Polynomial Timed Reductions to Solve Computer Security Problems in Access Control, Ethereum Smart Contract, Cloud VM Scheduling, and Logic Locking.

by

Jonathan Shahen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Adam Lee Assoc. Dean, School of Computing and Information, Associate Professor, Department of Computer Science, University of Pittsburgh
Supervisor:	Mahesh Tripunitara Professor, Dept. of Computer Engineering, University of Waterloo
Internal Member:	Wojciech Golab Assoc. Professor, Dept. of Computer Engineering, University of Waterloo
Internal Member:	Hiren Patel Assoc. Professor, Dept. of Computer Engineering, University of Waterloo
Internal-External Member:	Florian Kerschbaum Assoc. Professor, Dept. of Computer Science, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The Declaration of Contributions, at the start of each chapter includes a detailed list of all content that is co-authored within that chapter.

I am the sole author of chapters 1, 2, 4, and 6.

Chapter 3 is a modified version of the paper [116], which was authored by: Jonathan Shahen, Jianwei Niu, and Mahesh Tripunitara. The contributions of the chapter are my own, but some paragraphs were written by my coauthors. I am the sole author for Chapter 5, but some of the empirical implementations were performed by Alireza Takami. Chapter 7 is based off the paper [60], written by: Nahid Juma, Jonathan Shahen, Khalid Bijon, and Mahesh Tripunitara. The contributions of the chapter are my own, but some paragraphs were written by my coauthors. Chapter 8 contains research which was performed with the collaboration of Nahid Juma, Mahesh Tripunitara, and later on Mohamed El Massad. Section 8.1.1 is shared work with the above co-authors.

Abstract

This thesis addresses computer security problems in: Access Control, Ethereum Smart Contracts, Cloud VM Scheduling, and Logic Locking. These problems are solved using polynomially timed reductions to 2 complexity classes: **PSPACE**-complete and **NP**-complete. This thesis is divided into 2 parts, problems reduced to: Model Checking (**PSPACE**-complete) and Integer Linear Programming (ILP) (**NP**-complete). The **PSPACE**-complete problems are: Safety Analysis of Administrative Temporal Role Based Access Control (ATRBAC) Policies, and Safety Analysis of Ethereum Smart Contracts. The **NP**-complete problems are: Minimizing Information Leakage in Virtual Machine (VM) Cloud Environments using VM Migrations, and Attacking Logic Locked Circuits using a Reduction to Integer Linear Programming (ILP).

In Chapter 3, I create the Cree Administrative Temporal Role Based Access Control (ATRBAC)-Safety solver. Which is a reduction from ATRBAC-Safety to Model Checking. I create 4 general performance techniques which can be utilized in any ATRBAC-Safety solver.

1. Polynomial Time Solving, which is able to solve specific archetypes of ATRBAC-Safety policies using a polynomial timed algorithm.
2. Static Pruning, which includes 2 methods for reducing the size of the policy without effecting the result of the safety query.
3. Abstraction Refinement, which can increase the speed for reachable safety queries by only solving a subset of the original policy.
4. Bound Estimation, which creates a bound on the number of steps from the initial state, where a satisfying state must exist. This is directly used by the model checker's bounded model checking mode, but can be utilized by any solver with a bound limiting parameter.

In Chapter 4, I analyze ATRBAC-Safety policies to identify some of the "sources of complexity" which make solving ATRBAC-Safety policies difficult. I provide analysis of the sources of complexity that exists in the previously published datasets [133, 93, 56]. I perform analysis of Cree's performance techniques on the previous datasets. I create 2 new datasets, which are shown to be hard instances of ATRBAC-Safety. I analyze the new datasets to show how they achieve this hardness and how they differ from each other and the previous datasets.

In Chapter 5, I create a novel reduction from a Reduced-Solidity Smart Contract, subset of available Solidity features, to Model Checking. This reduction reduces Reduced-Solidity Smart Contract into a Finite State Machine and then reduces to an instance of a Model Checking problem. This provides the ability to test smart contracts published on the

Ethereum blockchain and test if there exists bugs or malicious code. I perform empirical analysis on select Smart contracts.

In Chapter 6, I create 2 methods for generating instances of ATRBAC policies into Solidity Smart Contracts. The first method is the Generic ATRBAC Smart Contract. This method requires no modification before deployment. After deployed the owner is able to create, and maintain, the policy using special access functions. The special action functions are automated with code that converts an ATRBAC policy into a series of transactions the owner can run. The second method is the Baked ATRBAC Smart Contract. This method takes an ATRBAC policy and reduces it to a Smart Contract instance with no special access functions. The smart contract can then be deployed by anyone, and that person will have no special access. I perform an empirical analysis on the setup costs, transaction costs, and security each provides.

In Chapter 7, I create a new reduction from Minimizing Information Leakage via Virtual Machine (VM) Migrations to Integer Linear Programming (ILP). I compare a polynomial algorithm by Moon et al. [73], my ILP reduction, and a reduction to CNF-SAT that is not included in this thesis. The polynomial method is faster, but the problem is **NP**-complete thus that solution must have sacrificed something to obtain the polynomial time speed (unless P = NP). I show instances in which the polynomial time algorithm does not produce the minimum total information leakage, but the ILP and CNF-SAT reductions are able to. In addition to this, I show that Total Information Leakage also has a security vulnerability for non-zero information leakage using the $\langle R, C \rangle$ model. I propose an alternative method to Total Information Leakage, called Max Client-to-Client Information Leakage, which removes the vulnerability at the cost of increased total information leakage.

In Chapter 8, I create a reduction from the Key Recovery Attack on Logic Locked Circuits to Integer Linear Programming (ILP). This is a recreation of the "SAT Attack" using ILP. I provide an empirical analysis of the ILP attack and compare it to the SAT-Attack. I show that "ILP Attack" is a viable attack, thus future claims of "SAT-Attack Resistant Logic Locking Techniques" need to also show resistance to all potential **NP**-complete attacks.

Acknowledgements

I would like to express my gratitude to my supervisor Mahesh Tripunitara. I first studied under you in Spring 2013 (ECE 458). Then we worked together briefly in Winter 2014 (ECE 499). After that you accepted me as your MASc student in Fall 2014, which I graduated in Spring 2016. We then continued on to PhD in Fall 2016 and now I have graduated in Spring 2020. The years working together have been interesting, enjoyable, involved, and very formative. Thank you.

I would like to thank Nahid Juma, who was a PhD student under Mahesh. We worked together on the VM Scheduling and Logic Locking papers. Working together on those projects was very interesting and enjoyable.

Without my friends I would not have had such an enjoyable time working on my masters and then my PhD. I name my friends by year we met: Kesigun Pillai, Johnny Yee, Anthony Silvio, Dan Grimmer, Denis Melanson, Michael Noden, and Kaylin Epp. Thank you Michael for helping me practice for my defence. Jessica Hall, my sister, graduated soon after I started my degree, but she has been rooting for me the entire way. Rick and Sue Storie have been my family since I was young, they helped me move to Waterloo and have been a constant support.

Angela Holmes, we met in Spring 2019 and since then you have become one of the most important people in my life. Thank you for all the love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis addresses computer security problems in: Safety Analysis of Administrative Temporal Role Based Access Control (ATRBAC) Policies, Safety Analysis of Ethereum Smart Contracts, Minimizing Information Leakage in Virtual Machine (VM) Cloud Environments, and Attacking Logic Locked Circuits. This thesis uses polynomial time reductions to Model Checking and Integer Linear Programming, where the decision problems which are **PSPACE**-complete and **NP**-complete.

1.1 Thesis Statement

My thesis is a conjunction of the following:

(A) Safety analysis of ATRBAC can be performed effectively and efficiently in practice via a direct reduction to model-checking, and then use of an off-the-shelf model checker.

(B) There exists "harder" instances of ATRBAC-Safety which take significantly longer to solve, compared to prior published datasets [133, 93, 116] of comparable sizes. The class of ATRBAC-Safety instances which require many rule firings to satisfy the ATRBAC safety query, is one of these "harder" instances.

(C) Safety analysis of Ethereum's Smart Contracts can be performed effectively on smart contracts written in the Solidity language using a reduction to model-checking and then the use of an off-the-shelf model checker.

(D) There exists an efficient reduction to minimize the Total Information Leakage in a cloud environment via VM Migrations using integer linear programming (ILP).

(E) There exists an ILP-attack against logic locking to protect digital Integrated Circuits (ICs) that is as effective as the well-researched and cited SAT-attack [127].

My approach to proving my thesis is by construction. For (A), I create a tool to perform the direct reduction from ATRBAC-Safety to model-checking. The effectiveness of this tool is measured against the prior work tools from Uzun et. al. [133] and Ranise et. al., using the published ATRBAC-Safety datasets from [133, 93, 116]. For (B), I analyze the "sources of complexity", similar to the work in [56], to identify which features of an ATRBAC-Safety policy require more computational resources. From this analysis I will create 2 new datasets, where the sizes of the individual test cases are similar to the test cases from [133, 93, 116]. I will test the new datasets against the best performing tool from the analysis performed in (A), to show that they are indeed "hard" instances of ATRBAC-Safety. For (C), I create a tool to perform a direct reduction from safety analysis of Ethereum Smart Contract to model-checking. This research is ongoing, and my contribution to this project is the initial development of the direct reduction which supports a subset of features from the Solidity language. The effectiveness of this tool will be the ability to solve safety queries for all features of Solidity it supports. For (D), I create a new reduction to ILP and compare the effectiveness of this implementation with the ILP implementation from Moon et. al [73] as well as their greedy algorithm Nomad. The effectiveness of each implementation will be on the ability to reduce the total information leakage and the time required to solve each epoch. For (E), I create a key recovery attack which utilizes an off-the-shelf ILP solver. The effectiveness of the ILP-attack is compared to the SAT-attack using the completion time and number of solved circuits under 10 hrs.

1.2 Motivation

This thesis is motivated by the existence of many custom-made tools which solve computer security problems (lists of these tools is provided in each chapter's "Prior Work" section). We utilize the following motivations to justify the use of reductions instead of programming a custom solution for each problem.

1. Reduces the technical debt that is accumulated when creating an empirical solution from scratch.
2. Leverages the optimizations, and variety of optimization parameters, available from mature solvers in the same complexity class.
3. Provides a baseline of performance that a dedicated solver should achieve.
4. Reduces the time for a viable solver to be created.

5. Can make the problem, and solution, easier to understand when reduced into a well known problem space.

1.3 Contributions and Outline

I provide an outline the contents contained in this thesis and list the contributions made in each chapter. At the beginning of each chapter, a Declaration of Contributions is included which states any shared contributions within the chapters.

- Chapter 2 – All background information is grouped in this chapter.
- Chapter 3 – Introduces the Cree ATRBAC-Safety solver, which is a reduction from ATRBAC-Safety to Model checking. Creates 4 performance techniques which can be used for any ATRBAC-Safety solver.
- Chapter 4 – Analysis of ATRBAC-Safety policies. Identifies some of the "sources of complexity" which make solving ATRBAC-Safety policies difficult. Provides an analysis of the sources of complexity that exists in the previously published datasets. Analysis of the Cree's performance techniques on the previous datasets. Creation of 2 new datasets, which are shown to be hard instances of ATRBAC-Safety. Analysis on the new datasets to show how they achieve this hardness and how they differ from each other.
- Chapter 5 – Safety Analysis on Ethereum/Solidity Smart Contracts. Created a novel reduction from a subset of Solidity Smart Contracts to Model Checking. Show that this subset language is **PSPACE**-complete. Reduces Solidity Smart Contract into a Java Finite State Machine and then converts to an instance of a Model Checking problem.
- Chapter 6 – Created 2 methods for creating instances of ATRBAC policies as Smart Contracts. The Generic ATRBAC Smart Contract requires no modification before deployment. A super user creates and maintains the policy using special access functions once the smart contract is deployed. The creation process is automatic through the use of `web3` commands. The Baked ATRBAC Smart Contract takes an ATRBAC policy and reduces it to a Smart Contract with no super user access. An empirical analysis is provided.
- Chapter 7 – Create an efficient reduction from Minimizing Information Leakage via VM Migrations to Integer Linear Programming (ILP). Compares polynomial algorithm by Moon et al. [73], the new ILP reduction, and a reduction to CNF-SAT that was not

included in this thesis. We show instances in which the polynomial time algorithm does not produce the minimum total information leakage, but the ILP and SAT reductions are able to. In addition to this, we show that Total Information Leakage has a security vulnerability for non-zero information leakage using the $\langle R, C \rangle$ model.

