract PRISM models dynamically and can handle millions of states whilst still being general enough to handle similar problems involving abstraction. However, we must acknowledge the limitations of our implementation.

Firstly, the code is not optimised and still needs a sizeable amount of memory to perform the abstraction. During our investigations, we found that PRISM's explicit engine was more efficient in building models compared to our tool extension, despite allocating 12GB of memory to the heap space in each case. Secondly, the results we obtained from our ADTMCs are only as good as the way we choose to abstract the concrete model, and there are many possibilities of doing this which will require multiple experiments to get right. To make the bound of the concrete and abstract results more contiguous, one solution involves adding more abstract variables to preserve certain information about the original models. Note the following caveat: the increase in the



(a) 4 Clients, 6 Servers

(b) 5 Clients, 5 Servers

(c) 6 Clients, 4 Servers

(d) 7 Clients, 3 Servers

Figure 5.5: Verification results for our concrete (IDTMC) and abstracted (ADTMC) models. In each case, we compared the minimal (**Pmin**, blue lines) and maximal (**Pmax**, red lines) results found for both versions of the network model.

number of states may be exponential and would not scale well if more clients get added to the models. It is also unknown if adding more complex expressions in the PRISM file can affect the tool extension efficiency.

To expand the capabilities of our implementation, since we have also added support for DTMCs, it may be a useful feature to include support for three-valued abstraction for PCTL properties, as it can be shown that the type of abstraction we used conserves qualitative verification results [119]. Future work can also include adding support for CTMC or MDP abstraction by implementing other techniques; see

for example *magnifying-lens abstraction* [56] and *game-based abstraction* [173].

5.4 Summary

In this chapter, to accommodate for any uncertainties in the transition probabilities, we build and verify IDTMCs using an extension of PRISM. Since IDTMCs are non-deterministic models, we can produce PCTL results showing the best- and worst-case outcomes for protocol performances and this is more effective than performing a random search. Since IDTMCs are more costly to perform verification on than DTMCs, we show how to ease this through abstraction, observing how there is a trade-off between efficiency and accuracy. Our implementation has the benefit of being more general purpose and uses existing PRISM functionalities, building the abstraction of the IDTMC on-the-fly rather than trying to build the larger, concrete model first. In the next chapter, building upon accommodating for uncertainty, we introduce a method which allows us to search over other uncertain parameters other than probabilities, allowing for more thorough analysis.

Chapter 6

Model Parameter Optimisation

In the previous chapter, IDTMCs provided a powerful way to capture probabilistic uncertainties within our models. However, they are limited in some respect since the uncertainties are only on the transition probabilities. This means we still need to keep certain parameters fixed, such as the number of client nodes of each type. In this section, by applying sequential-based model optimisation (SMBO) algorithms, we describe the process of how we could intelligently explore a mixture of both discrete and continuous parameters which define our models where, after a pre-determined number of trials in which different parameters are chosen by a 'black-box' optimiser to configure the network model, a set of parameters is suggested that produces a bad scenario which minimises a PCTL property we are interested in.

6.1 Deriving Network Model Parameters

This section describes the model space for our DTMC network models which, given a fixed network topology, contains the set of possible configurations that consists of the initial state, the client type probabilities and the frequency of each client type in the model. For a network topology NT, let $\mathcal{M}_{NT}$ be a DTMC which models the gossiping

that occurs in a sequence of rounds until a condition is met. If NT has N client types and M server types, we list these types as $C^1, C^2, \ldots, C^N$ and $S^1, S^2, \ldots, S^M$ respectively.

6.1.1 Client Type Frequency

In $\mathcal{M}_{NT}$, let $N' \in \mathbb{N}$ be the total number of client nodes in the model. Assuming each client node in $\mathcal{M}_{NT}$ belongs to a unique type, and at least one client of each type is always present, let $f^k \in \mathbb{N}$ denote the number of clients of type k in $\mathcal{M}_{NT}$ such that $\sum_{k=1}^{N} f^k = N'$. We define the *client type frequency* by the N-dimensional vector $\underline{\mathbf{F}} = (f^1, f^2, \ldots, f^N)$.

6.1.2 Set of Initial States

Our models must initialise the network state by assigning STH data to each client and server before gossiping begins, for which there are many possibilities. As before, we keep to our design assumptions whereby, in the initial state, only one client and one server node initially have the latest STH data, with the rest of the nodes having old STH data.

Given an initial setup of $\mathcal{M}_{NT}$, we define the initial local states for the clients and servers separately. Let $\left(C^1_{init}, C^2_{init}, \ldots, C^N_{init}\right)$ be an N-dimensional vector where $C^k_{init} = 1$ means exactly one client of type C^k has the newest data (or, with split-world, is treated as the victim with the fake data), and $C^k_{init} = 0$ otherwise. Due to our design assumptions, it is necessary the condition $\sum_{k=1}^{N} C^k_{init} = 1$ holds. Similarly, let $\left(S^1_{init}, S^2_{init}, \ldots, S^M_{init}\right)$ be an M-dimensional vector where $S^j_{init} = 1$ means a server of type S^j has the latest data and $S^j_{init} = 0$ if otherwise, with $\sum_{j=1}^{M} S^j_{init} = 1$. Denoting the client and server initial state vectors we previously described as U and V respectively, we let the pair (U, V) describe the unique initial state of $\mathcal{M}_{NT}$. To describe all the

initial states $\mathcal{M}_{NT}$ may have, let $\mathbf{I}_{\mathcal{C}} \subseteq \{0,1\}^N$ and $\mathbf{I}_{\mathcal{S}} \subseteq \{0,1\}^M$ be the set of all possible initial states for the client and server nodes, respectively:

$$\mathbf{I}_{\mathcal{C}} = \left\{ \left(C^1_{init}, C^2_{init}, \ldots, C^N_{init}\right) \,\middle|\, C^k_{init} \in \{0,1\} \text{ for every } k \text{ and } \sum_{k=1}^{N} C^k_{init} = 1 \right\},$$
$$\mathbf{I}_{\mathcal{S}} = \left\{ \left(S^1_{init}, S^2_{init}, \ldots, S^M_{init}\right) \,\middle|\, S^j_{init} \in \{0,1\} \text{ for every } j \text{ and } \sum_{j=1}^{M} S^j_{init} = 1 \right\}.$$

Lastly, we define the *set of possible initial states* for $\mathcal{M}_{NT}$ to be the product set $\underline{\mathbf{I}} = \mathbf{I}_{\mathcal{C}} \times \mathbf{I}_{\mathcal{S}}$.

6.1.3 Probabilities for Each Client Type

The gossiping rates for the client types can be represented by the vector $\underline{\mathbf{G}} = (g^1, g^2, \ldots, g^N)$, where $0 < g^k \leq 1$ for every $k = 1, 2, \ldots, N$. For the server connection probabilities, for a client type C^k and server type S^j, let $I^k_j = [\underline{p}^k_j, \overline{p}^k_j] \subseteq [0,1]$ be the probability interval which specifies the range of values for probability $p^k_j = \mathcal{P}(C^k, S^j)$, and let $\mathcal{I}^k = \left(I^k_1, I^k_1, \ldots, I^k_M\right)$ be a tuple which contains the intervals for each server connection probability for C^k, where the following condition must hold for probability distributions to exist:

$$\sum_{j=1}^{M} \underline{p}^k_j \leq 1 \leq \sum_{j=1}^{M} \overline{p}^k_j. \tag{6.1}$$

If the condition described in expression (6.1) does get violated e.g. if we have $\sum_{j=1}^{M} \overline{p}^k_j < 1$, then we could not derive a valid distribution since the total sum would always equal less than one. Similarly, if $\sum_{j=1}^{M} \underline{p}^k_j > 1$, the total sum will always be greater than one.

To make sure we do not initially choose values in I^k_1 where no probability

Algorithm 4 Deriving a probability distribution for client type c^k with surrogate choice parameters $\mathbf{x} = (x_1^k, \ldots, x_{M-1}^k)$ and tuple $\mathcal{I}^k = \left(I_1^k, I_2^k, \ldots, I_{(M-1)}^k, I_M^k\right)$

Require: For every $1 \leq j \leq M$ and $I_j^k = [\underline{p}_j^k, \overline{p}_j^k]$, $\sum_{j=1}^M \underline{p}_j^k \leq 1 \leq \sum_{j=1}^M \overline{p}_j^k$.

procedure DERIVEDISTRIBUTION($\mathbf{x}, \mathcal{I}^k$)
  for $j = 1 \ldots (M-1)$ **do**
    $p_j^k = \underline{p}_j^k + x_j^k \cdot (\overline{p}_j^k - \underline{p}_j^k)$
    $I_j^k \leftarrow p_j^k$
    for $\beta = (j+1) \ldots M$ **do**
      $\underline{p}_{(\beta,j)}^k \leftarrow \max\left(\underline{p}_{(\beta,j-1)}^k, 1 - \sum_{l=1}^j p_l^k - \sum_{\substack{(j+1)\leq\beta'\leq M \\ \beta'\neq\beta}} \overline{p}_{(\beta',j-1)}^k\right)$
      $\overline{p}_{(\beta,j)}^k \leftarrow \min\left(\overline{p}_{(\beta,j-1)}^k, 1 - \sum_{l=1}^j p_l^k - \sum_{\substack{(j+1)\leq\beta'\leq M \\ \beta'\neq\beta}} \underline{p}_{(\beta',j-1)}^k\right)$
      $I_\beta^k \leftarrow \left[\underline{p}_{(\beta,j)}^k, \overline{p}_{(\beta,j)}^k\right]$
    end for
  end for
  $p_M^k = 1 - \sum_{j=1}^{M-1} p_j$
  $I_M^k \leftarrow p_m^k$
  return $\mathcal{I}^k$
end procedure

distributions exist, we shall assume that $\mathcal{I}^k$ is a *delimited*, meaning that for every $1 \leq j \leq M$ and $p_j^k \in I_j^k$, there exists a probability distribution $\delta = (p_1^k, \ldots, p_j^k, \ldots, p_M^k)$.

To derive a probability distribution from $\mathcal{I}^k$, we devise an algorithm (see Algorithm 4) which applies normalisation [119] a finite number of times after selecting values from $I_1^k, I_2^k, \ldots, I_{M-1}^k$, in that order, finally deriving a value from I_M^k automatically. To do this, we define a set of surrogate parameters $x_j^k \in [0, 1]$ which each describe how far we must go along each truncated interval $\widetilde{I}_j^k \subseteq I_j^k$ derived from normalisation. For more details, please refer to Appendix B. We define $\underline{\mathbf{x}}^k = (x_1^k, \ldots, x_{M-1}^k)$ to be the surrogate choice parameter vector for client type C^k. Note that our algorithm is deterministic since we iterate through $\mathcal{I}^k$ and $\underline{\mathbf{x}}^k$ simultaneously in a fixed order.

6.1.4 The Modelling Space

To end this section, fixing tuples $\mathcal{I}^1, \ldots, \mathcal{I}^N$ which states the set of probability intervals for each client type, we denote χ to be the set of all configurations for $\mathcal{M}_{NT}$ of the form $(\underline{\mathbf{F}}, \underline{\mathbf{I}}, \underline{\mathbf{G}}, \underline{\mathbf{X}})$, called the *modelling space for* $\mathcal{M}_{NT}$, where:

- $\underline{\mathbf{F}}$ is the client type frequency vector,
- $\underline{\mathbf{I}}$ is the set describing all initial states,
- $\underline{\mathbf{G}}$ is the vector which lists the gossiping rate for each client,
- $\underline{\mathbf{X}} = [\underline{\mathbf{x}}^1, \ldots, \underline{\mathbf{x}}^N]^T$ is a $N \times (M-1)$ matrix which specifies the surrogate choice parameter vectors for each client type.

6.2 Adapting the Black-box Optimisation Problem

Analogous to the black-box problem optimisation problem (see Chapter 3, Problem 3.4.1), we define the problem we want to solve as follows:

Problem 6.2.1. (Black-box optimisation problem for model checking quantitative PCTL properties) *Let χ be the modelling space, ϕ a quantitative PCTL property and $F_\phi : \chi \to \mathbb{R}$ a function which constructs a model according to a configuration $x \in \chi$, performs verification on it using ϕ and outputs a real value. Suggest a point $x^* \in \chi$ which best minimises F_ϕ. Due to constraints, F_ϕ can only be called a finite number of times.*

To address the above problem, we apply SMBO introduced in Chapter 3, Section 3.4. To recap, SMBO seeks to find any trends of behaviour in the objective function we wish to optimise by using a statistical model which is relatively cheap to evaluate and is continuously re-adjusted through a series of inputs/output pairings, combining

Algorithm 5 F_{ϕ_P} for PRISM property ϕ_P.

Require: Characterisation vector $x = (\underline{\mathbf{F}}, \underline{\mathbf{I}}, \underline{\mathbf{G}}, [\underline{\mathbf{x}}^1, \ldots, \underline{\mathbf{x}}^N]^T) \in \chi$, fixed interval sets $\mathcal{I}^1, \ldots, \mathcal{I}^N$, ϕ_P is a quantitative property

1: **procedure** $F_{\phi_P}(x)$
2: **for** $k = 1 \ldots N$ **do**
3: $\delta^k \leftarrow$ DERIVEDISTRIBUTION$(\mathbf{x}^k, \mathcal{I}^k)$ ▷ See Algorithm 4
4: **end for**
5: Construct a PRISM model according to $x, \delta^1, \ldots, \delta^N$
6: Verify for property ϕ_P to obtain a result r'
7: Derive a value $r \in \mathbb{R}$ using r'
8: **return** r
9: **end procedure**

it with intelligent searching of χ to decide which input to evaluate next. By combining this algorithm with the capabilities of PRISM, we wish to construct a series of DTMC models and find the configuration which best optimises a metric related to ϕ. We define a function F_{ϕ_P}, where ϕ_P denotes ϕ written in the PRISM syntax, taking as input a characterisation vector $x \in \chi$. For each client type, we derive probability distributions according to their respective interval sets and surrogate choice parameters (See Algorithm 4). Next, we automatically construct a PRISM model and perform verification on it with ϕ_P. Lastly, we output a value $r \in \mathbb{R}$ by performing operations which use the verification result of ϕ_P. Algorithm 5 summarises this process.

6.3 Python Application

We present our Python application which uses optimisation libraries designed for SMBO, namely Hyperopt and Benderopt (see Chapter 3, Section 3.4.3), to find suggested parameters for our network model which optimises F_{ϕ_P}. We explain the workflow of the code, including the range of options available given by the user before

running it, and the different components of the application.

6.3.1 Workflow

Using the default options, if we let the trial data file path and the PRISM property file path be *trial_file* and *property_file* respectively, we can run the application using the command:

`python smbo.py -t` ⟨*trial_file*⟩ `-props` ⟨*property_file*⟩

To tell the application how to run, the user is provided with a wealth of options:

- **Use Hyperopt** (*-hopt*) - Use the Hyperopt library. The default state is to use the Benderopt library. We allow this option so we can compare the performance between both libraries (see Section 6.4).
- **Evaluations** (*-e*) - Specifies the number of trials to be done. Must be at least one. Default value is fifty.
- **Split-world models** (*-s*) - Tells the application we want to analyse split-world models instead of normal models. The default setting is to analyse normal models.
- **Database file** (*-db*) - The SQLite database to write experiment data to for further analysis. If no database exists, the application will automatically create one with the file name *text_db.sqlite*.
- **Property file** (*-props*) - Which property file to use when calling PRISM. This option is always required. An error is raised if no such file exists or any properties cannot be correctly parsed.

- **Trial data file** (*-t*) - The file path used to import/export formatted experiment data. This is required by the user and the contents must be in the correct format depending on the optimiser being used. The number of samples present in this file must not exceed the number of evaluations given by the *-e* option. If no such file exists, the code will create and write data after the experiment has finished as a dictionary formatted for the optimiser. Otherwise, the file will be overwritten with both the old and newer data.

- **Property number to evaluate** (*-p*) - Specify which property to use from the chosen property file. The default is to use the first property listed.

- **Use multiple initial state models** (*-mulinit*) - Analyse models with multiple initial states instead of single initial state models, considering all initial setups which respect our design principles.

- **Apply symmetry reduction** (*-symm*) - Make PRISM use symmetry reduction during the model building process (see Chapter 3). The application will automatically find the equivalent modules to collapse.

Tab. 6.1 summarises the list of arguments described. Next, we define the parameter space which characterises all network models by calling the *HyperoptParameters* class or *BenderoptParameters* class depending on the optimiser used. If *mulinit* is true, we use a 'dummy value' as model checking is performed over all possible initial setups and is equivalent to searching over $N' \cdot M$ inputs from χ simultaneously.

We fixed the gossiping rate for each client type to be 0.5 because increasing this probability will improve the dissemination/detection rate properties. The variables that we vary are:

i) *Client count* - A list that includes all allowable vectors for **F**.