- Chapter 8 – Reduction of a Logic Locked Attack to Integer Linear Programming (ILP). Recreation of the "SAT Attack" [33, 127] using ILP. Create 4 reductions from logic gates to ILP constraints. Provides an empirical analysis of the ILP attack. Able to show that this is a viable attack, thus future claims of "SAT Attack Resistant Logic Locking Techniques" must also show resistance to all potential **NP**-complete attacks.

Chapter 2

Background

Contents

This chapter contains the shared background for all chapters in this thesis. Section 2.1.1 describes Model Checking and is used by all chapters in Part I. Section 2.1.2 describes Integer Linear Programming (ILP) and is used by all chapters in Part II. Besides Section 2.2, each chapter will reference to the particular background sections required for that chapter.

2.1 Problems Reduced To

This section describes the problems that are reduced to in this thesis. Knowledge of Cook/Turing reductions is assumed by the reader. A well posed description of Cook/Turing reduction can be found from the book "Algorithm Design" (chapter 8) by Kleinberg et. al. [64] and from the book "Computational Complex: A Modern Approach" (chapter 2) by Arora et. al. [6].

```
MODULE main
VAR
State : {Locked,Unlocked};
IVAR
Action : {Push,Coin};
ASSIGN
init(State) := Locked;
next(State) := case
  Action = Push & State = Unlocked : Locked;
  Action = Coin & State = Locked : Unlocked;
  TRUE: State;
esac;
```

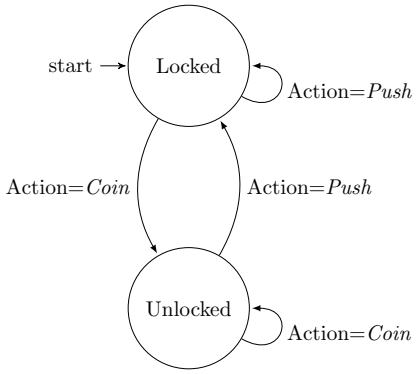


Figure 2.1: Example turnstile finite state machine and corresponding Model Checking problem formatted for NuSMV.

2.1.1 PSPACE-Complete – Model Checking

Model checking is becoming a mature field in computing. Multiple contests are performed annually to compete the newest optimizations in the field of model checking against a variety of problems. The two main contests are: Model Checking Contest [65] and Hardware Model Checkers Competition [17].

Model checking has been developed for the many decades since its inception in [26, 86]. The decision problem for model checking is thus: Given a finite state machine and a temporal logic specification, is the specification true for the given machine? This decision problem has been shown to be **PSPACE**-complete. **PSPACE** is the set of decision problems where a Turing Machine can decide the problem given work tapes whose size is polynomial to the size of the input.

An example model checking problem is shown in Figure 2.1. This models a coin operated turnstile, to allow a single user through and onto the train platform. The turnstile starts in the Locked position, and is unable to be pushed open. When a coin in inserted, the turnstile unlocks and allows someone to push it open. After a push, the turnstile returns to being locked. From this diagram and corresponding NuSMV code, we can perform analysis to determine if certain properties always hold for the model.

Many techniques to optimize the running time and to mitigate the state explosion have been explored. In this thesis 2 methods are utilized: Bounded Model Checking and Symbolic Model Checking. These methods can be found in many commercial and free model checkers.

Symbolic Model Checking, is a technique for solving instances of Model Checking and

Maximize $\sum_{k=1}^{N} value_k * x_k$

Subject To $\sum_{k=1}^{N} weight_k * x_k \leq capacity$

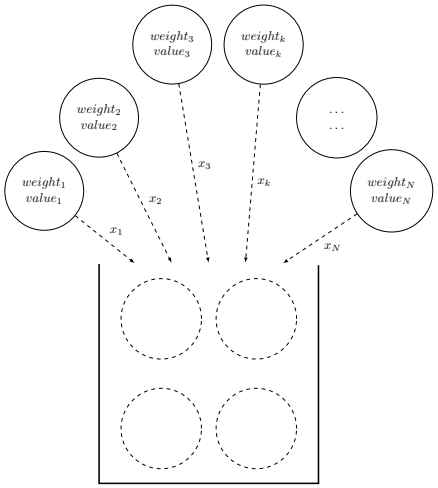


Figure 2.2: Example ILP problem. Given a knapsack with fixed weight capacity, find the maximum value the bag can hold without exceeding the bag's capacity. There are N items, each with a specific weight ($weight_k$) and value ($value_k$). x_k is a Boolean variable to which indicates if the item is in the knapsack.

was first described in [26, 86]. This was not very efficient, but the introduction of the Binary Decision Diagram (BBD) was able to achieve much better performance for large state spaces [21]. It allowed for "10^{30} states to be exhaustively searched in minutes", in 1992. Performance of symbolic model checking has greatly improved from better hardware, to better optimization techniques used in modern solvers.

Bounded Model Checking is an alternative to Symbolic Model Checking which doesn't use BBDs [16]. Bounded model checking allows for converting the model checking problem to a SAT problem. In some cases this can be faster and lower memory than symbolic model checking using BBDs.

See Appendix B for a detailed look at the specification format supported by NuSMV. This thesis uses only the Control Tree Logic (CTL) Specifications in Section B.1.

2.1.2 NP-Complete – Integer Linear Programming (ILP)

Integer Linear Programing (ILP), or Integer Programming, is an optimization problem with an objective function and a set of constraints. If no objective function is given, then it is a feasibility problem. In Chapter 7 we solve an optimization problem, and in Chapter 8 we solve a feasibility problem. Integer linear programming optimization problem is **NP**-hard

and the feasibility problem is **NP**-complete. **NP** is the set of decision problems where there exists a polynomially timed Turing Machine TM, called a verifier, that can decide if x is a solution to the problem given a polynomially sized certificate u. Such that, for all valid solutions x, there exists polynomially sized u such that $TM(x, u) = 1$.

In Figure 2.2, we provide an example ILP problem. This is a variant of the knapsack problem, where we are given a list of N valuable items. The objective is to find the maximum value we can fit in the knapsack does not exceed the weight limitations. Each item has a value and a weight associated with it. We utilize the Boolean variable x_k to indicate if the kth item is in the knapsack.

2.2 Access Control

In this section, we describe Administrative Temporal Role Based Access Control (ATRBAC), and then pose the ATRBAC-Safety problem. We do this in stages. We first introduce Role Based Access Control (RBAC), Administrative Role Based Access Control (ARBAC) and a version of ARBAC-safety that is relevant to ATRBAC-Safety. Then, we describe Temporal Role Based Access Control (TRBAC), ATRBAC, and ATRBAC-Safety.

We then clarify that various versions of ATRBAC- and ARBAC-safety are addressed in the literature, and discuss the choices we have made with regards to the various features of the problem. Specifically, that we have chosen the most general of each feature.

2.2.1 RBAC, ARBAC, and ARBAC-Safety

ATRBAC addresses temporal extensions to RBAC and ARBAC. In this section we discuss RBAC, ARBAC and the version of safety analysis in ARBAC that we call ARBAC-safety that is relevant to our work on ATRBAC-Safety.

RBAC RBAC [37] is used to specify an authorization policy — who has access to what. An RBAC policy, in the context of this work, is a set UA, the *user–role assignment* relation. An instance of UA is a set of pairs of the form $\langle u, r\rangle$. A user u is authorized to the role r if and only if $\langle u, r\rangle \in UA$. RBAC has other constructs, such as role-permission assignment and a role-hierarchy, that are not relevant to ATRBAC-Safety with which we deal in this thesis. Indeed, a role-hierarchy can be flattened as a pre-processing step without affecting the correctness or efficiency of our techniques.

ARBAC ARBAC is a syntax for specifying the ways in which an RBAC policy may

change. As our work deals with the UA component of an RBAC policy only, by ARBAC we mean its URA portion [99], via which users are authorized to and revoked from roles.

There are only two ways in which an instance of UA may change. One is the addition of an entry $\langle u, r\rangle$ to UA, which is the authorization of u to r. The other is the removal of an entry $\langle u, r\rangle$, which is the revocation of u's authorization to r. An instance of ARBAC is a collection of *rules*, and addresses two issues with regards to such changes to UA: who may carry out one of those operations, and under what conditions.

A set of *can_ assign* rules control additions to UA, and a set of *can_ revoke* rules control removals from UA. A *can_ assign* rule is of the form $\langle a, C, t\rangle$, where a, t are roles and C is a precondition. The precondition C is a set in which each entry is either a role r, or its negation, $\neg r$. The semantics of the *can_ assign* rule $\langle a, C, t\rangle$ is that a member of the role a may assign a user u to the role t provided u is already a member of every non-negated role in C and is not a member of any negated role in C.

In a *can_ assign* rule $\langle a, C, t\rangle$, the role a is called an administrative role and the role t is called a target role. A *can_ revoke* rule has the form $\langle a, t\rangle$ where both a and t are roles. The semantics is that a member of the administrative role a is allowed to revoke a user's authorization from the target role t. The reason that a *can_ revoke* rule has no precondition is that revocation is seen as an inherently safe operation [99].

ARBAC-safety We now discuss a version of safety analysis in ARBAC that is relevant to our work. As our work deals with user-role authorization only, the version of ARBAC-safety we talk about here refers to that aspect only. More general versions of safety analysis for ARBAC have been considered in the literature [101], that reconcile not only the user-role authorizations, but also role-role relationships. Nevertheless, all the versions of safety analysis in ARBAC of which we are aware lie in the same complexity-class — they are all **PSPACE**-complete.

ARBAC-safety is a state-reachability problem. It takes three inputs:

(1) A *query*, which is a pair $\langle u, r\rangle$, user u and role r.
(2) A current- or start-state, which is an instance of UA.
(3) A state-change specification, which is an instance of ARBAC, i.e., instances of *can_ assign* and *can_ revoke* rules.

The output of the ARBAC-safety instance is 'FALSE,' if there exists a state that is reachable from the start-state in which the user u from the query is a member of the role r from the query. Otherwise, the output is 'TRUE.'

ARBAC-safety is known to be **PSPACE**-complete [57]. Several techniques have been proposed to address instances that are likely to arise in practice. For example, Gofman et

al. [41] propose a tool called RBAC-PAT, and Jayaraman et al. [56] propose a tool called Mohawk.

2.2.2 TRBAC, ATRBAC, and ATRBAC-Safety

We now discuss the temporal extensions to RBAC and ARBAC that give us TRBAC [12] and ATRBAC respectively. We also the problem ATRBAC-Safety. We first present a model and encoding of time that is the basis for the syntax for temporality in ATRBAC. The version we adopt is the same as prior work [133].

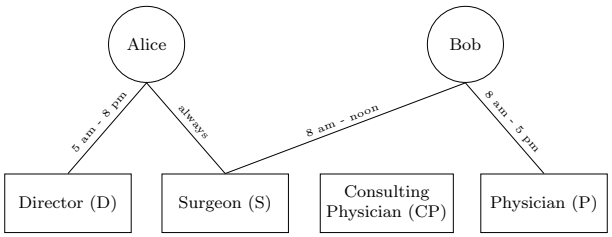
Time An intuition for an instance in time, m, can be thought of as represented by a real number. An example of real number time is the Unix timestamp, which is number of seconds since 00:00:00 Jan. 1, 1970 [79]. A time-slot represents some duration of time, and is represented as a non-negative integer. In an instance of ATRBAC-Safety, no two distinct time-slots overlap in time. Given time-slots i, j where $i < j$, the time-slot j is associated with a duration of time that occurs later than time-slot i. A time-instant m falls within a time-slot.

We assume that the earliest time-slot with which an instance of ATRBAC-Safety is associated is 0, and there is some integer, $T_{\max}$, such that $T_{\max} - 1$ is the latest time-slot that pertains to the ATRBAC-Safety instance. We discuss how time progresses under ATRBAC-Safety below.

A generalization of a time-slot is a time-interval. A time-interval is a pair of integers $\langle i, j \rangle$ where $i \leq j$. It represents the set of time-slots $\{i, i+1, \ldots, j\}$. We say that a time-instant m falls within a time-interval if m falls within one of the time-slots in that time-interval.

The mindset that underlies the above notions for time is that each time-slot represents some realistic, recurring, fixed time period, such as "9 AM – 10 AM." Example timeslots for T_i are: 9 AM to 5 PM (typical work day), Monday at 9 AM to Friday at 5 PM (typical work week), Jan 1 to March 31 (first quarter of the fiscal year). In ATRBAC, the current instance of time dictates what permissions a user has and what rules an administrator can execute. The particular, the actual time periods, to which time-slots in an instance of ATRBAC-Safety map, are irrelevant to the analysis.