Argument	Option	Type	Description	Default value
Use Hyperopt library	*-hopt*	`bool`	Use the Hyperopt library when optimising. If false, use the Benderopt library.	`False`
Evaluations	*-e*	`int`	The number of evaluations to do per iteration. Must be at least one.	50
Use split-world models	*-s*	`bool`	Perform model checking on split-world type models. If false, use normal type models.	`False`
Database	*-db*	`str`	Which SQLite database to write data to.	*test_db.sqlite*
Property file	*-prop*	`str`	Name of PRISM property file to use. Mandatory.	N/A
Trial data file	*-t*	`str`	Name of trial data file import or export data. Mandatory.	N/A
Property number	*-p*	`int`	Specify which property to evaluate. We analyse the first listed property by default.	1
Use multiple initial states	*-mulinit*	`bool`	Perform verification on multiple initial state models. If false, use single initial state models.	`False`
Use symmetry reduction	*-symm*	`bool`	Apply symmetry reduction during the model building process.	`False`

Table 6.1: List of options for the Python application.

ii) *Initial client states* - A list that includes all allowable vectors for $\mathbf{I}_{\mathcal{C}}$.

iii) *Initial server states* - A list that includes all allowable vectors for $\mathbf{I}_{\mathcal{S}}$.

iv) x_j^k - The surrogate choice value for client type C^k when deriving the probability of connecting with server type S^j. For server S^M, no such parameter exists since the corresponding probability will be automatically derived.

i), ii) and iii) are treated as discrete parameters and vi) a continuous uniform parameter, chosen within the interval $[0, 1)$ (by design, both Hyperopt and Benderopt always choose values strictly less than the upper bound).

Lastly, we need to specify what the objective function is before any optimising can begin. Depending on the library used, we call either the *benderopt* or *hyperopt* function, both of which sanitise the values for each parameter and feeds them into a wrapper *objective_function* which acts as the objective function. For a general

execution of *objective_function*, firstly it generates the probability distribution for each client type, invoking either the *HonestModels* or *SplitworldModels* class, depending on the user's preference, to construct the strings necessary to write the PRISM models. A *PRISMData* object is then created, designed to call PRISM using command-line tools, parameterised by user preference; for example, if the `-symm` flag was set, then the *-symm* option will be included in the command to call PRISM with the correct values to perform symmetry reduction. To obtain a verification result for ϕ_P, we call the *perform_verification* method from *PRISMData* and import it from a results file using the *extract_result* method. To accommodate for a range of results which may be obtained using multiple initial states, *extract_result* returns the result as a list from which the value can be extracted. Lastly, after performing further operations, a real value is returned. In our implementation, the verification result and corresponding model parameters are saved into an SQLite database for statistical analysis.

To perform the optimisation process, we either use *fmin* from the Hyperopt library or a customised *minimize* function using the Benderopt library which takes as an input the objective function, the search space, the file path to import/export trial data and the total number of evaluations we wish to complete. The latter is calculated as the difference between the desired number of evaluations and the number of imported trial data samples. For example, if the user inputs 100 evaluations and imports 20 samples from a previous experiment, then 80 trial runs will be conducted. If no such trial data exists, the code will perform the total number of evaluations. After this process ends, the code prints out the sample found which best minimises the objective function and saves trial data to the file path.

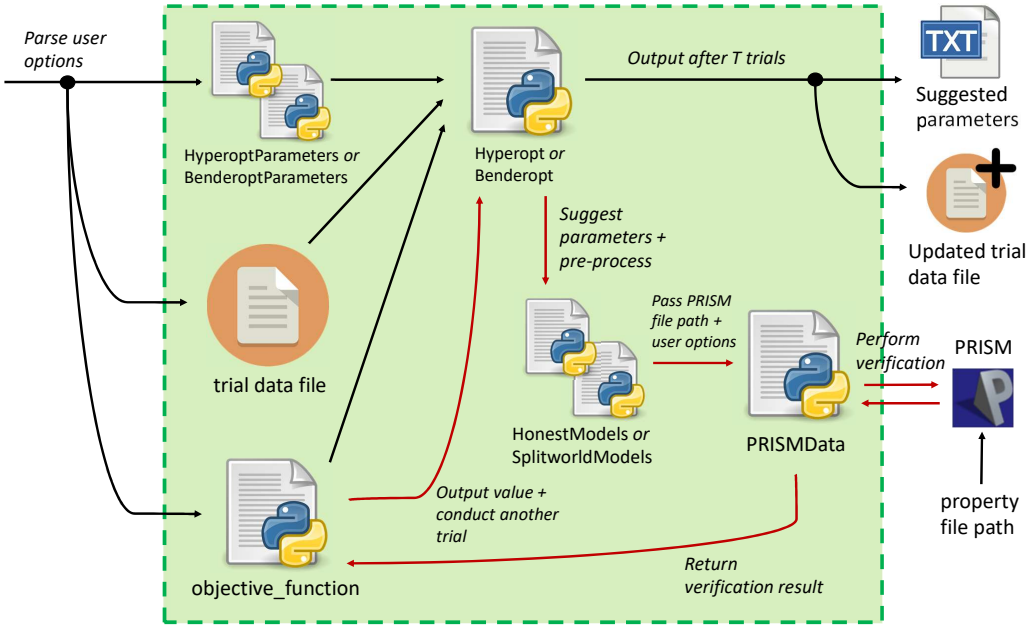


Figure 6.1: Overview of the workflow of the optimiser code. Users parse options to control the execution of the code. A minimise function is called from the chosen optimiser library, passing the number of trials to perform, the objective function, the file path to import/export trial data from previous experiments and the parameter space. For each trial, the optimiser suggests a newer set of parameters to use and generates the PRISM file. The user must supply the file path of the property file and the number of the property to analyse. After extracting the verification result, the objective function uses it to return a value to be fed back into the optimiser. After the maximum number of trials has been performed, the code outputs the best sample obtained to a text file and an saves all trial data to the given file path.

6.4 Experimental Results

To demonstrate our application, we perform optimisation over a given model space and find the suggested setup, which gives the worst case result for properties considered important; for the normal models, this is the expected number of rounds to spread the latest STH data to all clients; for the split-world models, this is the expected number of rounds until detection first occurs:

$$\begin{aligned} &\mathbf{R}^{rounds}_{=?}[\mathbf{F}\ \mathit{clients\_all\_updated}], \\ &\mathbf{R}^{rounds}_{=?}[\mathbf{F}\ \mathit{detect}]. \end{aligned} \tag{6.2}$$

We preserve our network topology to have three client types C^1, C^2 and C^3, and five server types, $S^1, \ldots, S^5$, using the server connection probability intervals as shown in Tab. 6.2. Our network models will have five client and server nodes each, with each server being of a unique type. After modifying the behaviour of the objective function to determine which models to construct and the result to output via a set of options (see Tab. 6.1 for details), it outputs the negation of the largest result found after verifying the PCTL property so we can maximise the quantitative results shown in (6.2). We focus on models where only client-to-server gossip is used; we reason that if server-to-server gossip can be shown to improve the quantitative aspects of a protocol's design in the worst case, then it can serve as a useful indicator for what we can expect for a wide range of networking scenarios allowed by our model space.

6.4.1 Suggested Parameters Which Produce Negative Outcomes

As in Chapter 4, we study four protocol variants and use both model checking and statistical methods to evaluate their respective performance over time by analysing

Server type \ Client type	C^1	C^2	C^3
S^1	$[0.3, 0.4]$	$[0.2, 0.25]$	$[0.02, 0.1]$
S^2	$[0.04, 0.3]$	$[0.2, 0.25]$	$[0.05, 0.3]$
S^3	$[0.02, 0.2]$	$[0.05, 0.1]$	$[0.03, 0.3]$
S^4	$[0.02, 0.3]$	$[0.05, 0.25]$	$[0.1, 0.3]$
S^5	$[0.07, 0.15]$	$[0.3, 0.35]$	$[0.2, 0.4]$

Table 6.2: Probability intervals used for each of our three client types to search over, denoted as C^1, C^2 and C^3. We assume they can connect with five distinct server entities, labelled from S^1 to S^5, in the network.

the same properties as before: data dissemination, protocol efficiency and detection rate. For both the normal and split-world cases, we use different objective functions to find parameters which maximise the PCTL properties in (6.2). Tab. 6.3 provides information on the suggested parameters which give the best result found when searching the modelling space using single and multiple initial state models.

We perform experiments with both optimiser libraries and compared their outputs for different models. For experiments using single initial state models, we perform 200 trials to ensure the search is thorough. For multiple initial state models, we only perform 100 trials due to the excessive time it takes to analyse such models. We give our findings in Fig. 6.2. In summary, searching over models with multiple initial states is slightly more effective in finding the best results compared to using single initial state models. In the normal case, the Benderopt library gave the best results whilst for the split-world case, Hyperopt gave slightly better results. We note that, while the results look more promising when searching over multiple initial state models, there is a trade-off in the time to perform the required number of trials (on average, for both libraries, twelve hours was needed to perform a batch of 25 trials).

(a)

	Best suggestion	
Parameter	One initial state	Multiple initial states
Client type frequency	C^1: 3, C^2: 1, C^3: 1	C^1: 3, C^2: 1, C^3: 1
Initial setup	Client of type C^3 and server S^3 have the newest data	N/A
C^1 distribution	$(0.367, 0.257, 0.027, 0.269, 0.08)$	$(0.399, 0.279, 0.021, 0.21, 0.091)$
C^2 distribution	$(0.214, 0.228, 0.061, 0.174, 0.323)$	$(0.235, 0.209, 0.09, 0.12, 0.346)$
C^3 distribution	$(0.04, 0.099, 0.294, 0.253, 0.314)$	$(0.05, 0.126, 0.282, 0.202, 0.34)$
Result	9.851	9.855

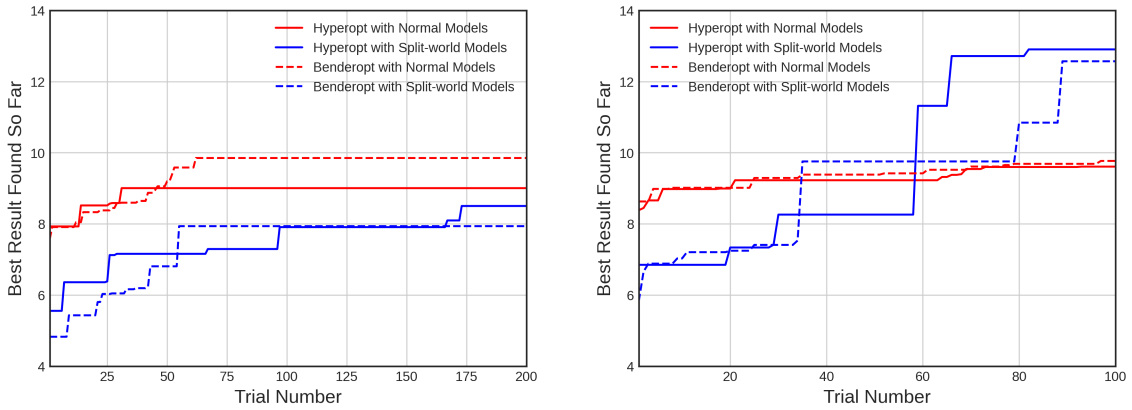
(b)

	Best suggestion	
Parameter	One initial state	Multiple initial states
Client type frequency	C^1: 1, C^2: 3, C^3: 1	C^1: 3, C^2: 1, C^3: 1
Initial setup	Client of type C^1 has fake data; Server S^3 has real data	N/A
C^1 distribution	$(0.334, 0.207, 0.035, 0.277, 0.147)$	$(0.368, 0.239, 0.023, 0.23, 0.139)$
C^2 distribution	$(0.248, 0.226, 0.063, 0.113, 0.35)$	$(0.225, 0.231, 0.066, 0.134, 0.344)$
C^3 distribution	$(0.055, 0.217, 0.077, 0.252, 0.398)$	$(0.061, 0.27, 0.035, 0.286, 0.347)$
Result	8.503	12.905

Table 6.3: The model parameters suggested for maximising (a) the expected number of rounds for all client nodes to update themselves with newer data in the normal case and (b) the expected number of rounds until client detection occurs in the split-world case. We also distinguish between the parameters found when using single and multiple initial state models to search over the modelling space.

6.4.2 Model Checking and Statistical Results Using Suggested Parameters

Using parameters obtained from searching over single initial state models (see Tab. 6.2), to look at how the protocols perform over time, we analyse for properties such as detection and data spread, as done in previous chapters. Inspecting the verification results, it appears the overall trends in the data are similar to those observed for our previous experiments regarding data dissemination (Fig. 6.3(a)), detection rates (Fig. 6.4(a)) and protocol efficiency (Fig. 6.3(b) and Fig. 6.4(b)). However, when looking at the proportion of clients having each STH data type in the split-world case, the clients are much more likely to possess fake data without server-to-server gossip,



(a) Using single initial state models

(b) Using multiple initial state models

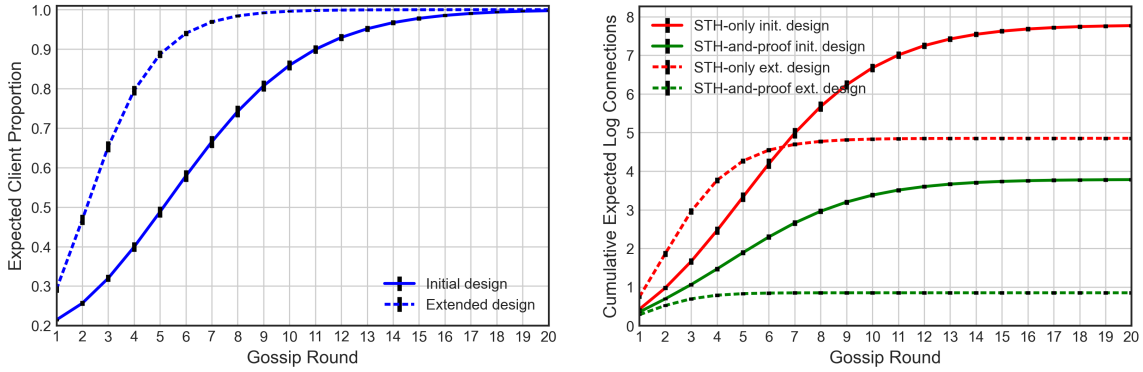
Figure 6.2: The best result found for our chosen properties after t trials using the optimiser libraries Hyperopt and Benderopt. We distinguish between searching over (a) single initial state models and (b) multiple initial state models.

	Client type frequency		
# Clients	C^1 type	C^2 type	C^3 type
50	10	10	30
100	20	20	60
200	40	40	120

Table 6.4: The number of clients of each type for the differently sized networks used for statistical model checking according to the suggested parameters used.

indicating fake data must reach further into a network before it can be detected, which should be avoided (Fig. 6.4(c)). With server-to-server gossip applied, a larger proportion of clients possess the real STH data (Fig. 6.4(d)).

Preserving the ratio of the client types present in the network and the rest of parameters used in Tab. 6.3, we approximate results for the same properties in larger networks using PRISM's discrete simulator, applying the CI method to generate 99% confidence intervals after sampling $2,000$ paths through the model. While we see some of our findings remain consistent from Chapter 4 (Fig. 6.5(a), (b) and (c)), we notice that server-to-server gossip *reduces* the efficiency of the STH-only protocol in the split-world case (Fig. 6.5(d)). As shown in Fig. 6.5(e), with the initial protocol design, more clients are needed to keep the proportion of them having fake data lower,

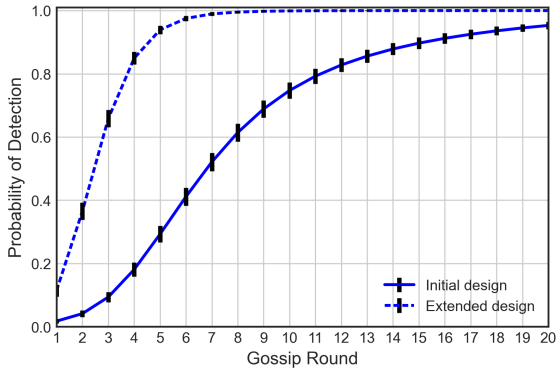


(a) Data dissemination

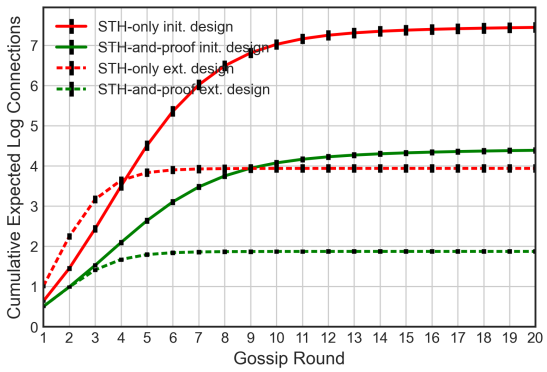
(b) Log connections

Figure 6.3: Verification results for normal models using the suggested parameters found which optimise our properties.