TRBAC From the standpoint of our work, TRBAC generalizes RBAC in two, temporal ways. (1) The set UA is generalized to TUA, each of whose elements is a triple $\langle u, r, l_{u,r} \rangle$, where $l_{u,r}$ is a time-interval. The semantics is that u is a member of r during the time-interval $l_{u,r}$ only. (2) Each role r that appears in TUA is annotated with a time-interval,



Rule Type	**Rule**				
CanAssign	⟨TRUE,	8am – 9am,	TRUE,	$[t_{8-12}]$,	CP⟩
CanAssign	⟨D,	8am – 9am,	CP ∧ ¬P,	$[t_{8-12}]$,	P⟩
CanAssign	⟨D,	8am – 9am,	P,	$[t_{8-12}]$,	S⟩
CanRevoke	⟨D,	8am – 9am,	TRUE,	$[t_{8-12}]$,	CP⟩
CanEnable	⟨TRUE,	6am – 8am,	TRUE,	$[t_{8-12}, t_{12-5}]$,	D⟩
CanDisable	⟨D,	always,	TRUE,	$[t_{8-12}, t_{12-5}]$,	D⟩

Figure 2.3: An example of the Start State component (referred to as *TUA* and *RS*) of a TRBAC policy is the figure on top. No roles are enabled. Example ATRBAC administrative rules are in the table. An ATRBAC policy contains: *TUA*, *RS*, and a rule set. Figure 2.4 contains examples of safety queries.

l_r. We say that l_r is the time-interval during which the role r is enabled. The semantics is that outside of the time-interval l_r, no user can exercise a permission that she acquires via the role r. The set of all pairs, $\langle r, l_r \rangle$, is denoted *RS*.

Thus, a TRBAC policy, and therefore a state in the verification problem we consider, is a 3-tuple, $\langle TUA, RS, m \rangle$, where *TUA* and *RS* are as described above, and m is a time-instant. A user u is authorized to a role r at the time-instant, m, if and only if there exists an entry $\langle u, r, l_{u,r} \rangle \in TUA$ such that m is within $l_{u,r}$. The entries in *RS* matter when a user attempts to make an administrative change, i.e., a change to the authorization state. We discuss this under ATRBAC below.

ATRBAC ATRBAC generalizes ARBAC by providing rules for changes to TRBAC policies. As we discuss under "Versions of the problem" below, the version we discuss generalizes prior versions. Under ATRBAC, there are two ways in which a state, which is a TRBAC policy, can change: (1) via an administrative action, or, (2) the passage of time.

Under (1), four kinds of administrative actions are possible to a TRBAC policy, ⟨*TUA*, *RS*, m⟩. It is possible to add an entry to, and remove an entry from *TUA*, and it is possible to add an entry to, and remove an entry from *RS*. The first two kinds of changes are called assign and revoke administrative actions, and the next two are called role enabling and disabling administrative actions. We have the corresponding sets of tuples *t_can_assign*,

t_ can_ revoke, *t_ can_ enable*, and *t_ can_ disable*. (As in Uzun et al. [133], we employ the prefix "*t_*" to distinguish clearly that these are rules for ATRBAC, rather than ARBAC.)

Each such set contains 5-tuples. Each tuple is of the form $\langle C_a, L_a, C_t, L_t, t\rangle$. The first two components, C_a, L_a are conditions on the administrator that seeks to effect the action. The next two components, C_t, L_t, are conditions on the user or role to which the rule pertains. The last component, t, is the target-role; the role that is affected by the action. C_a is either the mnemonic 'true,' or a condition, i.e., a set of negated and non-negated roles. L_a is a set of time-intervals. We specify their semantics below for each kind of administrative rule. The entry t is the target role, i.e., the role that is affected by the firing of the rule. C_t is a role-condition similar to C_a above. There are some important differences between C_a and C_t, however, and we discuss these below for each kind of rule. L_t is a set of time-intervals, similar to L_a above. We discuss the semantics of L_t below for each kind of rule as well.

Such a 5-tuple $\langle C_a, L_a, C_t, L_t, t\rangle$ applies when an administrator, say, Alice, attempts an administrative action at a particular time-instant, m. Each administrative action takes inputs, one of which is the administrator that attempts it, i.e., Alice, and others that we discuss below. In Figure 2.3 we show an example ATRBAC policy and in Figure 2.4 we show it in the context of a safety query. The example in Figure 2.3 is of 2 users that exist in a hospital. There are 4 roles, where only the Director role is able to act under delegation.

Role enabling: the inputs are Alice, a target role t, and a set of time-intervals, L. Alice succeeds in her attempt at enabling the role t if and only if there exists an entry $\langle C_a, L_a, C_t, L_t, t\rangle \in t\_\ can\_\ enable$ for which all of the following are true.

(1) The time-instant, m, at which Alice attempts the action falls within some time-interval in L_a. (2) Alice and the current time-instant m together satisfy the administrative condition, C_a. That is, if p is a non-negated role in C_a, then Alice is a member of p at time-instant m in the current state, $\langle TUA, RS, m\rangle$, and p is enabled at the time-instant m. If n is a negated role in C_a, then either Alice is not a member of n at time-instant m in the current state, or the role n is not enabled, or both. If C_a is the mnemonic 'true,' then the rule may fire provided m is within some time-interval in L_a. (3) The set of time-intervals L is contained within the set of time-intervals L_t. That is, for every time-interval $l \in L$, there exists a time-interval $l_t \in L_t$ such that l is within l_t. (4) The set of time-intervals L satisfies the target condition C_t for every $l \in L$. That is, if p is a non-negated role in C_t, then for every $l \in L$, p is enabled during the time-interval l, in the current-state, i.e., RS. And if n is a negated role in C_t, then for every $l \in L$, n is not enabled during l, in the current-state.

The effect of a successful role enabling by Alice is that the component RS of the current-

state is updated as follows to get a new state: $RS \leftarrow RS \cup \{\langle t, l\rangle : l \in L\}$.

Example: In Figure 2.3 Alice must first enable the role of "Director" so that she can later be allowed to exercise rules where the "Director" role is required by the administrative condition, C_a. Alice may exercise the *t_ can_ enable* rule during 6 am – 8 am as she satisfies C_a during that time. Once she enables it, Alice must wait before she exercises the *t_ can_ revoke* rule, where the "Director" role is required by *t_ can_ revoke*'s C_a, until the current time falls within 8 am – noon.

Role disabling: the inputs are Alice, a target role t, and a set of time-intervals, L. Alice succeeds in disabling t via her action at time-instant m if and only if there exists an entry $\langle C_a, L_a, C_t, L_t, t\rangle \in$ *t_ can_ disable* for which all of the following are true.

(1) The current time-instant, m, falls within some time-interval in L_a. (2) Alice and the current time-instant, m, together satisfy the administrative condition, C_a. (3) The set of time-intervals, L, is contained within the set of time-intervals, L_t. (4) The set of time-intervals L satisfies the target condition C_t for every $l \in L$.

The effect of a successful role disabling by Alice is that the component RS of the current-state is updated as follows to get a new state: $RS \leftarrow RS \setminus \{\langle t, l\rangle : l \in L\}$.

User-role assignment: the inputs are the administrator, Alice, a user u, a target role t to which she seeks to assign u, and a set of time-intervals L. The assignment action that she attempts at time-instant m succeeds if and only if there exists an entry $\langle C_a, L_a, C_t, L_t, t\rangle \in$ *t_ can_ assign* for which all of the following are true.

(1) The current time-instant, m, falls within some time-interval in L_a. (2) Alice and the current time-instant, m, together satisfy the administrative condition, C_a. (3) The set of time-intervals, L, is contained within the set of time-intervals, L_t. (4) The user u and the set of time-intervals L satisfy the target condition C_t for every $l \in L$. That is, if p is a non-negated role in C_t, then u is a member of p during every time-interval $l \in L$. If n is a negated role in C_t, then u is not a member of n in any time-interval $l \in L$. If C_t is the mnemonic 'true,' then there are no constraints on the current role-memberships of the user u.

The effect of a successful assignment by Alice is that the component TUA of the current-state is updated as follows to get a new state: $TUA \leftarrow TUA \cup \{\langle u, t, l\rangle : l \in L\}$.

Example: The example in Figure 2.3 shows that Alice is able to assign the "Consulting Physician" role to Bob during 8 am – noon. She is able to exercise this rule because Bob has the role "Surgeon," and does not have the role "Physician" during 8 am – noon, and

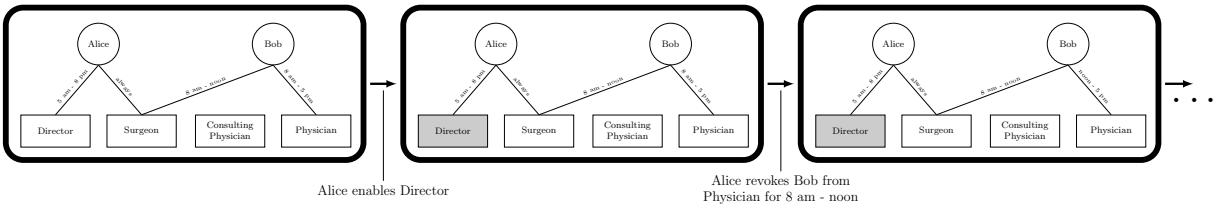


Figure 2.4: An example ATRBAC safety query for the policy in Figure 2.3. We show that the following safety query is true, provided we adopt a version of the problem that does not require the target role to be enabled. Security Query: "Can Bob ever become a member of the role Consulting Physician during 8 am to noon?" This safety query is shown to be true if there exists a path from the initial TRBAC state to a TRBAC state where Bob is a member of the Consulting Physician role for the timeslot representing 8 am to noon. There are many TRBAC states where this safety query is true, and many paths exists to reach these states. Our example path above starts with Alice enabling the Director role (shown shaded). This allows Alice to carry out administrative tasks. Alice then exercises the *t_ can_ revoke* rule so Bob is revoked from the Physician role for 8 am – noon. She then is able to assign him to the Consulting Physician role for 8 am – noon. This last state-change is not shown in the figure. "Can Bob ever become a member of the role Consulting Physician during 1 pm to 5 pm?," is an example of a safety query that is not true because there exists no paths from the initial TRBAC state to a state where the query is true.

Alice satisfies the administrative condition by having the role "Director."

User-role revocation: the inputs are an administrator Alice, a user u that she seeks to revoke from a role, a target role, t from which she seeks to revoke u, and a set of time-intervals, L. The revocation action she attempts at some time-instant m succeeds if and only if there exists an entry $\langle C_a, L_a, C_t, L_t, t\rangle \in t\_\ can\_\ revoke$ for which all of the following are true.

(1) The current time-instant, m, falls within some time-interval in L_a. (2) Alice and the current time-instant, m, together satisfy the administrative condition, C_a. (3) The set of time-intervals, L, is contained within the set of time-intervals, L_t. (4) The user u and the set of time-intervals L satisfy the target condition C_t for every $l \in L$.

The effect of a successful revocation by Alice is that the component TUA of the current-state is updated as follows to get a new state: $TUA \leftarrow TUA \setminus \{\langle u, t, l\rangle : l \in L\}$.

Time-change: Another way that a state, $\langle TUA, RS, m\rangle$, can change is in its time compo-

nent, m. The manner in which passage of time is modelled [93, 133] is simply by allowing the m component to increase without any change to the other two components, TUA and RS. That is, a possible state-change is from $\langle TUA, RS, m\rangle$ to a new state, $\langle TUA, RS, m'\rangle$, where $m' > m$.

An issue we clarify in this regard of passage of time is whether, once we reach the time-slot $T_{\max} - 1$ to which an instance of ATRBAC-Safety pertains, the time-slot 0 recurs, followed by time-slot 1 and so on, forever. The assumption in prior work [133] is that it does. The reason regards the semantics of a time-slot — it maps to some realistic, recurring period of time. We refer to this property as periodicity, and revisit it in the context of the software tools.

Example: Time periodicity is what allows the rules in Figure 2.3 to be described by just the time of day. The intention of the rules is that they are contained within a day. Thus when a day ends and the next day begins, the rules should still apply to the new day.

ATRBAC-Safety The safety analysis problem for ATRBAC takes three inputs. (1) A query, $\langle u, C, L, t\rangle$, user u, condition C (set of negated and non-negated roles), set of time-intervals L, and t is some units of time. (2) A start-state, $\langle TUA, RS, m\rangle$, which is an instance of TRBAC. (3) A state-change specification, which is an instance of ATRBAC, i.e., four sets of rules, $t\_\ can\_\ assign$, $t\_\ can\_\ revoke$, $t\_\ can\_\ enable$, and $t\_\ can\_\ disable$.