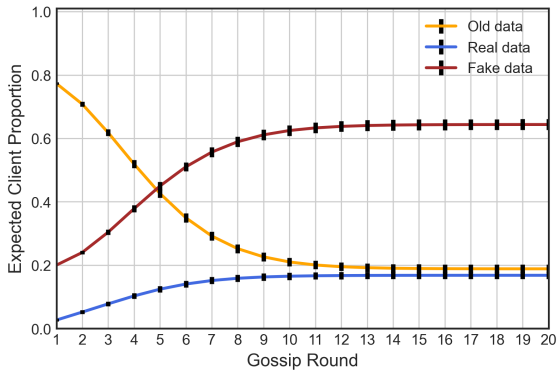
yet this metric remains constant when using the extended protocol designs.



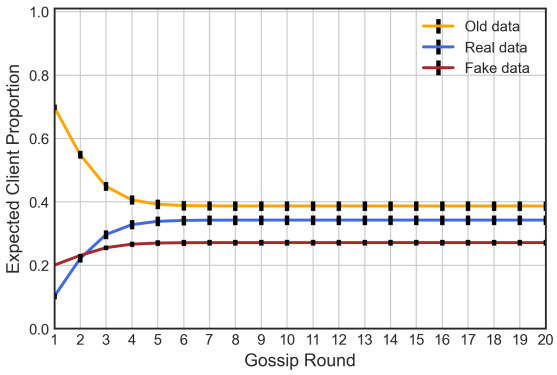
(a) Detection rate



(b) Log connections

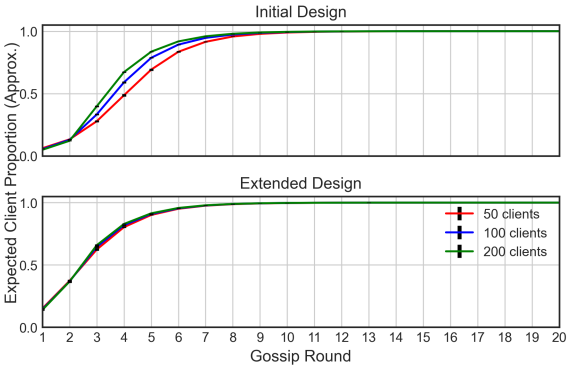


(c) STH distribution, initial design

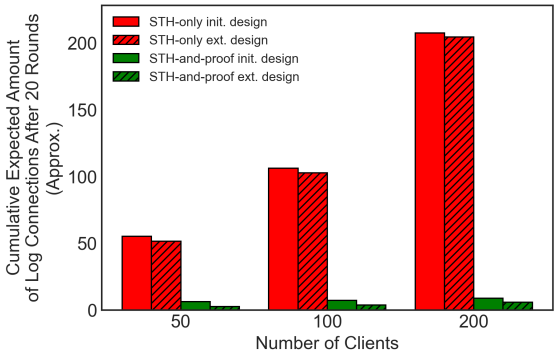


(d) STH distribution, extended design

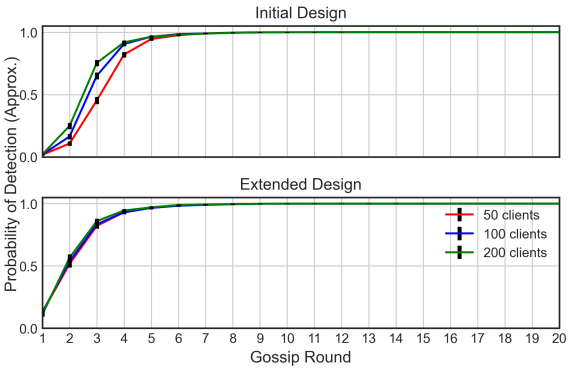
Figure 6.4: Verification results for split-world models using the suggested parameters found that optimise our properties.



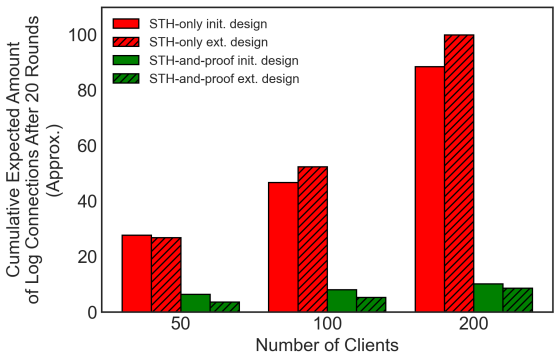
(a) Data dissemination (honest)



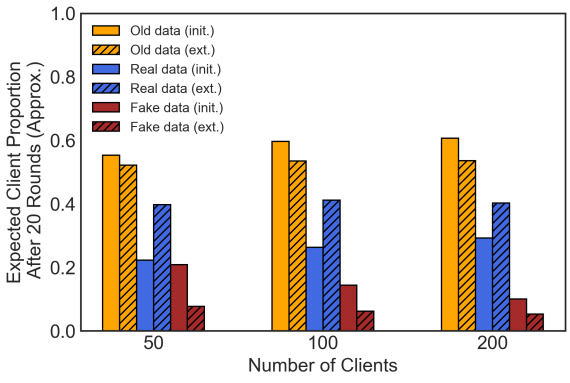
(b) Log connections (honest)



(c) Detection rate (split-world)



(d) Log connections (split-world)



(e) STH type distribution (split-world)

Figure 6.5: Statistical model checking results for our protocol variants using 50, 100 and 200 clients. We preserve the ratio of each client type when increasing the number of client nodes and preserving the probabilities and initial conditions used in our models for verification. Adding server-to-server gossip improves the data spread and detection (see (a) and (c)) but, in the case of measuring the expected cumulative amount of the log connections for the *STH-only* extended design in the split-world scenario, it does not scale well compared to the initial design, as shown by (d). The extended design also aids in the distribution of the real data whilst preventing the spread of the fake data (graph (e)).

6.4.3 Comparisons Using Randomly Sampled Data

Now we compare some model checking results from Section 6.4.2 with approximate results using randomly generated data with 100 samples per client type to validate the effectiveness of our methodology; we give the data statistics in Fig. 6.6 and Tab. 6.5. Whilst we still select random samples for our client probabilities to generate DTMC models, we also randomise the client type frequency and the initial state of the model. We approximated the expected client proportion to have newer data (normal case) and the probability of detection (split-world case) during the first twenty gossip rounds, running $1,000$ trials for each property before evaluating the range of results. The findings are presented in Fig. 6.7. For both cases, our verification results act as effective lower bounds for our simulation results and a helpful indicator of what a negative network traffic scenario looks like, showing how helpful it is over random searching using sample data; we expected this as finding a possible worst-case scenario in an infinitely sized search space would rely more on chance.

Even though there are no limits on the size of the model space to search over, it is unknown how SMBO will perform as the dimensionality of the model space increases or what is the sufficient number of trials needed to be performed to find optimal results. As for our stability testing (see Section 6.4.4), there is room for improvement in our searching so, for future work, we could investigate whether having a prior assumption on which parameters appear to give more robust results (instead of starting with no previous knowledge) produces a better search.

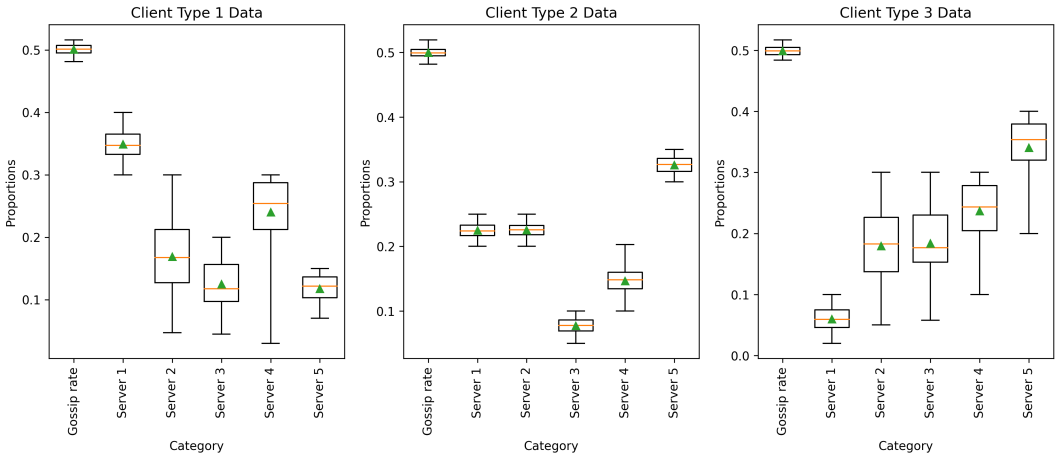


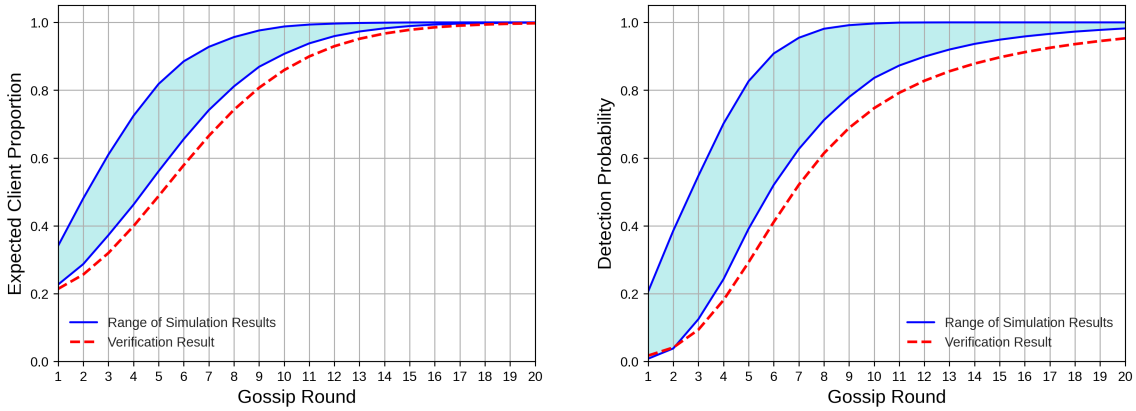
Figure 6.6: Box-and-whisker plots showing the overall range of values for each proportion randomly generated, broken down by client type.

	Category											
	Gossip Rate		Server 1		Server 2		Server 3		Server 4		Server 5	
Client type	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Type C^1	0.481	0.516	0.3	0.4	0.047	0.3	0.045	0.2	0.03	0.3	0.07	0.15
Type C^2	0.482	0.52	0.2	0.25	0.2	0.25	0.05	0.1	0.1	0.203	0.3	0.35
Type C^3	0.484	0.517	0.02	0.1	0.05	0.3	0.058	0.3	0.1	0.3	0.2	0.4

Table 6.5: Minimal/maximal values for each proportion, rounded to three decimal places each.

6.4.4 Investigating the Local Behaviour of the Objective Functions

As the analytic forms of the objective functions used to construct and verify different types of models are unknown, to end this section, we investigate how some of these functions behave locally near our parameters used for model checking (see Table 6.3) to see how erratic it is, determining if the parameters suggested by the optimisation algorithms were sensible choices and hard to improve upon. Using the data from Tab. 6.3, while restricting our investigations to the parameters used for single initial state models and fixing the discrete parameters, we selected probabilities to vary while keeping the other values in the distribution fixed. For each client type, we varied two probabilistic values, performed ten trials and recorded the result achieved



(a) Expected client proportion (normal)

(b) Detection rate (split-world)

Figure 6.7: Comparing the range of simulation results using random sampling (blue shaded area) against the corresponding verification results (red-dashed line) which used the suggested model parameters.

each time, repeating this procedure multiple times by fixing other probabilities. For each client type C^k, we varied $\mathcal{P}(C^k, S^j)$, where $j = 1, \ldots, 4$, and $\mathcal{P}(C^k, S^5)$ within their respective (normalised) intervals, using the value of the former to automatically find the latter.

Fig. 6.8 shows the results and compares them against the worst-case result found in the normal case (see Tab. 6.3). It appears that the results for the expected number of rounds for all clients to obtain the newest STH data remain more or less consistent and tend to lie between 9.5 and 10, though there were rare instances where we found marginally better results. In Fig. 6.9, where we show our findings for the split-world case, the results for the expected number of rounds until client detection appear more stable except in those cases where we randomised probability pairs $\mathcal{P}(C^i, S^3)/\mathcal{P}(C^i, S^5)$, where $i = 1, 2, 3$, with no significant improvements over the best result found. In conclusion, it appears that if we slightly altered some of the probabilities used in our normal scenario model parameters for single initial states, there will be no significant difference in our results. On the other hand, for the split-world model parameters, care must be taken when we slightly alter certain probabilities as it may drastically reduce

the quality of our results.

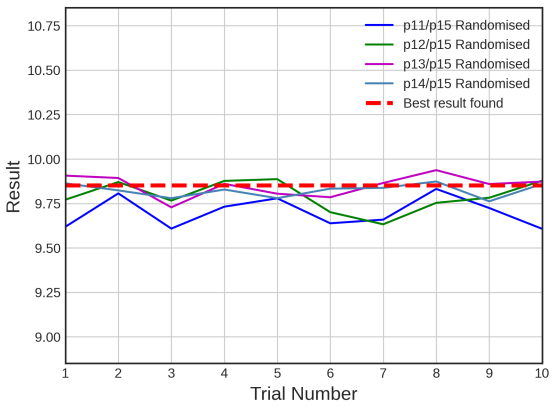
6.5 Combining IDTMCs With SMBO

Examining all the investigations from previous and current chapters, we see IDTMC model checking and SMBO possess both advantages and drawbacks when searching unknown networks and probabilistic parameters. Whilst we still need to fix the discrete parameters, IDTMCs are effective in finding optimal reachability probabilities to target states over a wide range of probabilistic behaviour which our simulation experiments could not achieve. SMBO allows us to search a mixture of parameters and appears to be effective at suggesting worst-case possibilities over a much wider range network modelling scenarios.

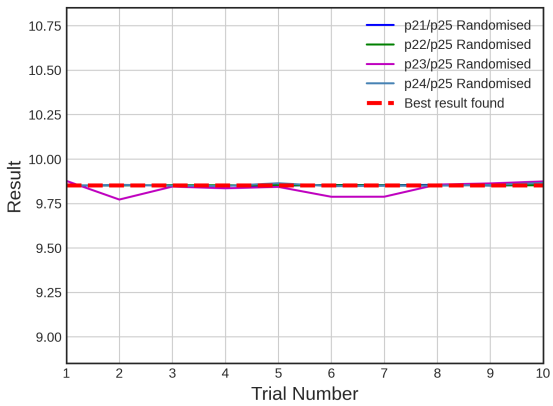
To take it further, we could combine both IDTMC model checking and SMBO by making slight modifications on the objective function and how we search the model space. Firstly, we let $\hat{\chi}$ be a smaller model space which represents all choices and combinations for each *discrete* parameter in the network model. Using a vector $\hat{x} \in \hat{\chi}$ as an argument for the objective function, we construct an IDTMC with the client profile probabilities replaced with fixed intervals from $\mathcal{I}^1, \ldots, \mathcal{I}^N$, and each gossip rate g^i replaced with $[g^i - \varepsilon_1, g^i + \varepsilon_2]$, with $\varepsilon_1, \varepsilon_2 > 0$, performing verification for similar properties. Whilst this improves results, a caveat is the massive computational cost this incurs, so searching in χ and model checking DTMCs (with single initial states) as before is likely to be the cheaper option. Given the prototype status of our IDTMC verification implementation and how it is only compatible with the explicit engine at the time of writing, there are limitations on the number of states/transitions which can be built.

6.6 Summary

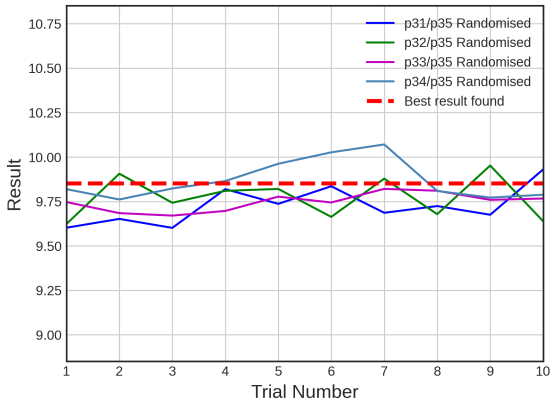
In this chapter, by defining a model space which describes all the set-ups and structures for a network model, we applied SMBO to search over a series of DTMC models to find worst-case scenarios for properties using a Python application that combines the use of PRISM and optimisation libraries. Our investigations show this approach is advantageous over random searching and produces stable results. For future work, we would like to experiment with SMBO and IDTMCs combined to see if this can further improve our results and investigate other optimisation techniques compatible with our problem.