The output is 'FALSE,' if there exists a TRBAC state $\langle TUA', RS', m'\rangle$ that is reachable from the start-state in which: (i) the user u is a member of every non-negated role in C in every time-interval in L, and is not a member of any negated role in C in any time-interval in L, (ii) every non-negated role in C is enabled for every time-interval in L, and no negated role in C is enabled in any time-interval in L, and, (iii) the time-instant m' of this state is within t time-units of the time-instant of the start-state. Otherwise, the output is 'TRUE.' We point out that it is possible to specify t that is large enough that the query pertains to any time-slot.

In Figure 2.4 we discuss two ATRBAC safety questions: "could Bob become a member of the role Consulting Physician between 8 am and noon?" and "could Bob become a member of the role Consulting Physician between 1 pm and 5 pm?". As the caption of the figure discusses, the former is true, provided we do not require the role Consulting Physician to be enabled when Bob becomes a member of it. The latter question is not true.

Versions of the problem Different versions of ATRBAC-Safety appear in relevant prior work. In [115], we created a general version from prior works of Uzun et al. [133], and

Ranise et al. [93]. We use this general version of the problem as input into our empirical implementation, Cree.

We previously provided a Reduction Toolkit [115], which allows for this general version to be reduced to previous version of ATRBAC-Safety and to a version of ARBAC-Safety. Here we outline the input format of the general version of ATRBAC-Safety we support.

(1) **Rule Types:** $t\_\ can\_\ assign$, $t\_\ can\_\ revoke$, $t\_\ can\_\ enable$, and $t\_\ can\_\ disable$

(2) **Rule Format:** $\langle R_a, Ti_a, C_t, Ts_t, R_t \rangle$; where:

- Administrator Condition R_a – single role or TRUE
- Admin Time Condition Ti_a – a time interval in the format of $t_a - t_b$ where $a, b \in \mathbb{Z}$ and $0 \leq a \leq b$
- Precondition C_t – a list of positive and negative roles (eg: $r_1 \wedge \neg r_2 \wedge r_4$)
- Target Timeslot Array Ts_t – a list of time slots (eg: t_2, t_3, t_5, t_6)
- Target Role R_t – single role

(3) **Initial Condition:** empty TUA and empty RS

(4) **Security Query:** $\langle t_g, R_g \rangle$

- Goal Timeslot t_g – single timeslot
- Goal Roles R_g – a list of goal roles

The example, revisited We now revisit the example from Figure 2.3 and Figure 2.4 to illustrate the non-triviality of ATRBAC-Safety analysis for even such a small example. The system has 6 rules only, and each rule is essential. For example, if the first $t\_\ can\_\ assign$ rule is removed, then no user can be assigned to the CP role except by a super-user who operates outside the constraints that the ATRBAC rules impose, which is exactly the scalability issue that such delegation schemes as ATRBAC address. Thus, even though the removal of any rule, in this example, is sufficient to ensure that "could Bob become a member of the role Consulting Physician between 8 am and noon?" is not true, such removal is not a viable way to address the unsafety of the system. One way to ensure that the system is safe for both the queries we discuss above is to change the "8 am – noon" component of the $t\_\ can\_\ revoke$ rule to "8 am – 5 pm." Another way is to change the "8 am – noon" constraint in the $t\_\ can\_\ assign$ rule to "8 am – 5 pm," and also change the current state so all surgeons, including Bob, are authorized to the Surgeon role during 8 am – 5 pm, rather than 8 am – noon only. We suggest that identifying that such a change simultaneously renders the system safe for both queries, and also from any other queries

of interest, is not necessarily straightforward. We point out also that the above discussion suggests that it is not straightforward to label a system as "under-" or "over-constrained." Changing the "8 am – noon" constraint in the *t_ can_ assign* rule to "8 am – 5 pm" can be seen as easing a constraint. But the consequence of then precluding Bob from certain privileges can be seen as further constraining the system.

2.3 Ethereum Blockchain

At the core of the Ethereum Network is a cryptocurrency, similar to Bitcoin, where a distributed network of computers (called "miners"), compete to add the next block to an immutable and public ledger of transactions (called a "blockchain"). When a miner finishes "mining" the next block, the miner broadcasts it to all miners in the network, they can verify if the block is correct and if so they award that miner money for its efforts and the block is added to all the miner's ledgers. The methods used for mining, and proof-of-work algorithm, are beyond the scope of this thesis.

2.3.1 DApps and Smart Contracts

The Ethereum Network adds the ability for Smart Contracts and Distributed Applications (DApps) to run within the distributed network of miners and interact and add to the blockchain. This allows for the creation of new coins within the Ethereum network, without the need to spend money on setting up the infrastructure required to run the cryptocurrency.

A Distributed Application (DApp) is a set of Smart Contracts which can run on the distributed network within the Ethereum Network. Smart contracts are immutable code that is placed on in the blockchain ledger. They have an API style interface, where people and other contracts can call functions within them. All data and code in Ethereum is public, thus external services are required to keep secret data off the public ledger.

Properly programmed Smart Contracts are able to act as trusted third parties in financial dealings. They can be trusted because their code is public and immutable and they run on a distributed network where 51% of the miners would need to be compromised in order to pass a fraudulent block. We have provided some example smart contracts in Appendix H. Below we list some of the use cases for smart contracts:

- **Escrow** – The ability to hold assets from 2 or more parties and then exchange them once all parties have provided the required assets for exchange.

Table 2.1: Shortlist of the denominations of Ether.

Unit Name	Wei Value	Description
Wei	1 wei	Smallest denomination
Gwei	1×10^9 wei	Used for converting gas to wei
Ether (eth)	1×10^{18} wei	Biggest/Standard denomination

- **Games** – Some of the most popular games revolve around trading cards/assets. Other games connect their in-game store to the Ethereum network to accept payments and handle asset delivery.
- **Initial Coin Offering (ICO)** – A way to raise funds for a new business where coin shares can be given special meaning.
- **Buy/Sell and Auctions** – The ability to replace e-commerce websites by transferring ether for access tokens or receipts of sale.
- **Tokens/Gift Cards** – The ability to exchange real money/ether for unique tokens that have arbitrary meaning and worth. These tokens are restricted to only be used in particular stores (gift card replacement).
- **Access Control** – Creating an Ethereum account is free, easy, and anonymous. Each account has a strong identity, meaning it is hard to impersonate an account. In order to impersonate an account/address in any Ethereum transaction, the transaction must be correctly signed with a 256-bit Elliptic Curve Digital Signature Algorithm (ECDSA). A secret 256-bit private key is stronger than most secret password based identities (based on key size). Private keys can be AES encrypted with a password, thus increasing the strength of identity. This can allow for more secure access control systems when relying on the strength of identity.

2.3.2 Currencies

Table 2.1 provides a list of the denominations used within this thesis. The standard currency that is traded is called Ether, this is similar to dollars in USD. Ether can be split similar to normal currencies. We deal with different denominations of Ether when dealing with transactions and when dealing with Gas prices (discussed in detail in Section 2.3.5). Under the hood all currencies are stored in Wei (similar to cents in USD), gas is calculated

using Gwei, and amounts to send to people/contracts are usually denoted in Ether. Using these currencies in different scenarios allows for smaller looking numbers; i.e. 1 ether vs 1×10^{18} wei.

2.3.3 Tools for Interacting with the Ethereum Network

There are many tools for interacting with the Ethereum Network. We list a few below:

- MetaMask (https://metamask.io) is a browser extension that can facilitate all interactions except for mining.
- TrustWallet (https://trustwallet.com) is a application that runs on smartphones and can facilitate all interactions except for mining.
- Go-Ethereum (https://geth.ethereum.org) is a computer application that can facilitate all interactions except for GPU mining. Go Ethereum is built by Ethereum and allows for simple to complex interactions with the Ethereum blockchain.
- Web3 (https://web3js.readthedocs.io) is an Ethereum Javascript API which acan facilitate all interactions except mining.
- EthMiner (https://github.com/ethereum-mining/ethminer) is a OpenCL/CUDA Ethereum GPU miner that is fast enough, given adequate hardware, to mine blocks on the main network.
- Remix (https://remix.ethereum.org) is an online Smart Contract creation tool. It allows for compiling, debugging, and deploying smart contracts to an Ethereum blockchain running in your browser. Solidity and Vyper are supported.

Creating an Account

With Go Ethereum installed, the executable `geth` will be available, execute the following command to create new accounts (as many as required).

```
$ geth account new
Your new account is locked with a password. Please give a password. Do not forget this password.
Passphrase:
Repeat Passphrase:
Address: {168bc315a2ee09042d83d7c5811b533620531f67}
```

The program has created an account with the public address `168b...`, and has saved your private key somewhere on your system. The password you provided is used to encrypt the private key and not your new Ethereum account. The only way to interact with the Ethereum network is by using an account and signing each message with the private key.

Receiving/Buying/Mining Ether

Receiving Ether is a passive action on the part of the account holder. A contract or account can send Ether and it will always be accepted.

Buying Ether requires interacting with an exchange, similar and often the same places as currency exchanges. Anyone can visit an exchange (online or in person) and exchange currency for Ether to be sent to an account of their choice.

Mining Ether is how new Ether is created. Mining is the process of adding a new block to the block chain. Blocks contain a certain number of transactions, and the process of mining is a computationally difficult task. The reward to the miner who successfully mines the next accepted block is 5 Ether, which is automatically transfered to the miner's account.

Sending Ether

Using Go Ethereum and the included Javascript console (utilizes web3), we can use this for sending transactions and interacting with the Ethereum Network in a more robust way (compared to command line options). To send Ether, use the following steps to open the console and then send money. You must have access to the `from` account via `geth` in order to properly sign the transaction. Once the transaction is sent, you must wait for the next block to be mined for your transaction to be added to the blockchain.

```
$ geth console
> eth.sendTransaction({from:168bc315a2ee09042d83d7c5811b533620531f67,
        to:eth.accounts[1], value: web3.toWei(0.05, "ether")})
Please unlock account 168bc315a2ee09042d83d7c5811b533620531f67.
Passphrase:
Account is now unlocked for this session.
'0xeeb66b211e7d9be55232ed70c2ebb1bcc5d5fd9ed01d876fac5cff45b5bf8bf4'
```

Interacting with a Smart Contract

In order to interact with a Smart Contract you need to know the interface it is using, this can be looked up on EtherScan or other websites. All Smart Contracts on the Ethereum

```
pragma solidity ^0.6.0;

contract HelloWorld {
  int count = 0; // Smart Contract state variable
  // Counts the number of calls, and returns the string "Hello World"
  function helloWorld() external returns (string memory) {
    count = count + 1;
    return "Hello, World!";
  }
}
```

Figure 2.5: Simple example of a Solidity Smart Contract.

Network have an Application Binary Interface (ABI) attached to them. This describes all the functions available in the smart contract. Using the ABI, and the address of the instance of the contract, the following code will call the `greet` function using Go-Ethereum's Javascript console. Once the transaction is sent, you must wait for the next block to be mined for your transaction to be added to the blockchain.

```
$ geth console
> var greeter =  eth.contract([{constant:false,inputs:[],name:'kill',outputs:[],type:'function'},
    {constant:true,inputs:[],name:'greet',outputs:[{name:'',type:'string'}],type:'function'},
    {inputs:[{name:'_greeting',type:'string'}],type:'constructor'}]
    ).at('0xdaa24d02bad7e9d6a80106db164bad9399a0423e');
> var greet_name = greeter.greet();
```

2.3.4 Solidity and Vyper

Solidity (Java like) [34] and Vyper (Python like) [138] are two popular higher-level languages that smart contracts can be written in. Both languages compile their smart contracts to EVM bytecode, which is the assembly language for the Ethereum Virtual Machine (EVM). Solidity has been recognized as the official language by Ethereum, but any language which compiles correct EVM byte-code can be used to create a Smart Contract. The website Etherscan (https://etherscan.io/), which tracks and reports on many aspects of the main Ethereum network, has published verified solidity and vyper smart contract code for some of the contracts on the Ethereum network.

Figure 2.5 shows a simple Hello World Smart Contract written in the Solidity language. This is the input language used in Chapter 5 and the output language used in Chapter 6.

2.3.5 Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine (EVM) is similar to the Java Virtual Machine, and is the sandboxed runtime environment for smart contracts in Ethereum. The EVM is used when

a transaction is directed at a Smart Contract address. Execution in the EVM is very limited, with no access to the network, filesystem, or other processes. Smart contracts even have limited access to other smart contracts and can only call functions with the label `public` or `external`.

Accounts

There are two kinds of accounts in the Ethereum Network: External accounts which are controlled by public-private key pairs (i.e. humans) and Smart Contract accounts which are controlled by the code stored together with the account.