(a) Randomising C^1 type probabilities

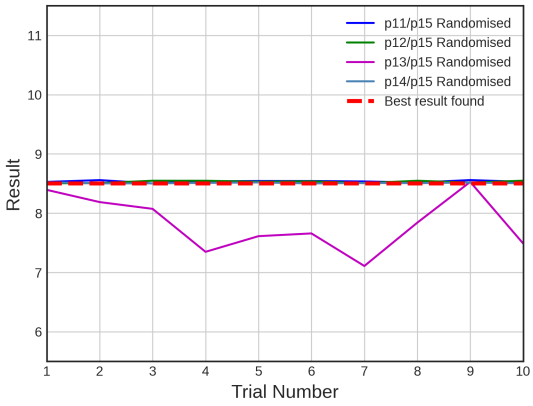


(b) Randomising C^2 type probabilities

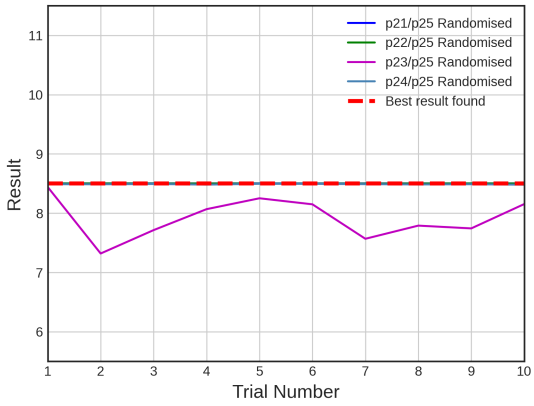


(c) Randomising C^3 type probabilities

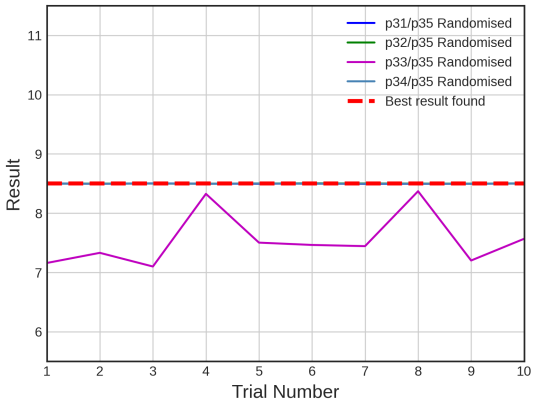
Figure 6.8: The results found when investigating the local behaviour of the objective function that constructs normal scenario models with single initial states. The worst case result found using our implementation (in this case, the expected number of rounds for the newer data to reach all clients) is given by the red dashed line. We randomise over a pair of probabilities whilst fixing the rest of the modelling parameters, seeing how small fluctuations in the continuous parameters impact the output of the objective function. From the graphs shown, it appears that if we slightly altered some of the probabilities used in our normal scenario model parameters for single initial states, there will be no significant difference in our results.



(a) Randomising C^1 type probabilities



(b) Randomising C^2 type probabilities



(c) Randomising C^3 type probabilities

Figure 6.9: The result found when investigating the local behaviour of the objective function that constructs split-world scenario models with single initial states. Due to the erratic behaviour shown when randomising probability pairs $\mathcal{P}(C^i, S^3)/\mathcal{P}(C^i, S^5)$, where $i = 1, 2, 3$, this indicates, care must be taken if we choose slightly alter the client profile probabilities as it will drastically reduce the quality of our results.

Chapter 7

Discussion and Conclusion

We believe that combining gossip with CT will be more prominent in the future to help keep the Internet more accountable, especially with the growing ubiquity of cloud computing [171]. It is therefore important to keep improving on the designs of these gossip protocols. Probabilistic verification is a valuable tool for analysing these protocols.

In this this thesis we show how to model and analyse a network where entities used a gossip protocol to verify for consistency for a CT log, using probabilistic verification to produce a rich amount of experimental data for each of the different protocol designs we studied. We suggest ways to improve the security and efficiency aspects of the protocol's performances by servers simply broadcasting their log digests randomly to other servers, backing up our claims using verification. To account for model uncertainty and unscalability, we show PRISM extension tools which can verify interval Markov chains to find best- and worst-case outcomes for quantitative PCTL properties, performing abstraction on these models to reduce the complexity of the model checking process. Lastly, we apply techniques designed specifically for machine learning problems to search over a set of configurations for our DTMC models to find

bad cases of networks where the gossip protocols may perform poorly, showing empirically that this is a better approach than using a random search.

The findings in this thesis validate some of the conclusions made by Chuat et al. [48], in particular that the *STH-and-proof* version of their protocol is more efficient for clients to use compared to the *STH-only* version as it makes less connections to the log on average when requesting proofs. The thesis also resolves an issue in the methodology used by Chuat et al. where it was unable to find the success rate of detecting a split-world attack, providing exact probabilistic values. Finally, the augmentations to the protocol designs described in this thesis help improve certain aspects over the initial designs, motivating new discussions on how to build upon the work already done with these protocols. We believe that the implications of this research for the wider CT community is that it can inspire the use of model checking for other CT-based protocols or mechanisms to investigate their efficacy in an abstracted setting, experimenting with a number of possible extensions to see if it can improve their performances without having to resort to running expensive or lengthy simulations.

Unlike previous work done in this field, we have been able to analyse relatively large network models, even if we had to resort to finding approximations for properties as we increased the number of clients. However, due to the prototype status of the tools, there is a limit on the size of the IDTMC models (abstracted or otherwise) that can be built due to their reliance on PRISM's explicit engine, suggesting that future work on the code may include how to make it compatible with the more powerful symbolic engines (hybrid or MTBDD).

We had to make some compromises in our model design assumptions and tried not to analyse the protocols at a low level since this would exponentially increase the state space. Realistically speaking, modelling for networks that contain clients with diverse behavioural patterns still remains a huge challenge and a main criticism of our work

is that many devices will have different levels of knowledge of the log and, as part of the protocol logic, may have to audit SCTs if they have not seen them before. When one compares the SCT audit process in the *STH-only* and *STH-and-proof* versions of the Chuat protocols [48], the logic for both of them is identical, so adding a similar feature into our models is unlikely to make our results more meaningful. Furthermore, since we used DTMCs, we had our clients regularly gossip and update data in lock step with each other, producing a coarse representation of an actual network; one way to remedy this is by transforming our models into CTMCs and have client types gossip at different rates, but the trade-off is that model checking becomes even more expensive. Like with IDTMCs, we could alleviate this problem through abstraction.

Lastly, we believe that our idea of combining PRISM and SMBO is a novel approach to finding worst-case scenarios for protocol properties and would like to see this investigated further, in particular how effective SMBO is over random search as the dimension of the modelling space increases. There were initial challenges in understanding and implementing the third-party libraries since we used them in a non-standard way, so we would like to see if other techniques or libraries exist that are appropriate for solving the problem we tried to address.

Bibliography

[1] *506227 - Certificate Transparency: Audit logs by Checking SCTs for Inclusion*. URL: `https://bugs.chromium.org/p/chromium/issues/detail?id=506227#c60` (visited on 17/01/2020).

[2] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison-Wesley Professional, 2003.

[3] E. Akkoyunlu, K. Ekanadham and R. Huber. "Some Constraints and Tradeoffs in the Design of Network Communications". In: *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP'75)*. ACM. Nov. 1975, pp. 67–74.

[4] M. Al-Bassam and S. Meiklejohn. "Contour: A Practical System for Binary Transparency". In: *International Workshop on Cryptocurrencies and Blockchain Technology (CBT'18)*. Vol. 11025. LNCS. Springer, Sept. 2018, pp. 94–110.

[5] B. Amann, M. Vallentin, S. Hall and R. Sommer. *Extracting Certificates from Live Traffic: A Near Real-time SSL Notary Service*. Tech. rep. ICSI, Nov. 2012.

[6] *Apple's Certificate Transparency policy*. URL: `https://support.apple.com/en-gb/HT205280` (visited on 23/05/2021).

[7] R. Ash and C. Doleans-Dade. *Probability and Measure Theory*. Academic Press, 2000.

[8] C. Audet and W. Hare. *Derivative-free and Blackbox Optimization*. Operations Research and Financial Engineering. Springer, 2017.

[9] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33.

[10] A. Aziz, V. Singhal, F. Balarin, R. Brayton and A. Sangiovanni-Vincentelli. "It Usually Works: The Temporal Logic of Stochastic Systems". In: *The 7th International Conference on Computer Aided Verification (CAV'95)*. Vol. 939. LNCS. Springer. 1995, pp. 155–165.

[11] C. Baier. "On Algorithmic Verification Methods for Probabilistic Systems". PhD thesis. habilitation thesis, University of Mannheim, 1998.

[12] C. Baier, B. Engelen and M. Majster-Cederbaum. "Deciding Bisimilarity and Similarity for Probabilistic Processes". In: *Journal of Computer and System Sciences* 60.1 (2000), pp. 187–231.

[13] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT press, 2008.

[14] R. Bakhshi, F. Bonnet, W. Fokkink and B. Haverkort. "Formal Analysis Techniques for Gossiping Protocols". In: *Operating Systems Review* 41.5 (Oct. 2007), pp. 28–36.

[15] R. Bardenet, M. Brendel, B. Kégl and M. Sebag. "Collaborative Hyperparameter Tuning". In: *The 30th International Conference on Machine Learning (ICML'13)*. Vol. 28. June 2013, pp. 199–207.

[16] A. Bart, B. Delahaye, P. Fournier, D. Lime, E. Monfroy and C. Truchet. "Reachability in Parametric Interval Markov Chains Using Constraints". In: *Theoretical Computer Science* 747 (2018), pp. 48–74.

[17] E. Bartocci, L. Bortolussi, T. Brázdil, D. Milios and G. Sanguinetti. "Policy Learning in Continuous-time Markov Decision Processes Using Gaussian Processes". In: *Performance Evaluation* 116 (Nov. 2017), pp. 84–100.

[18] E. Bartocci, L. Bortolussi, L. Nenzi and G. Sanguinetti. "On the Robustness of Temporal Properties for Stochastic Models". In: *The 2nd International Workshop on Hybrid Systems and Biology (HSB'13)*. Vol. 125. Open Access Publishing. Sept. 2013, pp. 3–19.

[19] D. Basin, C. Cremers, T. Kim, A. Perrig, R. Sasse and P. Szalachowski. "ARPKI: Attack Resilient Public-Key Infrastructure". In: *The 21st ACM Conference on Computer and Communications Security (CCS'14)*. ACM, Nov. 2014, pp. 382–393.

[20] J. Beekman, J. Manferdelli and D. Wagner. "Attestation Transparency: Building Secure Internet Services For Legacy Clients". In: *The 11th ACM Asia Conference on Computer and Communications Security (ASIACCS'16)*. May 2016, pp. 687–698.

[21] R. Bellman. "A Markovian Decision Process". In: *Journal of mathematics and mechanics* 4.5 (1957), pp. 679–684.

[22] *Bender - The Hyperparameters Optimiser*. URL: `https://bender.dreem.com/` (visited on 21/01/2020).

[23] M. Benedikt, R. Lenhardt and J. Worrell. "LTL Model Checking of Interval Markov Chains". In: *The 19th International Conference on Tools and*

Algorithms for the Construction and Analysis of Systems (TACAS'13). Vol. 7795. LNCS. Springer. Mar. 2013, pp. 32–46.

[24] J. Bergstra, R. Bardenet, Y. Bengio and B. Kégl. "Algorithms for Hyper-parameter Optimization". In: *The Twenty-fifth Annual Conference on Neural Information Processing Systems (NIPS'11)*. Dec. 2011, pp. 2546–2554.

[25] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins and D. Cox. "Hyperopt: A Python library for Model Selection and Hyperparameter Optimization". In: *Computational Science & Discovery* 8.1 (July 2015).

[26] J. Bergstra, D. Yamins and D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *The 30th International Conference on Mmachine Learning (ICML'13)*. Vol. 28. June 2013.

[27] A. Bianco and L. De Alfaro. "Model Checking of Probabilistic and Nondeterministic Systems". In: *The 15th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*. Vol. 1026. LNCS. Springer. Dec. 1995, pp. 499–513.

[28] J.-F. Bonnans, J. Gilbert, C. Lemaréchal and C. Sagastizábal. *Numerical Optimization: Theoretical and Practical Aspects*. Springer Science & Business Media, 2006.

[29] J. Bonneau. "EthIKS: Using Ethereum to Audit a CONIKS Key Transparency Log". In: *The 20th International Conference on Financial Cryptography and Data Security (FC'16)*. Vol. 9604. LNCS. Springer. Feb. 2016, pp. 95–105.

[30] L. Bortolussi, D. Milios and G. Sanguinetti. "Smoothed Model Checking for Uncertain Continuous-time Markov Chains". In: *Information and Computation* 247 (Apr. 2016), pp. 235–253.

[31] L. Bortolussi, D. Milios and G. Sanguinetti. "U-check: Model Checking and Parameter Synthesis Under Uncertainty". In: *The 12th International Conference on Quantitative Evaluation of Systems (QEST'15)*. Vol. 9259. LNCS. Springer. Sept. 2015, pp. 89–104.

[32] L. Bortolussi and G. Sanguinetti. "Learning and Designing Stochastic Processes from Logical Constraints". In: *The 10th International Conference on Quantitative Evaluation of Systems (QEST'13)*. Vol. 8504. LNCS. Springer. Aug. 2013, pp. 89–105.

[33] L. Bortolussi and S. Silvetti. "Bayesian Statistical Parameter Synthesis for Linear Temporal Properties of Stochastic Models". In: *The 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18)*. Vol. 10806. LNCS. Springer. Apr. 2018, pp. 396–413.

[34] E. Brochu, M. Cora and N. de Freitas. *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*. Tech. rep. University of British Columbia, Nov. 2009.

[35] P. Buchholz. "Exact and Ordinary Lumpability in Finite Markov Chains". In: *Journal of Applied Probability* 31 (1 Mar. 1994), pp. 59–75.

[36] J. Buchmann, E. Karatsiolis and A. Wiesmaier. *Introduction to Public Key Infrastructures*. Springer Science & Business Media, 2013.

[37] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezzé, Y. Rafiq and G. Tamburrelli. "Formal Verification with Confidence Intervals to Establish Quality of Service Properties of Software Systems". In: *IEEE Transactions on Reliability* 65.1 (Aug. 2015), pp. 107–125.

[38] R. Calinescu, K. Johnson and C. Paterson. "FACT: A Probabilistic Model Checker for Formal Verification with Confidence Intervals". In: *The 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*. Vol. 9636. LNCS. Springer. Apr. 2016, pp. 540–546.

[39] S. Cattani and R. Segala. "Decision Algorithms for Probabilistic Bisimulation". In: *The 16th International Conference on Concurrency Theory (CONCUR'02)*. Vol. 2421. LNCS. Springer. Aug. 2002, pp. 371–386.

[40] *Censys - Certificate Search*. URL: `https://censys.io/certificates` (visited on 17/01/2020).

[41] *Cert Spotter - Certificate Transparency Monitor by SSLMate*. URL: `https://sslmate.com/certspotter/` (visited on 17/01/2020).

[42] *Certificate Transparency*. URL: `https://www.certificate-transparency.org/` (visited on 08/07/2017).

[43] *Certificate Transparency Monitoring - Facebook for Developers*. URL: `https://developers.facebook.com/tools/ct/` (visited on 17/01/2020).

[44] S. Chakraborty and J.-P. Katoen. "Model Checking of Open Interval Markov Chains". In: *The 22nd International Conference on Analytical & Stochastic Modelling Techniques & Applications (ASMTA'15)*. Springer. May 2015, pp. 30–42.

[45] K. Chatterjee, K. Sen and T. Henzinger. "Model-checking ω-regular Properties of Interval Markov Chains". In: *The 11th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS'08)*. Vol. 4962. LNCS. Springer. Mar. 2008, pp. 302–317.

[46] T. Chen, T. Han and M. Kwiatkowska. “On the Complexity of Model Checking Interval-valued Discrete Time Markov Chains”. In: *Information Processing Letters* 113.7 (Apr. 2013), pp. 210–216.

[47] V. Chonev. “Reachability in augmented interval Markov chains”. In: *The 11th International Conference on Reachability Problems (RP’19)*. Vol. 11674. LNCS. Springer. Sept. 2019, pp. 79–92.

[48] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie and E. Messeri. “Efficient Gossip Protocols for Verifying the Consistency of Certificate Logs”. In: *The 18th Communications and Networking Simulation Symposium (CNS’15)*. IEEE. Apr. 2015, pp. 415–423.

[49] E. Clarke and E. Emerson. “Design and Synthesis of Synchronisation Skeletons Using Branching Time Temporal Logic”. In: *Workshop on Logics of Programs*. Vol. 131. LNCS. Springer, May 1981.

[50] E. Clarke, E. Emerson and A. Sistla. “Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications”. In: *Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (Apr. 1986), pp. 244–263.

[51] E. Clarke, W. Klieber, M. Nováček and P. Zuliani. “Model Checking and the State Explosion Problem”. In: *LASER Summer School on Software Engineering (LASER’11)*. Vol. 7682. LNCS. Springer. 2011, pp. 1–30.

[52] R. Dahlberg and T. Pulls. “Verifiable Light-weight Monitoring for Certificate Transparency Logs”. In: *The 23rd Nordic Conference on Secure IT Systems (NordSec’18)*. Vol. 11252. LNCS. Springer. Nov. 2018, pp. 171–183.

[53] R. Dahlberg, T. Pulls, J. Vestin, T. Høiland-Jørgensen and A. Kassler. “Aggregation-Based Gossip for Certificate Transparency”. In: *arXiv preprint* (June 2018).

[54] P. D'Argenio, B. Jeannet, H. Jensen and K. Larsen. "Reduction and Refinement Strategies for Probabilistic Analysis". In: *The 2nd Joint International Workshop von Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV'02)*. Vol. 2399. LNCS. Springer. July 2002, pp. 57–76.

[55] L. De Alfaro. "Formal verification of Probabilistic Systems". PhD thesis. Stanford University, 1997.

[56] L. De Alfaro and P. Roy. "Magnifying-lens Abstraction for Markov Decision Processes". In: *The 19th International Conference on Computer Aided Verification (CAV'07)*. Vol. 4590. LNCS. Springer. July 2007, pp. 325–338.

[57] C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J.-P. Katoen and E. Ábrahám. "PROPhESY: A PRObabilistic ParamEter SYnthesis Tool". In: *the 27th International Conference on Computer Aided Verification (CAV'15)*. Vol. 9206. LNCS. Springer. July 2015, pp. 214–231.

[58] C. Dehnert, S. Junges, J.-P. Katoen and M. Volk. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *The 29th International Conference on Computer Aided Verification (CAV'17)*. Springer. July 2017, pp. 592–600.

[59] B. Delahaye. "Consistency for Parametric Interval Markov Chains". In: *The 2nd International Workshop on Synthesis of Complex Parameters (SynCoP'15)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015, pp. 17–32.

[60] B. Delahaye, K. Larsen, A. Legay, M. Pedersen and A. Wasowski. "Consistency and Refinement for Interval Markov Chains". In: *The Journal of Logic and Algebraic Programming* 81.3 (2012), pp. 209–226.

[61] B. Delahaye, D. Lime and L. Petrucci. “Parameter Synthesis for Parametric Interval Markov Chains”. In: *The 17th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’16)*. Springer. Jan. 2016, pp. 372–390.

[62] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart and D. Terry. “Epidemic Algorithms for Replicated Database Maintenance”. In: *The 6th Annual ACM Symposium on Principles of Distributed Computing (PODC’87)*. Dec. 1987, pp. 1–12.

[63] S. Derisavi. “Solution of Large Markov Models Using Lumping Techniques and Symbolic Data Structures”. PhD thesis. University of Illinois, 2005.

[64] A. Donaldson and A. Miller. “Symmetry Reduction for Probabilistic Model Checking Using Generic Representatives”. In: *The 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA’06)*. Vol. 4218. LNCS. Springer. 2006, pp. 9–23.

[65] A. Donaldson, A. Miller and D. Parker. “Language-level Symmetry Reduction for Probabilistic Model Checking”. In: *The 6th International Conference on the Quantitative Evaluation of Systems (QEST’09)*. IEEE. 2009, pp. 289–298.

[66] B. Dowling, F. Günther, U. Herath and D. Stebila. “Secure Logging Schemes and Certificate Transparency”. In: *The 21st European Symposium on Research in Computer Security (ESORICS’16)*. Vol. 9879. LNCS. Springer. Sept. 2016, pp. 140–158.

[67] K. Driscoll, B. Hall, H. Sivencrona and P. Zumsteg. “Byzantine Fault Tolerance, from Theory to Reality”. In: *The International Conference on Computer Safety, Reliability, and Security (SAFECOMP’03)*. Vol. 2788. LNCS. Springer. 2003, pp. 235–248.

[68] P. Duggins. "A Psychologically-Motivated Model of Opinion Change with Applications to American Politics". In: *Journal of Artificial Societies & Social Simulation* 20.1 (Jan. 2017).

[69] L. Dykcik, L. Chuat, P. Szalachowski and A. Perrig. "BlockPKI: An Automated, Resilient, and Transparent Public-Key Infrastructure". In: *The 18th International Conference on Data Mining Workshops (ICDMW'18)*. IEEE. Nov. 2018, pp. 105–114.

[70] J. Eaton, D. Bateman and S. Hauberg. *GNU Octave*. Network theory London, 1997.

[71] P. Eckersley. *Sovereign Keys: A Proposal to Make HTTPS and Email More Secure*. 2011. URL: `https://www.eff.org/deeplinks/2011/11/sovereign-keys-proposal-make-https-and-email-more-secure` (visited on 27/02/2020).

[72] G. Edgecombe. *Certificate Transparency Monitor*. URL: `https://ct.grahamedgecombe.com/` (visited on 17/01/2020).

[73] J. Erman, M. Arlitt and A. Mahanti. "Traffic Classification Using Clustering Algorithms". In: *Proceedings of the 2006 SIGCOMM Workshop on Mining Network Data (MineNet'06)*. ACM, Sept. 2006, pp. 281–286.

[74] S. Eskandarian, E. Messeri, J. Bonneau and D. Boneh. "Certificate Transparency with Privacy". In: *Proceedings on Privacy Enhancing Technologies (PoPETS)* 2017.4 (2017), pp. 329–344.

[75] M. Etemad and A. Küpçü. "Efficient Key Authentication Service for Secure End-to-end Communications". In: *The 9th International Conference on Provable Security (ProvSec'15)*. Vol. 9451. LNCS. Springer. 2015, pp. 183–197.

[76] C. Evans, C. Palmer and R. Sleevi. *Public Key Pinning Extension for HTTP*. Tech. rep. Internet Engineering Task Force, Apr. 2015.

[77] S. Even, O. Goldreich and A. Lempel. "A Randomized Protocol for Signing Contracts". In: *Communications of the ACM* 28.6 (June 1985), pp. 637–647.

[78] *Facebook - Introducing our Certificate Transparency Monitoring tool*. URL: `https://www.facebook.com/notes/protect-the-graph/introducing-our-certificate-transparency-monitoring-tool/1811919779048165` (visited on 17/01/2020).

[79] C. Fang, J. Liu and Z. Lei. "Fine-grained HTTP Web Traffic Analysis Based On Large-scale Mobile Datasets". In: *IEEE Access* 4 (Aug. 2016), pp. 4364–4373.

[80] H. Fecher, M. Leucker and V. Wolf. "Don't Know in Probabilistic Systems". In: *The 13th International SPIN Workshop (SPIN'06)*. Vol. 3925. LNCS. Springer. Apr. 2006, pp. 71–88.

[81] A. Fehnker and P. Gao. "Formal Verification and Simulation for Performance Analysis for Probabilistic Broadcast Protocols". In: *ADHOC-NOW*. Vol. 6. Springer. Aug. 2006, pp. 128–141.

[82] V. Forejt, M. Kwiatkowska, G. Norman and D. Parker. "Automated Verification Techniques for Probabilistic Systems". In: *Formal Methods for Eternal Networked Software Systems (SFM'11)*. Vol. 6659. LNCS. Springer, June 2011, pp. 53–113.

[83] C. Fromknecht, D. Velicanu and S. Yakoubov. "A Decentralized Public Key Infrastructure with Identity Retention". In: *IACR Cryptology ePrint Archive* (Oct. 2014).

[84] P. Gagniuc. *Markov chains: From Theory to Implementation and Experimentation*. John Wiley & Sons, 2017.

[85] O. Gasser, B. Hof, M. Helm, M. Korczynski, R. Holz and G. Carle. "In Log We Trust: Revealing Poor Security Practices with Certificate Transparency Logs and Internet Measurements". In: *The 19th International Conference on Passive and Active Measurement (PAM'18)*. Vol. 10771. LNCS. Springer. 2018, pp. 173–185.

[86] *GitHub - Dreem-Organization/benderopt: Bender Brain*. URL: `https://github.com/Dreem-Organization/benderopt/` (visited on 31/05/2019).

[87] *Github - google/certificate-transparency*. URL: `https://github.com/google/certificate-transparency` (visited on 13/02/2018).

[88] *Github - google/keytransparency: A Transparent and Secure Way to Look up Public Keys*. URL: `https://github.com/google/keytransparency/` (visited on 27/02/2020).

[89] *Github - google/trillian: A Transparent, Highly Scalable and Cryptographically Verifiable Data Store*. URL: `https://github.com/google/trillian` (visited on 28/02/2020).

[90] *Github - hyperopt/hyperopt: Distributed Hyperparameter Optimization*. URL: `https://github.com/hyperopt/hyperopt` (visited on 04/01/2020).

[91] *GNU Octave*. URL: `https://www.gnu.org/software/octave/` (visited on 30/01/2020).

[92] P. Godefroid, M. Huth and R. Jagadeesan. "Abstraction-based Model Checking Using Modal Transition Systems". In: *The 15th International Conference on Concurrency Theory (CONCUR'01)*. Vol. 2154. LNCS. Springer. Aug. 2001, pp. 426–440.

[93] M. Groesser and C. Baier. "Partial Order Reduction for Markov Decision Processes: A Survey". In: *The 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*. Vol. 4111. LNCS. Springer. 2005, pp. 408–427.

[94] *Gurobi - The Fastest Solver*. URL: `https://www.gurobi.com` (visited on 30/01/2020).

[95] J. Gustafsson, G. Overier, M. Arlitt and N. Carlsson. "A First Look at the CT Landscape: Certificate Transparency Logs in Practice". In: *The 18th International Conference on Passive and Active Measurement (PAM'17)*. Vol. 10176. LNCS. Springer, Mar. 2017, pp. 87–99.

[96] S. Haesaert, P. Van den Hof and A. Abate. "Data-driven and Model-based Verification via Bayesian Identification and Reachability Analysis". In: *Automatica* 79 (May 2017), pp. 115–126.

[97] E. Hahn, H. Hermanns and L. Zhang. "Probabilistic Reachability for Parametric Markov Models". In: *International Journal on Software Tools for Technology Transfer* 13.1 (2011), pp. 3–19.

[98] P. Hallam-Baker and R. Stradling. *DNS Certification Authority Authorization (CAA) Resource Record*. Tech. rep. Internet Engineering Task Force, Jan. 2013.

[99] H. Hansson and B. Jonsson. "A Logic for Reasoning About Time and Reliability". In: *Formal aspects of computing* 6.5 (Sept. 1994), pp. 512–535.

[100] E. Heilman, A. Kendler, A. Zohar and S. Goldberg. "Eclipse Attacks on Bitcoin's Peer-to-peer Network". In: *The 24th USENIX Security Symposium (USENIX'15)*. Aug. 2015, pp. 129–144.

[101] T. Henzinger, M. Mateescu and V. Wolf. "Sliding Window Abstraction for Infinite Markov Chains". In: *The 21st International Conference on Computer Aided Verification (CAV'09)*. Vol. 5643. LNCS. Springer. June 2009, pp. 337–352.

[102] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker and M. Siegle. "On the Use of MTBDDs for Performability Analysis and Verification of Stochastic Systems". In: *The Journal of Logic and Algebraic Programming* 56.1-2 (2003), pp. 23–67.

[103] H. Hermanns, B. Wachter and L. Zhang. "Probabilistic CEGAR". In: *The 20th International Conference on Computer Aided Verification (CAV'08)*. Vol. 5123. LNCS. Springer. July 2008, pp. 162–175.

[104] J. Hoffman-Andrews. *Signed Certificate Timestamps Embedded in Certificates*. URL: `https://community.letsencrypt.org/t/signed-certificate-timestamps-embedded-in-certificates/57187` (visited on 17/01/2020).

[105] *How to Enable Certificate Transparency (CT)*. URL: `https://www.digicert.com/certificate-transparency/enabling-ct.html` (visited on 14/01/2020).

[106] R. Howard. *Dynamic Probabilistic Systems, Volume 2: Semi-Markov and Decision Processes*. John Wiley & Sons, 1971.

[107] M. Huth. "On Finite-state Approximants for Probabilistic Computation Tree Logic". In: *Theoretical Computer Science* 346.1 (2005), pp. 113–134.

[108] M. Huth, R. Jagadeesan and D. Schmidt. "Modal Transition Systems: A Foundation for Three-valued Program Analysis". In: *The 10th European Symposium on Programming (ESOP'01)*. Vol. 2028. LNCS. Springer. Apr. 2001, pp. 155–169.

[109] F. Hutter, H. Hoos and K. Leyton-Brown. "Sequential Model-based Optimization for General Algorithm Configuration". In: *The 4th International Conference on Learning and Intelligent Optimization (LION'11)*. Vol. 6683. LNCS. Springer. Jan. 2011, pp. 507–523.

[110] F. Hutter, H.r Hoos and K. Leyton-Brown. "An Evaluation of Sequential Model-based Optimization for Expensive Blackbox Functions". In: *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO'13)*. July 2013, pp. 1209–1216.

[111] F. Hutter, L. Kotthoff and J. Vanschoren. *Automated Machine Learning: Methods, Systems, Challenges*. SSCML. Springer Nature, 2019.

[112] *Internet World Stats - Usage and Population Statistics*. URL: `https://www.internetworldstats.com/` (visited on 06/10/2020).

[113] N. Jansen, F. Corzilius, M. Volk, R. Wimmer, E. Ábrahám, J.-P. Katoen and B. Becker. "Accelerating Parametric Probabilistic Verification". In: *The 11th International Conference on Quantitative Evaluation of Systems (QEST'14)*. Vol. 8657. LNCS. Springer. Sept. 2014, pp. 404–420.

[114] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec and M. Van Steen. "Gossip-based Peer Sampling". In: *Transactions on Computer Systems (TOCS)* 25.3 (Aug. 2007), 8–es.

[115] D. Jones. "A Taxonomy of Global Optimization Methods Based on Response Surfaces". In: *Journal of global optimization* 21.4 (2001), pp. 345–383.

[116] D. Jones, M. Schonlau and W. Welch. “Efficient Global Optimization of Expensive Black-box Functions”. In: *Journal of Global optimization* 13.4 (Dec. 1998), pp. 455–492.

[117] B. Jonsson and K. Larsen. “Specification and Refinement of Probabilistic Processes”. In: *The 6th Symposium on Logic in Computer Science (LICS’91)*. IEEE. July 1991, pp. 266–277.

[118] J.-P. Katoen. “How to Model and Analyze Gossiping Protocols?” In: *Performance Evaluation Review* 36.3 (Nov. 2008), pp. 3–6.

[119] J.-P. Katoen, D. Klink, M. Leucker and V. Wolf. “Three-valued Abstraction For Probabilistic Systems”. In: *The Journal of Logic and Algebraic Programming* 81.4 (May 2012), pp. 356–389.

[120] J.-P. Katoen, D. Klink and M. Neuhäußer. “Compositional Abstraction for Stochastic Systems”. In: *The 7th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS’09)*. Vol. 5813. LNCS. Springer. Sept. 2009, pp. 195–211.

[121] J. Kemeny, J. Snell and A. Knapp. *Denumerable Markov Chains*. Springer, 1976.

[122] W. Khan, M. Kamran, S. Naqvi, F. Khan, A. Alghamdi and E. Alsolami. “Formal Verification of Hardware Components in Critical Systems”. In: *Wireless Communications and Mobile Computing* 2020 (Feb. 2020).

[123] T. Kim, L. Huang, A. Perrig, C. Jackson and V. Gligor. “Accountable Key Infrastructure (AKI): A Proposal for a Public-key Validation Infrastructure”. In: *Proceedings of the 22nd International Conference on World Wide Web (WWW’13)*. ACM, May 2013, pp. 679–690.

[124] D. Klink, A. Remke, B. Haverkort and J.-P. Katoen. "Time-bounded Reachability in Tree-structured QBDs by Abstraction". In: *Performance Evaluation* 68.2 (2011), pp. 105–125.

[125] *Known Logs - Certificate Transparency*. URL: `https://www.certificate-transparency.org/known-logs` (visited on 31/07/2017).

[126] L. Kohnfelder. "Towards a Practical Public-key Cryptosystem". PhD thesis. Massachusetts Institute of Technology, 1978.

[127] I. Kozine and L. Utkin. "Interval-valued Finite Markov Chains". In: *Reliable computing* 8.2 (2002), pp. 97–113.

[128] M. Kwiatkowska, G. Norman and D. Parker. "Analysis of a Gossip Protocol in PRISM". In: *Performance Evaluation Review* 36.3 (Dec. 2008), pp. 17–22.

[129] M. Kwiatkowska, G. Norman and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *The 23rd International Conference on Computer Aided Verification (CAV'11)*. Vol. 6806. LNCS. Springer. July 2011, pp. 585–591.

[130] M. Kwiatkowska, G. Norman and D. Parker. "Probabilistic Model Checking: Advances and Applications". In: *Formal System Verification*. Springer, 2018, pp. 73–121.

[131] M. Kwiatkowska, G. Norman and D. Parker. "Symmetry Reduction for Probabilistic Model Checking". In: *The 19th International Conference on Computer Aided Verification (CAV'06)*. Vol. 4114. LNCS. Springer, July 2006, pp. 234–248.

[132] L. Lamport, R. Shostak and M. Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401.

[133] K Larsen. "Modal Specifications". In: *The 1st International Conference on Computer Aided Verification (CAV'89)*. Vol. 407. LNCS. Springer. June 1989, pp. 232–246.

[134] K. Larsen and A. Skou. "Bisimulation through Probabilistic Testing". In: *Information and Computation* 94.1 (1991), pp. 1–28.