The address of an external account is determined from the public key while the address of a contract is determined at the time the contract is created and deployed to the network (derived from the creator address and the transaction nonce). Each account type is treated equally by the EVM. Every account has a persistent key-value store mapping 256-bit words to 256-bit words called storage. Both account types have an associated balance in Ether.

Transactions

A transaction is a message that is sent from one account to another account. It can include binary data (called the "payload") and Ether. If the target account contains code, that code is executed and the payload is provided as input data.

If the target account is not set (the recipient is set to null), the transaction creates a new contract. The payload of a contract creation transaction is taken to be EVM bytecode and executed. The output data of this execution is permanently stored as the code of the contract. To create a contract, you do not send the code of the contract, but code that returns that code when executed.

A transaction, T, is a single cryptographically-signed instruction. Below we specify a number of common fields in transactions. More details of each transaction fields are found in [141].

- **nonce**: A scalar value equal to the number of transactions sent by the sender; formally T_n.
- **gasPrice**: A scalar value equal to the number of Wei to be paid per unit of gas for all computation costs incurred as a result of the execution of this transaction; formally T_p.

- **gasLimit**: A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later; formally T_g.
- **to**: The 160-bit address of the message call's recipient or, for a contract creation transaction, $\emptyset$, used here to denote the only member of $\mathcal{B}_0$; formally T_t.
- **value**: A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account; formally T_v.
- **v, r, s**: Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally T_w, T_r and T_s.

Gas

Each transaction provides a limit to the amount of gas the creator is willing to use to execute the transaction. All statements executed by the EVM costs a specific amount of gas. Gas is used to pay the miner who solved the previous block.

The gas price is a value set by the creator of the transaction, which is the conversion rate from gas to wei that they are willing to use. The creator has to pay `gas_price * gasLimit` up front from the sending account. The gas price allows miners to select which transactions to add to the next block, higher gas prices mean the transaction is more likely to be added to the next block.

The Gas Limit is too low If a transaction runs out of gas, then it is reverted back to its original state. The creator of the transaction must still pay the miner the fee for their computational costs; `gas_price * gasLimit`.

The Gas Limit is too high If an operation completes and there is leftover gas, the gas will get refunded to the creator. The process for refunding is complicated due to gas refunds when removing contracts and removing storage, which can lead to net gains in gas refunds for a transaction. The involved process for gas refunds is out of the scope of this thesis.

Block Gas Limit

The Block Gas Limit restricts the amount of gas that can be used for that block. The current process for how the Block Gas Limit is updated, is a voting system between the miners:

> "Currently, due to a lack of clear information about how miners will behave in reality, we are going with a fairly simple approach: a voting system. Miners have the right to set the gas limit for the current block to be within 0.0975% (1/1024) of the gas limit of the last block, and so the resulting gas limit should be the median of miners' preferences." [129]

Storage/Memory/Stack

There are three locations where data can be stored: Storage, Memory, and the Stack. In Solidity, we use keywords to indicate where variables should be stored.

Each account has a data area called storage, which is persistent between function calls and transactions. Storage is a key-value store that maps 256-bit words to 256-bit words. Storage is comparatively costly to read, and even more to initialize and modify storage. A contract cannot read or write to any storage apart from its own.

The memory data area is cleared for each message call. Memory is linear and can be addressed at byte level. Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (i.e. any offset within a word). There is a cost of gas to expand the memory size. Memory is more costly the larger it grows (it scales quadratically).

The EVM is a machine which utilizes the stack and has no registers. The stack data area has a maximum size of 1024 elements and contains words of 256 bits. Access to the stack is limited to the top end. It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it. All other operations take the topmost two elements from the stack and push the result onto the stack. It is not possible to access arbitrary elements deeper in the stack. It is possible to move stack elements to storage or memory in order to get deeper access to the stack.

2.3.6 Message Calls, Delegate Calls, and Call Code

There are 4 EVM call instructions which allow for a contract to invoke another contract.

CALL Contract D calls a function $E.f()$, where $E.f()$ runs in the context of contract E, the storage is E's storage.

CALLCODE Contract D calls a function $E.f()$, where $E.f()$ runs in the context of contract D, the storage is D's storage. The values of `msg.sender` and `msg.value` in E are

not the same as in D.

DELEGATECALL Contract D calls a function $E.f()$, where $E.f()$ runs in the context of contract D, the storage is D's storage. The values of `msg.sender` and `msg.value` in E are the same as in D. This gives contracts the ability to "update" contracts by changing the address of functions that are using a delegate call.

STATICCALL Contract D calls a function $E.f()$, where $E.f()$ runs in the context of contract E, but any attempt at modifying the state results in an exception (this also applies to any Calls made by $E.f()$).

2.3.7 The Halting Problem

In computing, the halting decision problem is: given a description of a program P and an input I, return if $P(I)$ will halt (finish running) or continue running forever. The halting problem is undecidable for Turing machines, and programming languages which are "Turing complete" (can be used to simulate any Turing machine). Programming languages like C, C++, Python, and Java are Turing complete (if we assume finite memory is omitted) and the halting problem is undecidable under each.

Smart contracts have strong programming abilities: branching statements, variable storage, loops. The language contains all features required to be considered a Turing complete language. Smart contracts would be Turing complete if it was run outside of the EVM. The halting program is decidable for all smart contracts. Each transaction contains the maximum amount of gas that transaction will consume, *Gas Limit*. Once a transaction runs out of gas it automatically halts. The maximum gas a transaction can consume is limited by the Block Gas Limit. The main Ethereum network has a gas limit ranging from 9,940,901 gas to 10,000,000 gas (as of May 2020).

Gas is calculated per EVM instruction. The gas limit ensures that all smart contract functions halt. If a smart contract function reaches the gas limit, the function will fail and rollback all changes it has made. Thus an infinite loop is not possible within the EVM.

2.4 Logic Locking

Logic Locking is a field in Computer Security which allows technology companies to protect their intellectual property (IP) when contracting circuit manufacturing. It protects the

circuits by incorporating an electronic key into the circuit, which will allow the circuit to function normally. The locked circuit is provided to the manufactures, but the key is not. This prevents the contractors from illegally over producing and selling the same product under a different name, and stealing the circuit description for use in another product. This technology relies upon the protection of the electronic key in sold units, so that this key cannot be stolen from a legitimately purchased product. Physically protecting the electronic key, and the key delivery bus to the locked circuit, are beyond the scope of this thesis.

2.4.1 Circuit Representations

```
# key=110
# also valid keys: 100, 010
INPUT(pi00)
INPUT(pi01)
INPUT(pi02)
INPUT(keyinput0)
INPUT(keyinput1)
INPUT(keyinput2)
OUTPUT(po0)
OUTPUT(po1)
OUTPUT(po2)
lo1 = or(pi00,pi01,pi02)
n01 = buf(pi02)
la1 = and(n01,pi01,pi02)
n02 = xor(keyinput0,lo1)
n03 = not(lo1)
n04 = xor(pi02, pi01)
n05 = or(n02,keyinput1)
n06 = not(keyinput2)
n07 = and(n06, n02)
n08 = not(pi00)
po0 = and(n07, n03, n05)
po1 = xnor(n06, la1)
po2 = or(la1, keyinput1, n08, n04)
```

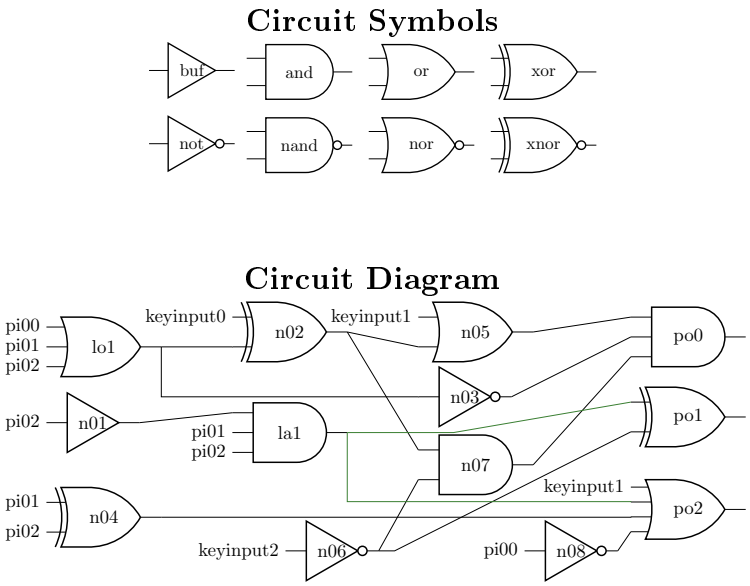


Figure 2.6: Example Circuit Netlist and Diagram. The netlist declares all inputs/outputs, followed by all logical gate definitions. This example has 3 keys which produce the correct outputs.

In Figure 2.6, we show 2 circuit representations: Circuit Netlist and a Circuit Diagram. These two equivalently represent the same circuit. The netlist is much easier for a computer to read, and the circuit diagram is easier for humans to read. All benchmark circuits used in this thesis are stored in netlist format.

The netlist format lists all inputs and outputs from the integrated circuit (IC) at the top of the file. Logic locked circuits can obfuscate which inputs are not inputs and which

inputs are key inputs. Protection methods are required for the key storage and the key bus. These mechanisms are expensive and can be incompatible with normal IC connection, thus it is assumed that the attacker is able to correctly identify the key inputs from the normal inputs.

The circuit in Figure 2.6 has the correct key of `110`, where `keyinput0=1`, `keyinput1=1`, `keyinput2=0`. There is an additional comment below which states that the keys `100` and `010` are also valid. Certain logic locking techniques produce locked circuits with more than 1 valid key. This reduces the key space and work required by the attacker.

2.4.2 Locking Techniques

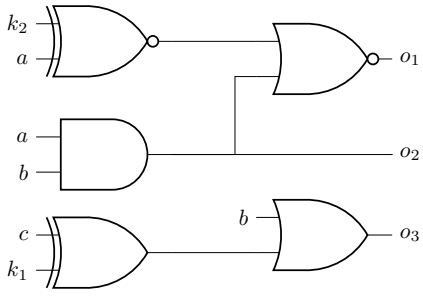
In this section we briefly describe 2 methods used in logic locking a circuit. These methods have be performed in many novel ways, we will only show the basics of how the XOR/XNOR and the MUX gates can be effective methods for logic locking circuits.

XOR/XNOR Gates

In Figure 2.7, we show a simple circuit that has been randomly locked with XOR and XNOR gates, this is similar to the EPIC method by Roy et al. [98]. As we can see, given the same fixed input, but different keys, the output is changed. This change might seem small, but in circuit design, changing a single bit can have significant ramifications to the functionality of the IC. Methods of modifying the circuit to incorporate the XOR and XNOR gates are varied and can be very complex.

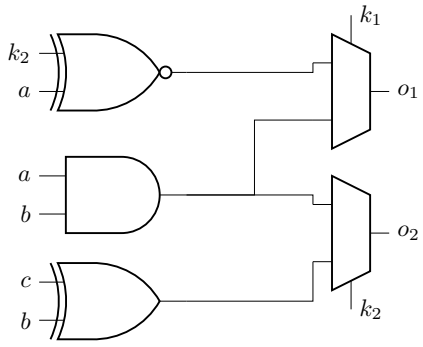
MUX Gates

The circuit shown in Figure 2.8 utilizes the multiplexer (MUX) gate to logic lock the circuit. A MUX gate selects which input to pass through the gate via the control inputs. In the figure we use the key bits as the control bits (the top and bottom of the MUX gates). When the control bit is 0, then the first input from the top is selected. If the control bit is 1, then the second input from the top is selected. The number of control bits is not restricted, but the number of inputs is exponentially bounded to the number of control bits. Thus we can use a 4 by 2 MUX gate, where there are 4 inputs and 2 control bits. A 8 by 3 would be the next size up, followed by a 16 by 4. This technique is similar to the one implemented in [89].



Input abc	**Key** k_1k_2	**Output** $o_1o_2o_3$
000	00	000
000	11	101
010	00	001
010	11	101
101	00	101
101	11	000
111	00	011
111	11	011

Figure 2.7: Simple example of XOR/XNOR locked circuit.



Input abc	**Key** k_1k_2	**Output** o_1o_2
000	00	10
000	11	00
010	00	10
010	11	01
101	00	00
101	11	01
111	00	01
111	11	10

Figure 2.8: Simple example of MUX locked circuit.

Part I

Problems Reduced to Model Checking

Chapter 3

Cree: a Performant Tool for Safety Analysis of Administrative Temporal Role-Based Access Control (ATRBAC) Policies

Contents

Declaration of Contributions

This chapter is a modified version of the paper [116], which was authored by: Jonathan Shahen (myself), Jianwei Niu, and Mahesh Tripunitara. The contributions contained within this chapter are my own, but without the help of my co-authors, the work would not have

been published as it was. The paper [116], was originally a journal extension for [115] (same authors), but during the review I rewrote the paper to completely focus on the Cree tool and remove any contributions made in [115].

3.1 Introduction

Access control deals with whether a principal may exercise a privilege on a resource; e.g. whether the user Alice is allowed to exercise the 'read' privilege on a file. It is an important aspect of the security of a system. Whether an attempted access is permitted is customarily specified in an access control policy. Such a policy may change over time; for example, Alice may, at some later time, lose the 'read' privilege to a file that she possesses.

Effecting and intuiting the consequences of changes to an access control policy is called administration. An aspect of administration is delegation, with which a trusted administrator empowers another principal to change the policy in limited ways. Delegation is used so administrative efficiency can scale with the size of an access control system. For example, a trusted administrator may delegate to a user Bob the ability to add users within his team, to a project Bob owns. Thus, Bob is able to control which members of his team can access the resources of a particular project without requiring assistance from a trusted administrator.

With delegation arises the need for safety analysis, which has been recognized as a fundamental problem in access control since the work of Harrison et al. [49]. Safety analysis asks, in the presence of delegation, whether some partially trusted users may effect changes to the authorization policy in a manner that a desirable security property is violated.

Safety analysis has been addressed for various access control schemes in the literature. Our focus is safety analysis in the context of Administrative Temporal Role-Based Access Control (ATRBAC) [133]. ATRBAC is an administrative scheme for Temporal Role-Based Access Control (TRBAC). TRBAC is Role-Based Access Control (RBAC) with temporal-extensions. In RBAC, rather than assigning a user directly to a permission, we adopt the indirection of a role. A user is authorized to a set of roles, and each role is authorized to a set permissions. The temporal extensions that TRBAC adds to RBAC constrain the time intervals that (i) a user may exercise a privilege, and, (ii) a role may be active. In addition, ATRBAC constrains the time intervals that (iii) an administrative action can change the authorization policy. (See Section 3.2 for more details.)

An instance of the safety analysis decision problem comprises the following three inputs. A concrete example of the scheme we consider, ATRBAC, is in Section 3.2.

(1) **Start State** – an instance of an access control policy, i.e., roles currently assigned to which users.
(2) **State Change Rules** – set of administrative rules by which a policy can change, e.g., rule: ⟨Bob may grant users membership to Bob's Project if that user is a member of Bob's team⟩.
(3) **Safety Query** – a statement that can judge if the system is secure or insecure; typically whether a particular user possesses a privilege. Examples: "Can Alice get write access to Payroll", "Can Bob adopt the role of Administrator", and "Can any user obtain both: read access to 'New Project' and write access to 'Public Repository'."

An instance of safety analysis is 'TRUE' if the safety query can never become true, i.e., the system is indeed safe, and 'FALSE' otherwise. The utility of such analysis is that a trusted administrator can assess whether the current, or a prospective, set of authorizations and delegations may lead to a violation of a desirable security property. She can then modify it if indeed such a violation may result.

ATRBAC-Safety is an important and non-trivial problem. We discuss this in the context of a concrete example for ATRBAC in Section 3.2 after we have presented the details of ATRBAC and safety-analysis in its context (paragraph titled, "The example, revisited"). Here we provide a broader discussion.

Jones [59] discusses why safety analysis can be a technical challenge: it is in the difference in the manner in which administrative rules and safety properties are specified. Administrative rules are "phrased in a procedural form", while safety properties are "formulated as predicates." "Though procedural definitions make individual system state transitions easy to understand and to implement, they combine to form a system that exhibits complex behaviour. It is difficult to intuit and to express the behaviour of a procedurally defined system."[59] As in the case of other access control schemes, safety queries in ATRBAC-Safety are in predicate form and the rules are in procedural form.

In addition to this difference in the manner, there is the issue of scale. Access control policies for enterprise can be large and thereby preclude manual safety analysis. For example, prior work [56] discusses an ARBAC policy, i.e., without the additional features of ATRBAC, of a financial company which comprises 1363 roles and 8885 rules. Such a policy is likely too large for manual safety analysis. We seek an automated approach.

Given the above discussions on the technical challenges that underlie safety analysis in general, one may ask what makes safety analysis in ATRBAC a particular challenge. That is, what technical challenges does our work specifically address that prior work on safety analysis, for example, in the context of ARBAC, does not? ATRBAC introduces new

syntactic constructs with associated semantics. We need safety analysis to "catch up" with these new constructs. As an example from the literature, Harrison et al. [49] originally proposed an administrative scheme for the access matrix model, and safety analysis results for it. Their work establishes that for their scheme, safety is undecidable. Sandhu [100] syntactically extends their work with the Monotonic Typed Access Matrix (MTAM) model. Safety analysis for MTAM, as it turns out, is decidable. The new features in ATRBAC are: (i) an administrative rule that specifies time periods during which (a) an administrative action may be effected, and, (b) a user is authorized to a role, and, (ii) two new kinds of administrative rules that are used to enable and disable administrative roles. It is unclear as to the manner in which these new features impact safety analysis. Our prior conference paper [115] establishes that the problem remains in **PSPACE**, and proposes an approach based on reduction to safety analysis in ARBAC. In this work, we reduce ATRBAC-Safety directly to model-checking. As our work establishes (see Section 3.6), and as we hypothesized at the start of this work, this approach is more efficient in practice. There are a number of technical innovations we have devised in carrying out this reduction, and complementing it with optimizations (see Section 3.5). While these are inspired by prior work, they are customized to ATRBAC-Safety. Nonetheless, our work in turn may be directly useful to future schemes, or at the minimum, provide inspiration on general directions as prior work has for this work.

Layout The remainder of this chapter is organized as follows. In the next section, we formalize and discuss the problem that this work addresses, ATRBAC-Safety. In Section 3.3, we discuss relevant prior work. In Section 3.4, we provide an overview of our work and contributions. In Section 3.5, we introduce our tool Cree and discuss the reduction to Model Checking and four optimizations that are inspired by, but different from, prior work: Polynomial Time Solving when possible, Static Slicing, Abstraction Refinement, and Bound Estimation. In Section 3.6, we discuss our empirical assessment and results across Cree, and five other prior tools. We conclude with Section 3.7.

3.2 ATRBAC-Safety Background

The Access Control and ATRBAC-Safety background has been moved to Section 2.2.

3.3 Prior work

Prior work that is relevant to ours can be dichotomized into: (a) work on access control models and schemes, such as RBAC and its variants, and, (b) work on safety and security analysis in the context of such models and schemes. A comprehensive survey of these is beyond the scope of this work. In this section, we discuss prior work from the standpoint of safety analysis, (b).

The work of Harrison et al. [49], in the context of the HRU access matrix model, is, to our knowledge, the first to pose and address safety analysis in the context of access control. Since that work, there has been work on safety analysis for variants of the HRU model [49], and other models such as those for trust management and negotiation [19, 66], usage control models [91, 92], and schemes based on RBAC [37, 38], including ARBAC [99, 126]. There has been work also in generalizing safety analysis to so-called security analysis [68], and the use of safety and security analysis to characterize the expressive power of authorization schemes [4, 130]. Extensions have been made with regards to ATRBAC safety in [134], most notably the security properties: availability, liveness, and mutual exclusion of privileges for TRBAC.

As for ATRBAC-Safety, the topic of this work, we are aware of the following prior work. Our prior work, [115, 107], identifies that safety analysis of ATRBAC is **PSPACE**-complete. In addition, it proposes an approach for safety analysis of ATRBAC based on reduction to ARBAC and then use of a prior solver for ARBAC-safety; we call this prior approach Mohawk+T. Our approach in this work, Cree, was designed and created after we gathered experience with Mohawk+T. Our experience with Mohawk+T suggested that directly reducing to model-checking in conjunction with other improvements, such as static slicing/abstraction refinement/bound estimation, would yield better performance. Thus, the work in this chapter is different from our prior work [115] – the two propose different tools, with different approaches to the problem.

The work of Uzun et al. [133] is, to our knowledge, the first work to propose ATRBAC and pose the safety-analysis problem for it. In addition, that work discusses the design of two software tools, TredRole and TredRule to address instances in practice. These tools are part of our empirical assessment in Section 3.6. The work of Ranise et al. [93] syntactically generalizes some aspects of the version of ATRBAC from Uzun et al. [133]. It then presents a result on the computational-complexity of ATRBAC-Safety — it proves that the problem is decidable. It then discusses the design, construction and evaluation of 2 different software tools, AsaspTime-Sa and AsaspTime-Nsa, to address problem instances in practice. These tools are also part of our empirical assessment in Section 3.6.

Safety analysis has been applied to ATRBAC where role hierarchies are allowed in [94, 131, 95]. Safety analysis with role hierarchy was also been applied to ARBAC in [131]. This suggests a limitation in Cree — we do not directly address role hierarchy. We hypothesize that an enhancement to our reduction is possible that incorporates role hierarchy and still yields an efficient approach in practice. We leave this for future work.

Techniques used in this chapter include static slicing, abstraction refinement, and bound estimation. For these techniques, we are inspired by prior work on ARBAC-Safety, but with different algorithms to address the new technical challenges posed by ATRBAC's new features. Jha et al. [57] propose forward and backward pruning for ARBAC-safety. This was augmented by Jayaraman et al. [56] with abstraction refinement and bound estimation. In [38], Ferrara et al. provides methods of pruning and bound estimations for ARBAC-Safety. Ferrara extends the forward- and backwards-pruning from Jha et al. [56]. They include more exclusion rules, to create a more aggressive static slicing method. Ferrara's bound estimation found an upper bound on the number of users to be the number of administrative users plus 1. We implement our own versions of static slicing, abstraction refinement, and bound estimation. This is due to the difference between ARBAC-Safety and ATRBAC-Safety. ATRBAC-Safety includes timeslot/time-intervals within to *can_assign* and *can_revoke* rules, and introduces 2 new rule types: *t_can_enable* and *t_can_disable*.

3.4 Our work

In our prior work [115], we propose a tool called Mohawk+T for ATRBAC-Safety. Mohawk+T reduces ATRBAC-Safety to ARBAC-Safety, and then uses the solver, Mohawk [56], that was built for ARBAC-Safety. Our empirical assessment of Mohawk+T suggests that while it is superior to prior tools for some classes of inputs, there are other classes of inputs where the other tools outperform it. This leads us to ask: is there a fundamentally different design we could adopt for ATRBAC-Safety that results in a tool that is more performant? Our answer to this question is 'yes,' and Cree, which we discuss in this work, is such a tool.

In Cree, we reduce ATRBAC-Safety directly to model checking. Such a direct reduction gives us performance gains. In addition, it gives us an avenue to design and implement other performance improvements: Polynomial Time Solving when possible, forward- and backward-pruning, abstraction refinement, and bound estimation (see Section 3.5 for the reduction to model checking, and these algorithms). Thus, Cree is a "from scratch," new tool for ATRBAC-Safety, and as our empirical results in Section 3.6 establish, it is superior to both Mohawk+T and other prior tools for several classes of inputs, and no worse for

others. The format of ATRBAC-Safety that Cree supports is the same as the one Mohawk+T formalized, and this version generalizes the tools from prior work [133, 93] (see Table 1 in [115]).

Static slicing is a method of reducing the input access control policy by removing rules/roles/timeslots from the policy, which has been shown to greatly improve performance. Static slicing has been discussed for ARBAC in [57, 126, 56, 38], but ATRBAC introduces not only time-intervals/timeslots to each *can_ assign* and *can_ revoke* rule, but introduces 2 new rule types *t_ can_ enable* and *t_ can_ disable*. These additions require new algorithms for static slicing. We outline our forward and backwards pruning techniques in Section 3.5.2.

Abstraction refinement is an optimization technique for solving problems by solving sub-problems with positive one-sided error. Abstraction refinement starts with the smallest sub-problem, solves the problem and halts if we get a positive answer (the system is unsafe). Otherwise we refine the sub-problem and make it bigger by adding rules and repeat until we halt or have tested a policy equivalent to the original policy. The hope of abstraction refinement is that the solution to the original problem can be solved using only a small percentage of the original problem. Abstraction refinement allows parallel execution for ATRBAC-Safety. We have provided our abstraction refinement technique in Section 3.5.3.

Our static slicing and abstraction refinement techniques have the ability to produce low complexity ATRBAC-Safety problems. We leverage this by creating an optimization technique called Polynomial Time Solving when possible. This technique is a set of linear/low-order-polynomial timed algorithms that are able to quickly solve specific classes of input of ATRBAC-Safety polices. The class of randomly generated inputs, created in [133], is able to be solved almost exclusively using this technique. When this technique is able to solve our policy, we are able to skip running the model checker, which has high overhead cost. We discuss our Polynomial Time Solving algorithms in Section 3.5.1.