[135] B. Laurie. "Certificate Transparency". In: *Queue* 12.8 (Sept. 2014), p. 10.

[136] B. Laurie. *Github - google/certificate-transparency-rfcs/dns/draft-ct-over-dns*. URL: `https://github.com/google/certificate-transparency-rfcs/blob/master/dns/draft-ct-over-dns.md` (visited on 18/01/2020).

[137] B. Laurie. *Improving SSL Certificate Security*. URL: `https://security.googleblog.com/2011/04/improving-ssl-certificate-security.html` (visited on 24/09/2020).

[138] B. Laurie and E. Kasper. *Revocation Transparency*. Tech. rep. Google, 2012.

[139] B. Laurie, A. Langley and E. Kasper. *Certificate Transparency*. Tech. rep. Internet Engineering Task Force, June 2013.

[140] B. Laurie, A. Langley, E. Kasper, E. Messeri and R. Stradling. *Certificate Transparency Version 2.0*. Internet-Draft. Work in Progress. Internet Engineering Task Force, Feb. 2019.

[141] J. Leyden. *Inside 'Operation Black Tulip': DigiNotar Hack Analysed*. URL: `https://www.theregister.co.uk/2011/09/06/diginotar_audit_damning_fail/` (visited on 14/01/2020).

[142] B. Li, J. Lin, F. Li, Q. Wang, J. Li Qi.and Jing and C. Wang. "Certificate Transparency in the Wild: Exploring the Reliability of Monitors". In: *The 26th ACM Conference on Computer and Communications Security (CCS'19)*. ACM, 2019, pp. 2505–2520.

[143] Y. Liu, J. Sun and Jin S. Dong. "PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers". In: *The 22nd International Symposium on Software Reliability Engineering (ISSRE'11)*. IEEE. Nov. 2011, pp. 190–199.

[144] A. Loewenstern and A. Norberg. *The BitTorrent DHT Protocol*. URL: `http://www.bittorrent.org/beps/bep_0005.html` (visited on 09/10/2018).

[145] J. Löfberg. "Automatic Robust Convex Programming". In: *Optimization methods and software* 27.1 (2012), pp. 115–129.

[146] G. Lowe. "Breaking and Fixing the Needham-Schroeder Public-key Protocol Using FDR". In: *The 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*. Vol. 1055. LNCS. Springer. Mar. 1996, pp. 147–166.

[147] V. Lynch. *What are SSL Precertificates?* 2017. URL: `https://www.thesslstore.com/blog/ssl-precertificates/` (visited on 14/01/2020).

[148] D. Madala, M. Jhanwar and A. Chattopadhyay. "Certificate Transparency Using Blockchain". In: *The 18th IEEE International Conference on Data Mining Workshops (ICDM'18 Workshop)*. IEEE. Nov. 2018, pp. 71–80.

[149] Y. Marcus, E. Heilman and S. Goldberg. "Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network". In: *IACR Cryptology ePrint Archive* (2018).

[150] B. Marinković, P. Glavan, Z. Ognjanović, D. Doder and T. Studer. "Probabilistic Consensus of the Blockchain Protocol". In: *European Conference on Symbolic and Quantitative Approaches with Uncertainty (ECSQARU'19)*. Vol. 11726. LNCS. Springer. Sept. 2019, pp. 469–480.

[151] S. Matsumoto and R. Reischuk. "IKP: Turning a PKI Around with Blockchains". In: *IACR Cryptology ePrint Archive* (Oct. 2016).

[152] P. Maymounkov and D. Mazieres. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric". In: *The 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*. Vol. 2429. LNCS. Springer. Mar. 2002, pp. 53–65.

[153] D. Mazieres and D. Shasha. "Building Secure File Systems out of Byzantine Storage". In: *The 21st Annual Symposium on Principles of Distributed Computing (PODC'02)*. ACM. July 2002, pp. 108–117.

[154] S. Meier, B. Schmidt, C. Cremers and D. Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *The 25th International Conference on Computer Aided Verification (CAV'13)*. Vol. 8044. LNCS. Springer. July 2013, pp. 696–701.

[155] M. Melara, A. Blankstein, J. Bonneau, E. Felten and M. Freedman. "CONIKs: Bringing Key Transparency to End Users". In: *The 24th USENIX Security Symposium (USENIX'15)*. Aug. 2015, pp. 383–398.

[156] R. Merkle. "Secrecy, Authentication, and Public Key Systems". PhD thesis. Stanford University, 1979.

[157] E. Messeri. *Privacy implications of Certificate Transparency's DNS-based protocol*. Tech. rep. Google, Jan. 2017.

[158] A. Micheloni, K.-P. Fuchs, D. Herrmann and H. Federrath. "Laribus: Privacy-preserving Detection of Fake SSL Certificates with a Social P2P Notary Network". In: *The 8th International Conference on Availability, Reliability and Security (ARES'13)*. IEEE. Sept. 2013, pp. 1–10.

[159] C. Mirto, J. Yu, V. Rahli and P. Esteves-Verissimo. "Probabilistic Formal Methods Applied to Blockchain's Consensus Protocol". In: *DSN Workshop on Byzantine Consensus and Resilient Blockchains (BCRB'18)*. June 2018.

[160] D. Monniaux. "Abstract Interpretation of Programs as Markov Decision Processes". In: *Science of Computer Programming* 58.1-2 (2005), pp. 179–205.

[161] A. Moore and D. Zuev. "Internet Traffic Classification Using Bayesian Analysis Techniques". In: *SIGMETRICS Performance Evaluation Review (SIGMETRICS'05)*. ACM, June 2005, pp. 50–60.

[162] S. Nakamoto. *Bitcoin: A Peer-to-peer Electronic Cash System*. 2008. URL: `https://bitcoin.org/bitcoin.pdf` (visited on 27/02/2020).

[163] R. Needham and M. Schroeder. "Using Encryption for Authentication in Large Networks of Computers". In: *Communications of the ACM* 21.12 (Dec. 1978), pp. 993–999.

[164] T. Nguyen and G. Armitage. "A Survey of Techniques for Internet Traffic Classification Using Machine Learning". In: *IEEE communications surveys & tutorials* 10.4 (Oct. 2008), pp. 56–76.

[165] V. Nimal. "Statistical Approaches for Probabilistic Model Checking". MSc Mini-project Dissertation. Oxford University Computing Laboratory, 2010.

[166] L. Nordberg, D. Gillmor and T. Ritter. *Gossiping in CT*. Internet-Draft. Work in Progress. Internet Engineering Task Force, Jan. 2018.

[167] G. Norman and V. Shmatikov. “Analysis of Probabilistic Contract Signing”. In: *Journal of Computer Security* 14.6 (Dec. 2006), pp. 561–589.

[168] D. O’Brien. *Certificate Transparency Enforcement in Google Chrome*. URL: https://groups.google.com/a/chromium.org/g/ct-policy/c/wHILiYf31DE/m/iMFmpMEkAQAJ (visited on 10/12/2020).

[169] D. O’Brien and R. Sleevi. *Github - chronium/ct-policy*. URL: https://github.com/chromium/ct-policy (visited on 17/01/2020).

[170] A. Ouadjaout and A. Miné. “Quantitative Static Analysis of Communication Protocols Using Abstract Markov Chains”. In: *Formal Methods in System Design* 54.1 (2019), pp. 64–109.

[171] M. Oxford, D. Parker and M. Ryan. “Quantitative Verification of Certificate Transparency Gossip Protocols”. In: *The Sixth International Workshop on Security and Privacy in the Cloud (SPC‘20)*. IEEE. July 2020.

[172] C. Palmer. *Intent To Deprecate And Remove: Public Key Pinning*. URL: https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/he9tr7p3rZ8/eNMwKPmUBAAJ (visited on 30/09/2020).

[173] D. Parker, G. Norman and M. Kwiatkowska. “Game-based Abstraction for Markov Decision Processes”. In: *The 3rd International Conference on the Quantitative Evaluation of Systems (QEST’06)*. IEEE. Sept. 2006, pp. 157–166.

[174] *Pickle - Python Object Serialization - Python 3.8.6 Documentation*. URL: https://docs.python.org/3/library/pickle.html (visited on 29/09/2020).

[175] E. Polgreen, V. Wijesuriya, S. Haesaert and A. Abate. “Data-efficient Bayesian verification of parametric Markov chains”. In: *The 13th International Conference on Quantitative Evaluation of Systems (QEST’16)*. Vol. 9826. LNCS. Springer. Aug. 2016, pp. 35–51.

[176] J. Polhemus. *Alum Adventures: Andrew Ayer Keeps Certificate Authorities Honest With Certificate Transparency*. 2017. URL: https://blog.cs.brown.edu/2017/12/19/alum-adventures-andrew-ayer-keeps-certificate-authorities-honest-certificate-transparency/ (visited on 26/02/2020).

[177] *PRISM Manual — Main / Welcome*. URL: https://www.prismmodelchecker.org/manual/ (visited on 21/01/2020).

[178] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

[179] J. Queille and J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In: *The 5th International Symposium on Programming (Programming'82*. Vol. 137. LNCS. Springer. Apr. 1982, pp. 337–351.

[180] A. Rahman, V. Srikumar and A. Smith. "Predicting Electricity Consumption for Commercial and Residential Buildings Using Deep Recurrent Neural Networks". In: *Applied Energy* 212 (Feb. 2018), pp. 372–385.

[181] C. Rasmussen and C. Williams. *Gaussian Processes in Machine Learning*. MIT Press, 2006.

[182] P. Roberts. *Phony SSL Certificate Issued for Google, Yahoo, Skype, others*. URL: https://threatpost.com/phony-ssl-certificates-issued-google-yahoo-skype-others-032311/75061/ (visited on 17/02/2018).

[183] P. Roy, D. Parker, G. Norman and L. de Alfaro. "Symbolic Magnifying Lens Abstraction in Markov Decision Processes". In: *The 5th International Conference on Quantitative Evaluation of Systems (QEST'08)*. IEEE. Sept. 2008, pp. 103–112.

[184] M. Ryan. "Enhanced Certificate Transparency and End-to-End Encrypted Mail". In: *The 21st Network and Distributed System Security Symposium (NDSS'14)*. 2014.

[185] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin and C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. Tech. rep. Internet Engineering Task Force, June 2013.

[186] Q. Scheitle, O. Gasser, T. Nolte, J. Amann, L. Brent, G. Carle, R. Holz, T. Schmidt and M. Wählisch. "The Rise of Certificate Transparency and its Implications on the Internet Ecosystem". In: *The 18th Internet Measurement Conference (IMC'18)*. ACM, Oct. 2018, pp. 343–349.

[187] J. Schlyter and P. Hoffman. *The DNS-based Authentication of Named Entities (DANE) Transport Layer Security (TLS) protocol: TLSA*. Tech. rep. Internet Engineering Task Force, Aug. 2012.

[188] *Sectigo Ltd. - Certificate Search*. URL: `https://crt.sh/` (visited on 17/01/2020).

[189] R. Segala and N. Lynch. "Probabilistic Simulations for Probabilistic Processes". In: *Nordic Journal of Computing* 2.2 (1995), pp. 250–273.

[190] K. Sen, M. Viswanathan and G. Agha. "Model-checking Markov Chains in the Presence of Uncertainties". In: *The 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*. Vol. 3920. LNCS. Springer. Mar. 2006, pp. 394–410.

[191] K. Sen, M. Viswanathan and G. Agha. "On Statistical Model Checking of Stochastic Systems". In: *The 17th International Conference on Computer Aided Verification (CAV'05)*. Vol. 3576. LNCS. Springer, July 2005, pp. 266–280.

[192] A. Singh, N. Tsuen-Wan, P. Druschel and D. Wallach. “Eclipse Attacks on Overlay Networks: Threats and Defenses”. In: *INFOCOM*. IEEE. Apr. 2006.

[193] Damjan Škulj. “Finite Discrete Time Markov Chains with Interval Probabilities”. In: *Soft Methods for Integrated Uncertainty Modelling*. Springer, 2006, pp. 299–306.

[194] R. Sleevi and E. Messeri. *Certificate Transparency in Chrome: Monitoring CT Logs Consistency*. Tech. rep. Google, 2017.

[195] J. Snoek, H. Larochelle and R. Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *The 26th Annual Conference on Neural Information Processing Systems (NIPS’12)*. Dec. 2012, pp. 2951–2959.

[196] J. Sproston. “Qualitative Reachability for Open Interval Markov Chains”. In: *The 10th International Conference on Reachability Problems (RP’18)*. Vol. 11123. LNCS. Springer. Sept. 2018, pp. 146–160.

[197] N. Srinivas, A. Krause and M. Seeger. “Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design”. In: *The 27th International Conference on Machine Learning (ICML’10)*. June 2010.

[198] G. Steel. “Formal Analysis of PIN Block Attacks”. In: *Theoretical Computer Science* 367.1-2 (Nov. 2006), pp. 257–270.

[199] W. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[200] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Vol. 12. Springer Science & Business Media, 2013.

[201] M. Strecker. “Formal Verification of a Java Compiler in Isabelle”. In: *The 18th International Conference on Automated Deduction (CADE-18)*. Vol. 2392. LNCS. Springer. 2002, pp. 63–77.

[202] *Supporting material*. URL: https://github.com/MCOxford/phd_resources (visited on 10/11/2020).

[203] E. Syta, I. Tamas, D. Visher, D. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi and B. Ford. "Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning". In: *The 37th IEEE Symposium on Security and Privacy (S&P'16)*. IEEE. May 2016, pp. 526–545.

[204] P. Syverson, R. Dingledine and N. Mathewson. "Tor: The Second-generation Onion Router". In: *The 13th USENIX Security Symposium (USENIX'04)*. June 2004, pp. 303–319.

[205] P. Szalachowski. "PADVA: A Blockchain-Based TLS Notary Service". In: *The 25th International IEEE Conference on Parallel and Distributed Systems (ICPADS'19)*. IEEE. Dec. 2019, pp. 836–843.

[206] P. Szalachowski. "SmartCert: Redesigning Digital Certificates with Smart Contracts". In: *arXiv preprint* (Mar. 2020).

[207] P. Szalachowski, S. Matsumoto and A. Perrig. "PoliCert: Secure and Flexible TLS Certificate Management". In: *The 21st ACM Conference on Computer and Communications Security (CCS'14)*. ACM, Nov. 2014, pp. 406–417.

[208] *The EEF SSL Observatory*. URL: https://www.eff.org/observatory (visited on 07/08/2017).

[209] O. Toker and H. Ozbay. "On the NP-hardness of Solving Bilinear Matrix Inequalities and Simultaneous Stabilization with Static Output Feedback". In: *Proceedings of 1995 American Control Conference (ACC'95)*. IEEE. June 1995, pp. 2525–2526.

[210] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos and S. Devadas. "Transparency Logs via Append-only Authenticated Dictionaries". In: *The 26th ACM Conference on Computer and Communications Security (CCS'19)*. Nov. 2019, pp. 1299–1316.

[211] A. Tomescu and S. Devadas. "Catena: Efficient Non-equivocation via Bitcoin". In: *The 38th IEEE Symposium on Security and Privacy (S&P'17)*. IEEE. May 2017, pp. 393–409.

[212] B. VanderSloot, J. Amann, M. Bernhard, Z. Durumeric, M. Bailey and J. Halderman. "Towards a Complete View of the Certificate Ecosystem". In: *The 16th Internet Measurement Conference (IMC'16)*. ACM, Nov. 2016, pp. 543–549.

[213] J. Villemonteix, E. Vazquez and E. Walter. "An Informational Approach to the Global Optimization of Expensive-to-evaluate Functions". In: *Journal of Global Optimization* 44.4 (2009), p. 509.

[214] X. Wang and M. El-Said. "DomainPKI: Domain Aware Certificate Management". In: *The 21st Annual Conference on Information Technology Education (SIGITE'20)*. ACM. Oct. 2020, pp. 419–425.

[215] J. Wanner, L. Chuat and A. Perrig. "A Formally Verified Protocol for Log Replication with Byzantine Fault Tolerance". In: *The 39th International Symposium on Reliable Distributed Systems (SRDS'20)*. IEEE. Sept. 2020.

[216] M. Webster, M. Breza, C. Dixon, M. Fisher and J. McCann. "Formal Verification of Synchronisation, Gossip and Environmental Effects for Critical IoT Systems". In: *The 18th International Workshop on Automated Verification of Critical Systems (AVoCS'18)*. July 2018.

[217] J. Wei, D. Duvenaud and A. Aspuru-Guzik. "Neural Networks For the Prediction of Organic Chemistry Reactions". In: *ACS central science* 2.10 (Oct. 2016), pp. 725–732.

[218] C. Williams. *GlobalSign Screw-up Cancels Top Websites' HTTPS Certificates*. URL: http://www.theregister.co.uk/2016/10/13/globalsigned_off/ (visited on 17/12/2016).

[219] J. Wolff. *How a 2011 Hack You've Never Heard of Changed the Internet's Infrastructure*. URL: https://slate.com/technology/2016/12/how-the-2011-hack-of-diginotar-changed-the-internets-infrastructure.html (visited on 14/01/2020).