The previous 3 techniques can be used in conjunction with all ATRBAC-Safety solvers. Our last technique can only be applied to solvers where maximum path length is taken into consideration. Our last optimization technique is Bound Estimation. Where, given an ATRBAC-Safety policy, we calculate an upper-bound on the path length for the smallest path from initial state to goal state. The upper bound from Bound Estimation can be used by the model checker NuSMV with no modification. We discuss our Bound Estimation upper bound calculation in Section 3.5.4.

Cree is written in Java and is able to run on all modern operating systems. We have provided Cree's source code for public download [103].

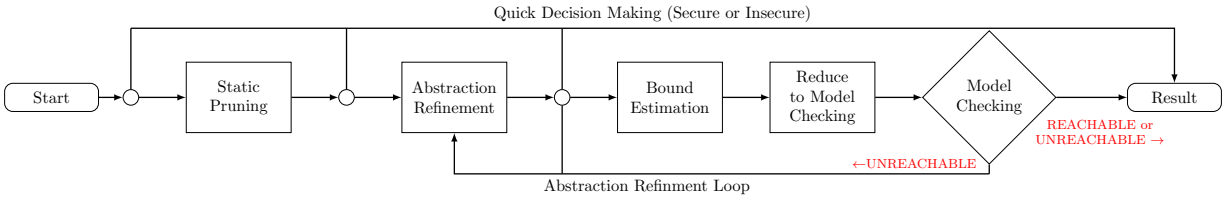


Figure 3.1: Flow diagram for the reduction from ATRBAC-Safety to Model Checking. The Polynomial Time Solving are run at each of the white circles. We translate the Model checking results as follows: REACHABLE inputs are UNSAFE; UNREACHABLE inputs are SAFE. This diagram represents the tool Cree.

3.5 Cree

We now discuss our tool, Cree. Cree's ATRBAC-Safety input format is the same as that of Mohawk+T [115] which in turn generalizes those of prior tools [133, 93]. Following is a list of how the safety query and rules are formatted. The following definitions differ from the theoretical definition described in Section 2.2.2, as prior work tools do not support all theoretical features of ATRBAC-Safety. The supported features of prior work tools vary and I describe their differences in Section 2.2.2 under the paragraph "Versions of the problem".

Security Query The query is a tuple $\langle s_q, R_q \rangle$, where s_q is a timeslot and R_q is a set of positive roles. The safety query is satisfied if any user u is assigned to every role in R_q for the time-slot s_q.

Initial Conditions All ATRBAC-Safety instances have the same initial conditions. These are: (1) The user–role assignment set, TUA, is empty, and (2) The role–enablement set, RS, is empty.

Rules Every rule takes the format $\langle a, L_a, C_t, S_t, t \rangle$. (1) a is an administrative role or the mnemonic 'TRUE'. (2) L_a is a time-interval, ex: $t1 - t5$, or the mnemonic 'T_{all}'. (3) C_t is a set of positive and/or negative roles. (4) S_t is a set of time-slots. (5) t is a role that is to be assigned/revoked/enabled/disabled. Please refer to Section 2.2.2 to compare formats of the implemented and theoretical rules.

Cree solves instances of ATRBAC-Safety by reducing directly to Model Checking and then running a state-of-the-art model checker, NuSMV [78]. This gives us more control over, and insight into, input instances that are harder for our previous tool (Mohawk+T) when compared to prior tools. In the next few sections, we present a number of techniques

that exploit this stronger control we have. Specifically, we are able to design and incorporate a number of complementary techniques into Cree that make it considerably more scalable than without those techniques. How Cree operates is shown in Figure 3.1. The following sub-sections discuss the constituent modules in Cree that are shown in the figure.

3.5.1 Polynomial Time Solving when possible for ATRBAC-Safety

In practice there are many classes of input where the solution can be determined by using a polynomial timed algorithm. The choice to perform these quick solvers frequently throughout Cree's operation provides many opportunities for increased performance by identifying these classes and skipping labour intensive analysis. The quick solvers run after each modification to the ATRBAC-Safety policy.

The following Polynomial Time Solving are performed at each white bubble in Figure 3.1. The order of execution is important.

(1) Empty Query Role Array Such an input is unsafe when no roles are provided in the query's role array. The result is true for all instances because all users satisfy the condition, regardless of the roles they are members of. Run time of $\Theta(1)$.

(2) No Can Assign Rules for the Security Query's Role Array and Timeslot Such an input is safe because the initial conditions have no users assigned to any roles and with no $t\_\ can\_\ assign$ rules to obtain the query's roles, for the query's timeslot, it is impossible for a user to obtain the goal roles. Run time of $\Theta(|t\_\ can\_\ assign|)$.

(3) No "Truly-Startable" Can Assign Rules A "Truly-Startable" rule is one that has C_t = TRUE and a = TRUE. This means that a Truly-Startable $t\_\ can\_\ assign$ rule is satisfied in the initial conditions. If no Truly-Startable $t\_\ can\_\ assign$ rules exists then the initial user-role assignment TUA can never be changed. Thus, such an input is safe. Run time of $\Theta(|t\_\ can\_\ assign|)$.

(4) Truly-Startable CanAssign Query Rules The safety query is reachable if there exists a set of truly-startable rules that can satisfy the safety query. This has a run time of $\Theta(|t\_\ can\_\ assign|)$ and can be combined with (2) and (3) for parallel iteration of the same list.

3.5.2 Static Slicing

Static Slicing is a method of removing Rules, Roles, and Timeslots from an ATRBAC-Safety policy that are irrelevant for safety analysis. Static slicing is one of the first steps to be performed, because reducing policy size can increase performance and minimizes memory usage for all operations following. Figure 3.2 shows an example of a policy that can be made smaller without affecting the safety analysis. Specifically, the policy has several rules (highlighted in red) that can be removed without impacting safety analysis. We are able to remove these rules using a combination of forward and backwards pruning. This, in turn, can improve performance by reducing the memory size required to store each state, and by reducing the number of states needed to search. Static slicing has been discussed in ARBAC [57, 126, 56, 38]. We have modified the methods of forward and backwards pruning, from [57, 126, 56], to update the algorithms for the introduction of time into *t_ can_ assign*/*t_ can_ revoke* rules and to apply similar pruning methods to *t_ can_ enable*/*t_ can_ disable* rules (which do not exists in ARBAC). Both pruning techniques can be turned off in Cree, but due to empirical evidence, both techniques were turned on to obtain the best results in Section 3.6.

Forward Pruning is a technique which ignores the safety query and focuses on removing rules that are unable to fire. It does this by looking at every rule in the policy and verifying if the admin role is assignable and enable-able, and if each positive role in C_t is assignable. If any of these are false then the rule cannot fire and we can safely remove the rule from the policy. The previous assertion is only true if we start from an empty state, or if we start from a state where every user has obtained their roles using the current set of rules.

ATRBAC forward pruning is in P, our algorithm has a run time of $\Theta(|\text{rules}|)$. Our forward pruning algorithm differs from previous works by introducing the *t_ can_ enable* *t_ can_ disable* pruning and adding the condition that administrator roles must also be enable-able. The forward pruning algorithm is supplied in Section C.1.

Backward Pruning focuses on the safety query, $\langle R_q, s_q \rangle$, to determine which rules it can add to a new policy P''. Backward pruning collects all *t_ can_ assign* rules, $\langle a, L_a, C_t, S_t, t \rangle$, where the target role t equals a role in the safety query's role set R_q and the timeslot array S_t contains the safety query's timeslot s_q. These initial *t_ can_ assign* rules is the set of rules that is required to obtain each of the goal roles in the safety query. Not all of the rules in this set are necessary to satisfying the safety query, but a subset of these *t_ can_ assign* rules are required. From this initial set we extract all of the conditions that each rule requires in order to fire: (1) all positive roles in C_t must be satisfied for all timeslots in S_t, (2) all negated roles in C_t, that also appear as a target role in some *t_ can_ assign* rule,

```
// Path: CE1(admin) → CE3(a) → CA6(a) → CA6(user) → CA4(u) → CR2(u) → CA2(u) → CA6(u)
Query : t2, [r3, r4]
CanAssign:
  /*CA1*/ <TRUE, t1-t3, TRUE, [t2,t3], r1>
  /*CA2*/ <r3, t1-t3, r2 & NOT r3, [t2,t3], r4>
  /*CA3*/ <r1, t1-t3, r1, [t2,t3], r5>
  /*CA4*/ <r3, t1-t3, r3, [t1], r2>
  /*CA5*/ <r1, t1-t3, r5 & NOT r3, [t2,t3], r6>
  /*CA6*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r3>
CanRevoke:
  /*CR1*/ <TRUE, t1-t3, TRUE, [t1,t2], r1>
  /*CR2*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r3>
  /*CR3*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r2>
CanEnable:
  /*CE1*/ <TRUE, t1-t2, TRUE, [t1], r1>
  /*CE2*/ <TRUE, t1-t2, TRUE, [t2], r1>
  /*CE3*/ <TRUE, t1-t2, r1 & NOT r2, [t1], r3>
  /*CE4*/ <TRUE, t1-t2, r1, [t1], r2>
CanDisable:
  /*CD1*/ <TRUE, t1-t2, TRUE, [t1], r1>
  /*CD2*/ <TRUE, t1-t2, TRUE, [t1], r2>
```

Figure 3.2: Example ATRBAC-Safety policy which highlights the rules that can be removed without changing the safety analysis. All lines highlighted in red are not required to show that this policy is unsafe. Only the lines highlighted in green are required. Removing all unnecessary rules/roles/timeslots with static slicing ensures: (1) the pruned policy's size will be less than or equal to the input policy's size, and (2) the safety response remains unchanged.

must be revocable (if possible), (3) all administrator roles a must be assigned (matching t_can_assign rule(s)) and enabled (matching t_can_enable rule(s)). We generate the first t_can_assign set, extract all conditions that the rules require to fire, and then find all corresponding rules that can satisfy these conditions. We loop this process until all conditions are possibly met; some policies have conditions that cannot be satisfied because no satisfying rule exists.

ATRBAC backward pruning is in P, our algorithm has a run time of $\Theta(|\text{rules}|^2)$, in the worst case. Our backward pruning algorithm differs from previous works by introducing $t\_can\_enable/t\_can\_disable$ rules, and the more difficult addition of introducing time to every satisfying condition. The algorithm for backward pruning is provided in Section C.1.

3.5.3 Abstraction Refinement

Abstraction Refinement is a technique that can increase performance of safety analysis for insecure policies. It uses the assumption that only a small portion of an ATRBAC policy is required to determine if a policy is insecure. Abstraction refinement works by creating a very small policy P_0 from a policy P, then performing safety analysis. If the small policy P_0 is insecure, then we know P is insecure and can halt early. If P_0 is secure, then we don't know if P is secure or insecure and thus we create a slightly bigger policy P_1 and repeat. We repeat until we find a policy that is insecure or we do a safety analysis on the original policy P.

In static slicing we removed rules that could not fire in any situation and rules not relevant to the safety query, this ensures the security is unchanged due to pruning for all instances of ATRBAC-Safety. Abstraction refinement has 1-sided error, if a sub-policy P_i is insecure, then the original policy is also insecure. For secure policies we must perform security analysis on all sub-policies and the original in order to confirm that the policy is secure.

In this section we devise a strategy to produce a small sub-policy and then iteratively increase the policy size by adding more rules from the original policy until the original policy is reproduced without any irrelevant rules.

Abstraction Refinement improves performance for instances where only a small subset of rules are required to prove that a policy is insecure. In cases where a large set of rules is required for an insecure result, or if the policy is secure, this option would have a negative impact on the run-time. We can eliminate the negative performance impact by running the original policy in parallel with the abstraction refinement steps; halting as soon as one finishes. Abstraction Refinement can be turned off and is suggested if the ATRBAC-Safety policy is assumed secure.

The aggressiveness of the initial policy and the iterative steps can effect performance. More aggressive algorithms reduce the number of abstraction refinement steps by allowing more rules with each iteration, thus speeding up performance for secure policies. Less aggressive algorithms reduce the load on the solver by producing smaller and easier to solve sub-policies, thus speeding up performance for insecure policies. We have found a balance that works for the empirical testing set we used. To have an apples-to-apples comparison, Cree was run with abstraction refinement turned on for all empirical results reported in this chapter. All algorithms below run in polynomial time, and the number of abstraction refinement steps is bounded linearly by the number of rules. Thus Abstraction Refinement is in P.