[220] G. Wood. "Ethereum: A Secure Decentralised Generalised Transaction Ledger". In: *Ethereum Project Yellow Paper* (2014).

[221] G. Xie, M. Iliofotou, T. Karagiannis, M. Faloutsos and Y. Jin. "Resurf: Reconstructing Web-surfing Activity from Network Traffic". In: *IFIP Networking Conference*. IEEE. May 2013, pp. 1–9.

[222] J. Yu, V. Cheval and M. Ryan. "DTKI: A New Formalized PKI with Verifiable Trusted Parties". In: *The Computer Journal* 59.11 (Nov. 2016), pp. 1695–1713.

[223] J. Yu, M. Ryan and C. Cremers. "DECIM: Detecting Endpoint Compromise in Messaging". In: *IEEE Transactions on Information Forensics and Security* 13.1 (Aug. 2017), pp. 106–118.

Appendices

Appendix A

Constructing Components from IDTMCs and Updating ADTMCs

In this section, we explain in detail the stages ***ConstructComponent*** and ***UpdateADTMC*** as described in Chapter 5. To recap, suppose we have an ADTMC $\mathcal{M} = (S, s_\iota, P^l, P^u)$ constructed from the concrete IDTMC model $\mathcal{M}^c = (S^c, s^c_{iota}, \widetilde{P}^l, \widetilde{P}^u)$ after scanning states $s_1, s_2, \ldots, s_{(n-1)} \in S^c$. Assume that $\mathcal{M}^c$ has a unique initial state and has no deadlock states. We wish to update $\mathcal{M}$ with newer information after exploring state $s_n \in S^c$ with the set of neighbouring states $Next(s_n)$ and the lower/upper probability bound function $\delta^l(s_n)/\delta^u(s_n)$.

To map the variables of s_n to variables that make up an abstracted state, we let $Exp_{abs} = \{F_1, F_2, \ldots, F_N\}$ be a collection of functions which maps a state $s \in S^c$ to a value and is one of two types:

$$F^{\text{Int}} : S^c \to \mathbb{Z}, \text{ or}$$

$$F^{\text{Bool}} : S^c \to \{\top, \bot\}.$$

The function ***FindAbstractState*** uses Exp_{abs} to find which abstract state

represents s_n. We express this abstract state as $s_{abs} = (F_1(s_n), F_2(s_n), \ldots, F_N(s_n))$, where each $F_i \in Exp_{abs}$ for every $i = 1, \ldots, n$.

A.1 ConstructComponent

Loop over $Next(s_n)$, the set of neighbouring states of s_n, and for each state $t \in Next(s_n)$, we call ***FindAbstractState*** to find the abstract state t_{abs}. If $t_{abs} \notin Next_{abs}(s_n)$, assign $\delta^l_{abs}(s_n)(t_{abs}) = \widetilde{P}^l(s_n, t)$ and $\delta^u_{abs}(s_n)(t_{abs}) = \widetilde{P}^u(s_n, t)$. Otherwise, denoting the current upper and lower bound as $\widehat{\delta^u}_{abs}(s_n)(t_{abs})$ and $\widehat{\delta^l}_{abs}(s_n)(t_{abs})$ respectively, we let:

$$\delta^l_{abs}(s_n)(t_{abs}) = \widehat{\delta^l}_{abs}(s_n)(t_{abs}) + \widetilde{P}^l(s_n, t),$$
$$\delta^u_{abs}(s_n)(t_{abs}) = \min(1, \widehat{\delta^u}_{abs}(s_n)(t_{abs}) + \widetilde{P}^u(s_n, t)).$$

Finally, if s_n is the unique initial state in the IDTMC, then we set $s_\iota = s_{abs}$.

A.2 UpdateADTMC

We have two distinct cases dependent when whether s_{abs} has already been added to the state space of the ADTMC:

- $s_{abs} \in S$: Find the support of s_{abs} in $\mathcal{M}$, $Sup(s_{abs})$ i.e. the set of neighbouring states of s_{abs} before we update $\mathcal{M}$. For each $t_{abs} \in Next_{abs}(s_n)$, if $t_{abs} \in Sup(s_{abs})$, then let

$$P^l(s_{abs}, t_{abs}) = \min\left(P^l(s_{abs}, t_{abs}), \delta^l_{abs}(s_n)(t_{abs})\right),$$
$$P^u(s_{abs}, t_{abs}) = \max\left(P^u(s_{abs}, t_{abs}), \delta^u_{abs}(s_n)(t_{abs})\right).$$

  Otherwise, add t_{abs} to S if it is a newly found state, letting $P^u(s_{abs}, t_{abs}) =$

$\delta^u_{abs}(s_n)(t_{abs})$ and

$$P^l(s_{abs}, t_{abs}) = \begin{cases} 0, & \text{if } Sup(s_{abs}) \neq \emptyset \\ \delta^l_{abs}(s_n)(t_{abs}), & \text{otherwise} \end{cases}.$$

Lastly, for each $u \in Sup(s_{abs}) \setminus Next_{abs}(s_n)$, we set $P^l(s_{abs}, u) = 0$ as there is a state (i.e. s_n) abstracted by s_{abs} where it cannot possibly transition to any state abstracted by u.

- $s_{abs} \notin S$: Then s_{abs} is added to S. For each $t_{abs} \in Next_{abs}(s_n)$, add t_{abs} to S and have $P^l(s_{abs}, t_{abs}) = \delta^l_{abs}(s_n)(t_{abs})$ and $P^u(s_{abs}, t_{abs}) = \delta^u_{abs}(s_n)(t_{abs})$.

APPENDIX A. CONSTRUCTING COMPONENTS FROM IDTMCS AND UPDATING ADTMCS

Appendix B

Deriving Distributions Using Surrogate Parameters

In this section, we explain in the detail the process of deriving a probability distribution using surrogate parameters. To recap, for a client type C^k and server type S^j, let $I_j^k = [\underline{p}_j^k, \overline{p}_j^k] \subseteq [0,1]$ be the probability interval which specifies the range of values for probability $p_j^k = \mathcal{P}(C^k, S^j)$, and let $\mathcal{I}^k = \left(I_1^k, I_1^k, \ldots, I_M^k\right)$ be a tuple which contains the intervals for each server connection probability for C^k, where the following condition must hold for probability distributions to exist:

$$\sum_{j=1}^{M} \underline{p}_j^k \leq 1 \leq \sum_{j=1}^{M} \overline{p}_j^k.$$

We shall further assume that each $\mathcal{I}^k$ is a *delimited*, meaning that for every $1 \leq j \leq M$ and $p_j^k \in I_j^k$, there exists a probability distribution $\delta = (p_1^k, \ldots, p_j^k, \ldots, p_M^k)$.

We begin the process with the following tuple:

$$\mathcal{I}^k = \mathcal{I}_0^k = \left(I_{(1,0)}^k, I_{(2,0)}^k, \ldots, I_{(M-1,0)}^k, I_{(M,0)}^k\right),$$

where each $I^k_{(j,0)} = \left[\underline{p}^k_{(j,0)}, \overline{p}^k_{(j,0)}\right]$. Let η be the normalisation operator that takes as input a tuple of real-valued intervals $\mathcal{I}$ and outputs a delimited tuple $\widetilde{\mathcal{I}}$.

Step 1: Set $p^k_1 = \underline{p}^k_{(1,0)} + x^k_1 \cdot \left(\overline{p}^k_{(1,0)} - \underline{p}^k_{(1,0)}\right)$ and, in $\mathcal{I}^k_0$, set $I^k_{(1,0)} = p^k_1$ (i.e. $\overline{p}^k_{(1,0)} = \underline{p}^k_{(1,0)} = p^k_1$). Normalise $\mathcal{I}^k_0$ to obtain:

$$\eta\left(\mathcal{I}^k_0\left[I^k_{(1,0)} \to p^k_1\right]\right) = \mathcal{I}^k_1 = \left(p^k_1, I^k_{(2,1)}, \ldots, I^k_{(M,1)}\right) = \left(p^k_1, \ldots, [\underline{p}^k_{(M,1)}, \overline{p}^k_{(M,1)}]\right).$$

Step 2: Set $p^k_2 = \underline{p}^k_{(2,1)} + x^k_2 \cdot \left(\overline{p}^k_{(2,1)} - \underline{p}^k_{(2,1)}\right)$ and, in $\mathcal{I}^k_1$, set $I^k_{(2,1)} = p^k_2$. Normalise $\mathcal{I}^k_1$ to obtain:

$$\eta\left(\mathcal{I}^k_1\left[I^k_{(2,1)} \to p^k_2\right]\right) = \mathcal{I}^k_2 = \left(p^k_1, p^k_2, I^k_{(3,2)}, \ldots, I^k_{(M,2)}\right) = \left(p^k_1, p^k_2, \ldots, [\underline{p}^k_{(M,2)}, \overline{p}^k_{(M,2)}]\right).$$

$\vdots$

Step j: We have the interval tuple

$$\mathcal{I}^k_{j-1} = \left(p^k_1, p^k_2, \ldots, p^k_{j-1}, I^k_{(j,j-1)}, \ldots, I^k_{(M,j-1)}\right),$$

where $I^k_{(\alpha,j-1)} = \left[\underline{p}^k_{(\alpha,j-1)}, \overline{p}^k_{(\alpha,j-1)}\right]$ for every $j \leq \alpha \leq M$. Set p^k_j to equal

$$p^k_j = \underline{p}^k_{(j,j-1)} + x^k_j \cdot \left(\overline{p}^k_{(j,j-1)} - \underline{p}^k_{(j,j-1)}\right).$$

In $\mathcal{I}^k_{j-1}$, set $I^k_{(j,j-1)} = p^k_j$. Next, for each $\beta > j$, normalise $\mathcal{I}^k_{j-1}$ by truncating each

interval $I^k_{(\beta,j-1)}$ to equal $I^k_{(\beta,j)} = \left[\underline{p}^k_{(\beta,j)}, \overline{p}^k_{(\beta,j)}\right]$, where:

$$\underline{p}^k_{(\beta,j)} = \max\left(\underline{p}^k_{(\beta,j-1)}, 1 - \sum_{l=1}^{j} p^k_l - \sum_{\substack{(j+1)\leq\beta'\leq M \\ \beta'\neq\beta}} \overline{p}^k_{(\beta',j-1)}\right),$$

$$\overline{p}^k_{(\beta,j)} = \min\left(\overline{p}^k_{(\beta,j-1)}, 1 - \sum_{l=1}^{j} p^k_l - \sum_{\substack{(j+1)\leq\beta'\leq M \\ \beta'\neq\beta}} \underline{p}^k_{(\beta',j-1)}\right).$$

We get the resulting tuple

$$\eta(\mathcal{I}^k_{j-1}\left[I^k_{(j,j-1)} \to p^k_j\right]) = \mathcal{I}^k_j = \left(p^k_1, p^k_2, \ldots, p^k_{j-1}, p^k_j, I^k_{(j+1,j)}, \ldots, I^k_{(M,j)}\right).$$

$\vdots$

Step *M:* Finally, we get $p^k_M = \underline{p}^k_M = \overline{p}^k_M = 1 - \sum_{j=1}^{M-1} p_j$ and set $I^k_{(M,M-1)} = p^k_M$. We output the probability distribution $\delta = \mathcal{I}^k_M = \left(p^k_1, p^k_2, \ldots, p^k_j, \ldots, p^k_M\right)$.

APPENDIX B. DERIVING DISTRIBUTIONS USING SURROGATE PARAMETERS

Appendix C

Snapshots of PRISM Code

In this section, we provide a couple of sample of the PRISM model used for this thesis.

C.1 Normal Scenario Model (Without Server Gossip)

A PRISM model (DTMC) we used to find verification results for Chapter 4, describing a normal gossip scenario within a network of five clients and servers each:

```
1  //author: mco. Timestamp: 2018-04-12
2
3  // Comments here removed for convenience
4
5  dtmc
6
7  //--------------
8  // CLIENTS
9  //--------------
10
```

```
11 //initial states for each client
12 const bool c1_sth_init = false;
13 const bool c2_sth_init = false;
14 const bool c3_sth_init = false;
15 const bool c4_sth_init = false;
16 const bool c5_sth_init = true;
17
18 //client connect rates
19 prob g1= 0.8;
20 prob g2= 0.6;
21 prob g3= 0.6;
22 prob g4= 0.6;
23 prob g5= 0.2;
24
25 //client profiles
26 prob p_1_1 = 1/50;
27 prob p_1_2 = 7/25;
28 prob p_1_3 = 7/10;
29 prob p_1_4 = 0;
30 prob p_1_5 = 0;
31
32 prob p_2_1 = 1/50;
33 prob p_2_2 = 7/25;
34 prob p_2_3 = 0;
35 prob p_2_4 = 7/10;
36 prob p_2_5 = 0;
37
38 prob p_3_1 = 1/50;
39 prob p_3_2 = 7/25;
40 prob p_3_3 = 0;
41 prob p_3_4 = 7/10;
```

```
prob p_3_5 = 0;

prob p_4_1 = 1/50;
prob p_4_2 = 7/25;
prob p_4_3 = 0;
prob p_4_4 = 7/10;
prob p_4_5 = 0;

prob p_5_1 = 1/50;
prob p_5_2 = 7/25;
prob p_5_3 = 0;
prob p_5_4 = 0;
prob p_5_5 = 7/10;

module Client1

    //global state.
    c1 : [0..5] init 0;
    //connectivity state.
    c1s : [0..5] init 0;
    //current STH data stored by the client. 'True' means the client
   has the latest STH.
    c1_sth : bool init c1_sth_init;
    //skip the round?
    c1_skip : bool init false;

    //client decides randomly to participate in the round.
    [connect] c1=0 -> g1 : (c1'=1) + 1-g1 : (c1'=1) & (c1_skip'=true);

    //client randomly chooses a server if participating in the round.
```

```
71     [choose] c1_skip=false & c1=1 -> p_1_1 : (c1'=2)&(c1s'=1) + p_1_2
       : (c1'=2)&(c1s'=2) + p_1_3 : (c1'=2)&(c1s'=3) + p_1_4 : (c1'=2)&(
       c1s'=4) + p_1_5 : (c1'=2)&(c1s'=5);
72     [choose] c1_skip=true & c1=1 -> (c1'=2);
73
74     //client updates itself using data retrieved from server so long
       as c1_skip=false or c1_sth=true.
75     [update] c1_skip=false & c1=2 & connected_server_has_sth  -> (c1
       '=3) & (c1_sth'=true);
76     [update] c1=2 & ((c1_skip=false & !connected_server_has_sth) |
       c1_skip=true) -> (c1'=3);
77
78     //round complete - reset if there is a client who is not yet
       updated. Otherwise, stop.
79     [round_complete] c1=3 & !clients_all_updated -> (c1'=0) & (c1s'=0)
       & (c1_skip'=false);
80     [round_complete] c1=3 & clients_all_updated -> (c1'=4);
81
82     //self-looping state - go here when every client is updated.
83     [END] c1=4 -> true;
84
85 endmodule
86
87 formula connected_server_has_sth = ((c1s=1 & s1_sth) | (c1s=2 & s2_sth
       ) | (c1s=3 & s3_sth) | (c1s=4 & s4_sth) | (c1s=5 & s5_sth));
88 formula clients_all_updated = c1_sth&c2_sth&c3_sth&c4_sth&c5_sth;
89
90 module Client2=Client1[p_1_1=p_2_1,p_1_2=p_2_2,p_1_3=p_2_3,p_1_4=p_2_4
       ,p_1_5=p_2_5, g1=g2,c1=c2,c1s=c2s, c1_sth=c2_sth, c2_sth=c1_sth,
       c1_skip=c2_skip, c1_sth_init = c2_sth_init] endmodule
91
```

```
92 module Client3=Client1[p_1_1=p_3_1,p_1_2=p_3_2,p_1_3=p_3_3,p_1_4=p_3_4
       ,p_1_5=p_3_5, g1=g3,c1=c3,c1s=c3s, c1_sth=c3_sth, c3_sth=c1_sth,
       c1_skip=c3_skip, c1_sth_init = c3_sth_init] endmodule
93
94 module Client4=Client1[p_1_1=p_4_1,p_1_2=p_4_2,p_1_3=p_4_3,p_1_4=p_4_4
       ,p_1_5=p_4_5, g1=g4,c1=c4,c1s=c4s, c1_sth=c4_sth, c4_sth=c1_sth,
       c1_skip=c4_skip, c1_sth_init = c4_sth_init] endmodule
95
96 module Client5=Client1[p_1_1=p_5_1,p_1_2=p_5_2,p_1_3=p_5_3,p_1_4=p_5_4
       ,p_1_5=p_5_5, g1=g5,c1=c5,c1s=c5s, c1_sth=c5_sth, c5_sth=c1_sth,
       c1_skip=c5_skip, c1_sth_init = c5_sth_init] endmodule
97
98 //--------------
99 // SERVERS
100 //--------------
101
102 //needed for module relabelling
103 const int S1 = 1;
104 const int S2 = 2;
105 const int S3 = 3;
106 const int S4 = 4;
107 const int S5 = 5;
108
109 //initial states for each server
110 const bool s1_init = true;
111 const bool s2_init = false;
112 const bool s3_init = false;
113 const bool s4_init = false;
114 const bool s5_init = false;
115
116 module Server1
```

```
117
118     //does the server have the latest STH?
119     s1_sth : bool init s1_init;
120
121     //update if a connected client has the latest STH
122     [update] !s1_sth & connected_client_has_sth -> (s1_sth'=true);
123     [update] s1_sth | (!s1_sth & !connected_client_has_sth) -> true;
124
125     //end when every client is updated
126     [END] true -> true;
127
128 endmodule
129
130 formula connected_client_has_sth = (c1s=S1 & c1_sth) | (c2s=S1 &
      c2_sth) | (c3s=S1 & c3_sth) | (c4s=S1 & c4_sth) | (c5s=S1 & c5_sth)
      ;
131
132 module Server2=Server1[s1_sth=s2_sth, s2_sth=s1_sth, S1=S2, s1_init=
      s2_init] endmodule
133 module Server3=Server1[s1_sth=s3_sth, s3_sth=s1_sth, S1=S3, s1_init=
      s3_init] endmodule
134 module Server4=Server1[s1_sth=s4_sth, s4_sth=s1_sth, S1=S4, s1_init=
      s4_init] endmodule
135 module Server5=Server1[s1_sth=s5_sth, s5_sth=s1_sth, S1=S5, s1_init=
      s5_init] endmodule
136
137 //--------------
138 // FORMULAS
139 //--------------
140
141 const double f = 1/5;
```

```
142
143 //Keeps track of the proportion of clients that have the latest STH -
    max sum should equal 1.
144 formula no_clients_updated = (c1_sth?f:0)+(c2_sth?f:0)+(c3_sth?f:0)+(
    c4_sth?f:0)+(c5_sth?f:0);
145
146 //For STH-Only, client requests an extension proof if the retrieved
    gossip message has a different tree size to what the client has
147 formula client1_getConsistency_STHOnly = c1_skip=false & c1=2 & ((c1s
    =1 & c1_sth!=s1_sth) | (c1s=2 & c1_sth!=s2_sth) | (c1s=3 & c1_sth!=
    s3_sth) | (c1s=4 & c1_sth!=s4_sth) | (c1s=5 & c1_sth!=s5_sth));
148 formula client2_getConsistency_STHOnly = c2_skip=false & c2=2 & ((c2s
    =1 & c2_sth!=s1_sth) | (c2s=2 & c2_sth!=s2_sth) | (c2s=3 & c2_sth!=
    s3_sth) | (c2s=4 & c2_sth!=s4_sth) | (c2s=5 & c2_sth!=s5_sth));
149 formula client3_getConsistency_STHOnly = c3_skip=false & c3=2 & ((c3s
    =1 & c3_sth!=s1_sth) | (c3s=2 & c3_sth!=s2_sth) | (c3s=3 & c3_sth!=
    s3_sth) | (c3s=4 & c3_sth!=s4_sth) | (c3s=5 & c3_sth!=s5_sth));
150 formula client4_getConsistency_STHOnly = c4_skip=false & c4=2 & ((c4s
    =1 & c4_sth!=s1_sth) | (c4s=2 & c4_sth!=s2_sth) | (c4s=3 & c4_sth!=
    s3_sth) | (c4s=4 & c4_sth!=s4_sth) | (c4s=5 & c4_sth!=s5_sth));
151 formula client5_getConsistency_STHOnly = c5_skip=false & c5=2 & ((c5s
    =1 & c5_sth!=s1_sth) | (c5s=2 & c5_sth!=s2_sth) | (c5s=3 & c5_sth!=
    s3_sth) | (c5s=4 & c5_sth!=s4_sth) | (c5s=5 & c5_sth!=s5_sth));
152
153 //Keeps track of the log connection currently being made using the STH
    -Only protocol
154 formula log_connections_STHOnly = (client1_getConsistency_STHOnly?1:0)
    +(client2_getConsistency_STHOnly?1:0)+(
    client3_getConsistency_STHOnly?1:0)+(client4_getConsistency_STHOnly
    ?1:0)+(client5_getConsistency_STHOnly?1:0);
155
```

```
156 //For STH-and-Proof, client requests an extension proof if it updated
    and the connecting server does not.
157 formula client1_getConsistency_STHAndProof = c1_skip=false & c1=2 & ((
    c1s=1 & c1_sth & !s1_sth) | (c1s=2 & c1_sth & !s2_sth) | (c1s=3 &
    c1_sth & !s3_sth) | (c1s=4 & c1_sth & !s4_sth) | (c1s=5 & c1_sth &
    !s5_sth));
158 formula client2_getConsistency_STHAndProof = c2_skip=false & c2=2 & ((
    c2s=1 & c2_sth & !s1_sth) | (c2s=2 & c2_sth & !s2_sth) | (c2s=3 &
    c2_sth & !s3_sth) | (c2s=4 & c2_sth & !s4_sth) | (c2s=5 & c2_sth &
    !s5_sth));
159 formula client3_getConsistency_STHAndProof = c3_skip=false & c3=2 & ((
    c3s=1 & c3_sth & !s1_sth) | (c3s=2 & c3_sth & !s2_sth) | (c3s=3 &
    c3_sth & !s3_sth) | (c3s=4 & c3_sth & !s4_sth) | (c3s=5 & c3_sth &
    !s5_sth));
160 formula client4_getConsistency_STHAndProof = c4_skip=false & c4=2 & ((
    c4s=1 & c4_sth & !s1_sth) | (c4s=2 & c4_sth & !s2_sth) | (c4s=3 &
    c4_sth & !s3_sth) | (c4s=4 & c4_sth & !s4_sth) | (c4s=5 & c4_sth &
    !s5_sth));
161 formula client5_getConsistency_STHAndProof = c5_skip=false & c5=2 & ((
    c5s=1 & c5_sth & !s1_sth) | (c5s=2 & c5_sth & !s2_sth) | (c5s=3 &
    c5_sth & !s3_sth) | (c5s=4 & c5_sth & !s4_sth) | (c5s=5 & c5_sth &
    !s5_sth));
162
163 //Keeps track of the log connection currently being made using the STH
    -and-Proof protocol
164 formula log_connections_STHAndProof = (
    client1_getConsistency_STHAndProof?1:0)+(
    client2_getConsistency_STHAndProof?1:0)+(
    client3_getConsistency_STHAndProof?1:0)+(
    client4_getConsistency_STHAndProof?1:0)+(
    client5_getConsistency_STHAndProof?1:0);
```

```
165
166 //--------------
167 // REWARD STRUCTURES
168 //--------------
169
170 rewards "rounds"
171     true : 1/4;
172 endrewards
173
174 rewards "client_proportion"
175     true : no_clients_updated;
176 endrewards
177
178 rewards "client_proportion_sq"
179     true : no_clients_updated*no_clients_updated;
180 endrewards
181
182 rewards "log_connections_STHOnly"
183     true: log_connections_STHOnly;
184 endrewards
185
186 rewards "log_connections_STHOnly_sq"
187     true : log_connections_STHOnly*log_connections_STHOnly;
188 endrewards
189
190 rewards "log_connections_STHAndProof"
191     true : log_connections_STHAndProof;
192 endrewards
193
194 rewards "log_connections_STHAndProof_sq"
195     true : log_connections_STHAndProof*log_connections_STHAndProof;
```

```
196 endrewards
```

C.2 Split-world Scenario Model With Intervals (Without Server Gossip)

A PRISM model (IDTMC) we used to find verification results for Chapter 5, describing a split-world gossiping scenario within a network of five clients and server each:

```
1  // Author: mco
2  // Timestamp: 16-09-2020
3
4  dtmc
5
6  //initial states for each client
7  const int c1_sth_init = 0;
8  const int c2_sth_init = 0;
9  const int c3_sth_init = 2;
10
11 //client connect rates
12 prob g1 = 0.5 ;
13 prob g2 = 0.5 ;
14 prob g3 = 0.5 ;
15
16 //client profiles
17 prob p_1_1_L = 0.01 ;
18 prob p_1_2_L = 0.2 ;
19 prob p_1_3_L = 0.2 ;
20 prob p_1_4_L = 0.3;
21 prob p_1_5_L = 1E-14 ;
22 prob p_1_1_H = 0.1 ;
```

```
23 prob p_1_2_H = 0.4 ;
24 prob p_1_3_H = 0.3 ;
25 prob p_1_4_H = 0.4;
26 prob p_1_5_H = 0.29 ;
27
28 prob p_2_1_L = 0.01 ;
29 prob p_2_2_L = 0.2;
30 prob p_2_3_L = 0.4;
31 prob p_2_4_L = 0.2 ;
32 prob p_2_5_L = 1E-14;
33 prob p_2_1_H = 0.1 ;
34 prob p_2_2_H = 0.4;
35 prob p_2_3_H = 0.5;
36 prob p_2_4_H = 0.3 ;
37 prob p_2_5_H = 0.19 ;
38
39 prob p_3_1_L = 0.01;
40 prob p_3_2_L = 0.2;
41 prob p_3_3_L = 0.15;
42 prob p_3_4_L = 0.15;
43 prob p_3_5_L = 1E-14;
44 prob p_3_1_H = 0.1 ;
45 prob p_3_2_H = 0.4;
46 prob p_3_3_H = 0.25 ;
47 prob p_3_4_H = 0.25 ;
48 prob p_3_5_H = 0.49 ;
49
50 module Client1
51
52     //global state
53     c1 : [0..4] init 0;
```

```
54
55    //connectivity state
56    c1s : [0..5] init 0;
57
58    //Record which root hash this node currently has.
59    c1_sth : [0..2] init c1_sth_init;
60
61    //detection state
62    c1d : bool init false;
63
64    //skip the round?
65    c1_skip : bool init false;
66
67    //client decides randomly to participate in the round
68    [connect] c1=0 -> g1 : (c1'=1) + 1-g1 : (c1'=1) & (c1_skip'=true);
69
70    //client randomly chooses a server if participating in the round.
71    [choose] c1_skip=false & c1=1 -> [p_1_1_L, p_1_1_H] : (c1'=2)&(c1s
     '=1) + [p_1_2_L, p_1_2_H] : (c1'=2)&(c1s'=2) + [p_1_3_L, p_1_3_H] :
      (c1'=2)&(c1s'=3) + [p_1_4_L, p_1_4_H] : (c1'=2)&(c1s'=4) + [
     p_1_5_L, p_1_5_H] : (c1'=2)&(c1s'=5);
72    [choose] c1_skip & c1=1 -> (c1'=2);
73
74    //client updates itself using data retrieved from server, checking
      that no issues are present (so long as c1_skip=false).
75    [update] !c1_skip & c1=2 & s_data_ok -> (c1_sth'=c_update) & (c1
     '=3);
76    [update] !c1_skip & c1=2 & !s_data_ok -> (c1d'=true) & (c1'=3);
77    [update] c1_skip & c1=2 -> (c1'=3);
78
```

```
79      //round complete - start the next round if there no client has
        found an issue with the log. Otherwise, go to self-looping state
        and stop.
80      [round_complete] c1=3 & !detect -> (c1'=0) & (c1s'=0) & (c1_skip'=
        false);
81      [round_complete] c1=3 & detect -> (c1'=4) & (c1s'=0) & (c1_skip'=
        false);
82
83      //self-looping state
84      [END] c1=4 -> true;
85
86  endmodule
87
88  //pr_req_successful is true if either a) log returns a valid proof or
        b) client does not need to contact log in the first place
89
90  formula pr_req_successful = (c1s=1 & s1_sth+c1_sth!=3) | (c1s=2 &
        s2_sth+c1_sth!=3) | (c1s=3 & s3_sth+c1_sth!=3) | (c1s=4 & s4_sth+
        c1_sth!=3) | (c1s=5 & s5_sth+c1_sth!=3);
91
92  // Warning messages are used when someone has already found an
        inconsistency and starts to report this to other nodes through
        gossiping.
93  formula warn_msg = (c1s=1 & s1d) | (c1s=2 & s2d) | (c1s=3 & s3d) | (
        c1s=4 & s4d) | (c1s=5 & s5d);
94
95  // Clients update when given brand new data via a server
96  formula c_update = c1_sth + ((c1s=1 & s1_sth>c1_sth)?s1_sth-c1_sth:0)
        + ((c1s=2 & s2_sth>c1_sth)?s2_sth-c1_sth:0) + ((c1s=3 & s3_sth>
        c1_sth)?s3_sth-c1_sth:0) + ((c1s=4 & s4_sth>c1_sth)?s4_sth-c1_sth
        :0) + ((c1s=5 & s5_sth>c1_sth)?s5_sth-c1_sth:0);
```

```
97
98 // An issue will be found in the data if either a) log cannot provide
       a valid proof or b) client gets a warning message.
99 formula s_data_ok = pr_req_successful & !warn_msg;
100
101 formula detect = c1d | c2d | c3d;
102 label "detect" = c1d | c2d | c3d;
103
104 module Client2=Client1[p_1_1_L = p_2_1_L, p_1_2_L = p_2_2_L, p_1_3_L =
        p_2_3_L, p_1_4_L = p_2_4_L, p_1_5_L = p_2_5_L, p_1_1_H = p_2_1_H,
       p_1_2_H = p_2_2_H, p_1_3_H = p_2_3_H, p_1_4_H = p_2_4_H, p_1_5_H =
       p_2_5_H, g1=g2, c1=c2, c1s=c2s, c1_sth=c2_sth, c1_skip=c2_skip, c1d
       =c2d, c2d=c1d, c1_sth_init = c2_sth_init] endmodule
105
106 module Client3=Client1[p_1_1_L = p_3_1_L, p_1_2_L = p_3_2_L, p_1_3_L =
        p_3_3_L, p_1_4_L = p_3_4_L, p_1_5_L = p_3_5_L, p_1_1_H = p_3_1_H,
       p_1_2_H = p_3_2_H, p_1_3_H = p_3_3_H, p_1_4_H = p_3_4_H, p_1_5_H =
       p_3_5_H, g1=g3, c1=c3, c1s=c3s, c1_sth=c3_sth, c1_skip=c3_skip, c1d
       =c3d, c3d=c1d, c1_sth_init = c3_sth_init] endmodule
107
108 //needed for module relabelling
109 const int s1_init = 1;
110 const int s2_init = 0;
111 const int s3_init = 0;
112 const int s4_init = 0;
113 const int s5_init = 0;
114
115 //needed for module relabelling
116 const int S1 = 1;
117 const int S2 = 2;
118 const int S3 = 3;
```

```
119 const int S4 = 4;
120 const int S5 = 5;
121
122 module Server1
123
124     //does the server have the latest STH?
125     s1_sth : [0..2] init s1_init;
126
127     //detection state
128     s1d : bool init false;
129
130     //If any clients sends inconsistent data, cease protocol execution
        and go into detection state.
131     [update] !s1d & c_data_ok -> (s1_sth'=s_update);
132     [update] !s1d & !c_data_ok -> (s1d'=true) & (s1_sth'=0);
133     [update] s1d -> true;
134
135     //self-looping state
136     [END] true -> true;
137
138 endmodule
139
140 //The following formulae is used for the 'update' stage in server 1.
      Server-side, an issue will be found in the data if either a) log
      cannot provide a valid proof or b) a pair of clients connect with
      inconsistent data.
141 formula server_pr_req_fail = (c1s=S1 & s1_sth+c1_sth=3) | (c2s=S1 &
      s1_sth+c2_sth=3) | (c3s=S1 & s1_sth+c3_sth=3);
142
143 formula pairwise_inconsistency = ((c1s=S1 & c1_sth=1) | (c2s=S1 &
      c2_sth=1) | (c3s=S1 & c3_sth=1)) & ((c1s=S1 & c1_sth=2) | (c2s=S1 &
```

```
    c2_sth=2) | (c3s=S1 & c3_sth=2));
144
145 formula c_data_ok = !server_pr_req_fail & !pairwise_inconsistency;
146
147 // Servers update when given brand new data via a connected client.
148 formula s_update = s1_sth + max((c1s=S1&c1_sth>s1_sth?c1_sth-s1_sth:0)
    ,(c2s=S1&c2_sth>s1_sth?c2_sth-s1_sth:0),(c3s=S1&c3_sth>s1_sth?
    c3_sth-s1_sth:0));
149
150 module Server2=Server1[s1_sth=s2_sth, s2_sth=s1_sth, s1d=s2d, s2d=s1d,
    S1=S2, s1_init=s2_init] endmodule
151 module Server3=Server1[s1_sth=s3_sth, s3_sth=s1_sth, s1d=s3d, s3d=s1d,
    S1=S3, s1_init=s3_init] endmodule
152 module Server4=Server1[s1_sth=s4_sth, s4_sth=s1_sth, s1d=s4d, s4d=s1d,
    S1=S4, s1_init=s4_init] endmodule
153 module Server5=Server1[s1_sth=s5_sth, s5_sth=s1_sth, s1d=s5d, s5d=s1d,
    S1=S5, s1_init=s5_init] endmodule
```