ALGORITHM 1: Step 0 for Abstraction Refinement

Input : $P \leftarrow$ Cree Policy

Result: $P_0 \leftarrow$ Reduced Cree Policy (Size: $|P_0| \leq |P|$)

Func `AbsRef_Step0(P)`

$P_0 \leftarrow$ Empty Policy; P_0.query $\leftarrow \langle s_q, R_q \rangle \leftarrow P$.query;

$R_0 \leftarrow \{P.R_q\}$; $T_0 \leftarrow \{P.s_q\}$;

for $r \in$ *all rules in* P **do**

  if $(r.t \in R_0) \wedge (\exists s | s \in r.S_t \wedge s \in T_0)$ **then** $P_0 \leftarrow P_0.rules \cup r$;

Step 0: Initialization and Smallest Policy

Algorithm 1 shows how that initial abstraction refinement step is performed and the smallest sub-policy, P_0, is created. P_0 is a new empty policy, with the same safety query as the policy. We first create 2 variables: R_0 and T_0. $R_0 = \{$all of the roles from the Query$\}$. $T_0 = \{$timeslot from the Query$\}$. P_0 contains the set of rules from the original policy P if the rule satisfies the following conditions: the target role is in R_0, and at least 1 time-slot s in the target timeslot array must be in T_0 $(\exists s | s \in r.S_t \wedge s \in T_0)$.

Empirically we have found that this semi-aggressive algorithm performed best. Other initial algorithms considered were: limiting rule selection to only CanAssign rules, removing the timeslot restriction (using the ARBAC abstraction refinement initial step in [56]), and skipping the initial step and using the first refinement iteration.

Step N: Iterative Refinement

Algorithm 2 shows how iterative sub-policies are created from the previous sub-policy, P_{N-1}, and the original ATRBAC-Safety policy P. The initial abstraction refinement step uses the safety query to add all rules from P where $t \in R_q$ and $s_q \in S_t$. Each iterative step after that increases the set of target roles to include all roles administrative a and roles in precondition C_t from all rules in the previous abstraction refinement step. Increasing the target role set and the timeslot set increases the number of rules that can be added to the new policy.

Algorithm 2 is a semi-aggressive algorithm and was chosen based on empirical evidence. A less aggressive alternative is similar to Tightening 3 in Section 3.5.4, where R_N is split into 2 sets, roles that need to be assigned and roles that require enablement. A more aggressive alternative would be the same as the ARBAC scheme in [56], where there is no time constraint.

ALGORITHM 2: N^{th} Step for Abstraction Refinement

Input :

- P and P_{N-1} – Cree Policies
- R_{N-1} – Set of Roles from P_{N-1}
- T_{N-1} – Set of Timeslots from P_{N-1}

Result: P_N - Reduced Policy (Size: $|P_{N-1}| \leq |P_N| \leq |P|$)

Func `AbsRef_StepN`$(P, P_{N-1}, R_{N-1}, T_{N-1})$

P_N.query $\leftarrow P_{N-1}$.query; $R_N \leftarrow R_{N-1}$; $T_N \leftarrow T_{N-1}$;

for $r \in$ *all rules from* P_{N-1} **do**

$R_N \leftarrow R_N \cup r.t \cup r.C_t \cup r.a$;

$T_N \leftarrow T_N \cup r.S_t \cup r.L_a$;

for $r' \in$ *all rules from* P **do**

if $(r'.t \in R_N) \wedge (\exists s | s \in r'.S_t \wedge s \in T_N)$ **then**

$P_N \leftarrow P_N.rules \cup r'$

Step i	ΔR_i	ΔT_i	Δ **Rules**
0	r_3, r_4	t_2	CA2,CA6,CR2,CE3
1	r_2, r_1	t_1, t_3	CA1,CA4,CR1,CR3,CE1, CE2,CE4,CD1,CD2
2	$\emptyset$	$\emptyset$	$\emptyset$

Table 3.1: Example Abstraction Refinement steps using Algorithm 1 for step 0 and Algorithm 2 for steps 1 and 2. Notice that the columns are showing the difference in sets from the current step and the previous step. It should be noted that the policies below will be reduced if static slicing is applied before step 0.

Step M: Termination The last refinement step occurs when $|P_{M-1}.rules| = |P_M.rules|$. This usually occurs when $R_{M-1} \cap R_M = \emptyset$ and $T_{M-1} \cap T_M = \emptyset$. Note that $|P_M.rules| \leq |P.rules|$. $|P_M.rules| < |P.rules|$ can occur when there exists rules in P that do not help a user satisfy the safety query. These rules will not be added to P_M. Table 3.1 shows the abstraction refinement steps when applied to the example policy in Figure 3.2. In that example, the rules CA3 and CA5 are not in the last abstraction step. If we had performed static slicing before abstraction refinement we could have reduced the size of the last policy by 5 rules (rules highlighted in red in Figure 3.2).

3.5.4 Bound Estimation

NuSMV has two modes of operation: Symbolic Model Checking (SMC) and Bounded Model Checking (BMC). SMC mode combines states that share aspects in the state tree and utilizes these groups to traverse the tree more efficiently then using single step. BMC mode uses single step to traverse the state tree, but limits the distance from the initial state. In BMC mode, an additional integer input, called a *bound*, is required by NuSMV. This bound changes the problem to whether an unsafe state can be reached within that diameter from the start-state. To ensure we get a correct answer to our original safety query, we require a bound significantly large enough that if unsafe states exists, then at minimum 1 unsafe state must exist within the state tree with our fixed bound. Tightening this bound allows NuSMV to run more efficiently in BMC mode.

Part of Cree is an algorithm for estimating this bound; we call this algorithm bound estimation. It works by calculating an initial upper bound and then applying what we call tightenings that reduce this upper-bound, while maintaining the invariant that the original input is safe if, and only if, it is safe with our estimate for the bound. In practice, the initial estimate tends to be loose upper bound, with the tightenings gradually decreasing it.

Cree has the option to use SMC mode and forgo bound estimation; all results in Section 3.6 were found using BMC mode and bound estimation. If we used only the initial upper bound, we found that SMC was empirically quicker on average, but when utilizing tightening 3 we found BMC mode to be on average the quicker mode.

Initial Upper Bound calculates the length of a simple path that visits every possible state in an ATRBAC policy. In the context of ATRBAC, a state represents the user-role assignments and the role-enablement status; i.e. which users are assigned to which roles for which timeslots and which roles are enabled for which timeslots. We can represent this state as a binary string, where the length is equal to |User Role Assignments| + |Role Enablement| = (|Users| · |Roles| · |Timeslots|) + (|Roles| · |Timeslots|). From this we can calculate the maximum simple path to visit every state; the length of this simple path is calculated using $2^{|\text{Binary String}|}$. If the User-Role Assignment or the Role Enablement status had a more complex state than on/off, then the above calculation for simple path would not be accurate.

We require a *Users* variable to compute the initial upper bound. If there exists no known limit to the number of users to a specific ATRBAC policy, then we can use the worst case: in a single state we have 1 user able to represent every role for every single timeslot at once (|Users| = |Roles| · |Timeslots|), thus we have the ability to solve every

possible administration condition in an ATRBAC policy.

User-Role-Timeslot Assignment:

$$\begin{aligned} u &= |\text{Users}| \cdot |\text{Roles}| \cdot |\text{Timeslots}| \\ &= (|\text{Roles}| \cdot |\text{Timeslots}|) \cdot |\text{Roles}| \cdot |\text{Timeslots}| \end{aligned}$$

Role-Timeslot Enablement:

$$r = |\text{Roles}| \cdot |\text{Timeslots}|$$

Upper Bound:

$$d_0 = 2^{u+r} = 2^{(|\text{Roles}|^2 \cdot |\text{Timeslots}|^2) + (|\text{Roles}| \cdot |\text{Timeslots}|)}$$

Tightening 1: Required Number of Users The initial upper-bound uses the variable *Users*, which is not defined in a standard ATRBAC policy, thus it uses a large value which it knows will contain enough users. This tightening defines a tighter upper-bound by reducing the number of users required. It is important to notice that in an ATRBAC policy User A cannot effect User B's assignments unless User A acts as an administrator; i.e. there is no condition in our CA/CR rule which checks which roles another user is assigned to, we only have conditions checking the administrator and the target user. From this observation we can limit the number of users that we need to satisfy in our simple path to the number of administrative roles (1 user per role) and 1 user target user. To intuit the proof, in the worst case we require that each administrator role be assigned to a different user (i.e. *t_ can_ assign* rules: $\langle \text{TRUE}, t_{all}, \neg a_1 \wedge \neg a_2 \wedge \ldots \wedge \neg a_m, [t_0], a_0 \rangle \ldots$, no *t_ can_ revoke* rules). This means that we require the number of administrator users and 1 target user. We then calculate the length of a simple path to visit all states for all users, as this ensures that if an unsafe state exists then it will exist within our bound. This result is similar to that reported in [38], for the number of users required for analysis in an ARBAC policy.

$$\begin{aligned} |\text{Users}| &= |\text{Admin Roles}| + 1 \\ d_1 &= 2^{(|\text{Admin Roles}|+2) \cdot |\text{Roles}| \cdot |\text{Timeslots}|} \end{aligned}$$

Tightening 2: Leveraging Initial Conditions The initial conditions of an ATRBAC policy is that every user is assigned with no roles and every role is disabled. We can infer that the only rules that can change the initial condition is *t_ can_ assign* and *t_ can_ enable* rules. To get an intuitive sense of how this is true, imagine every possible user to a system

User-Role-Timeslot Assignment:

$$|u_2| = (|\text{Admin-Roles}| + 1) \cdot |\text{CA-Target-Roles} \cap (\text{CA-CR-Positive-Precondition} \cup \text{Admin-Roles} \cup \text{Goal-Roles})| \cdot |\text{CA-Target-Timeslots}|$$

Role-Timeslot Enablement:

$$|r_2| = |\text{CE-Target-Roles} \cap (\text{CE-CD-Positive-Precondition} \cup \text{Admin-Roles})| \cdot |\text{CE-Target-Timeslots}|$$

Upper Bound:

$$d_2 = 2^{|u_2|+|r_2|}$$

Figure 3.3: Tightening 2, reduces the required number of roles and timeslots required by the Assignment and Enablement portions of the upper bound. By leveraging the initial state of ATRBAC-Safety analysis, we can determine that unless a *t_ can_ assign* rule or *t_ can_ enable* is executed, then all *t_ can_ revoke* or *t_ can_ disable* rules will not change the state tree. This allows us to limit the number of roles to only the role that are required to be assigned and required to be enabled. We further limit this by only including the roles that are able to be assigned and able to be enabled. We limit the timeslots based on just on the *t_ can_ assign* and *t_ can_ enable* rules.

as a 3 dimensional array: users$\times$roles$\times$time-slots $\mapsto \{0,1\}$, and imagine whether a role is enabled or disabled as a 2 dimensional array: roles$\times$time-slots $\mapsto \{0,1\}$. The first array, a specific user u, role r, and timeslot t point to a binary variable stating whether user u is a member of role r for the timeslot t. The second array is similar but indicating if a role is enabled for a specific timeslot. The initial state sets all of these binary values to 0, thus any call to any *t_ can_ revoke* or *t_ can_ disable* will not change the state ($\underset{init}{\longrightarrow} 0 \underset{CR}{\longrightarrow} 0$). In order for *t_ can_ revoke* or *t_ can_ disable* to change state, a specific *t_ can_ assign* or *t_ can_ enable* call must precede it ($\underset{init}{\longrightarrow} 0 \underset{CA}{\longrightarrow} 1 \underset{CR}{\longrightarrow} 0$). From this intuitive knowledge, and by building on tightening 1, we obtain the upper bound in Figure 3.3.

From our observation, we can limit the number of roles and the number of time slots required by tightening 1. For User-Role-Timeslot Assignment, we only require to visit roles which we might need to obtain as a target user or as an administrator. The list of roles can be found in the set (CA-CR-Positive-Precondition $\cup$ Admin-Roles $\cup$ Goal-Roles). We can limit this set to only include roles which can actually be assigned using *t_ can_ assign* rules in the ATRBAC policy (i.e. target roles of all *t_ can_ assign* rules). We can limit the number of timeslots required to be the set of target timeslots which appear in all *t_ can_ assign* rules. We then perform similar tightenings for the Role-Timeslot Enablement path.

Tightening 3: Longest Simple Path to Goal State Tightening 3 uses the longest

$$U_{goal} = \left| \text{CA-Target-Roles} \bigcap \left[\bigcup_{g \in \text{Goal-Roles}} \text{Get-Roles}[LSP_{CA,CR}(g)] \right] \right| \cdot \left| \bigcup_{g \in \text{Goal-Roles}} \text{Get-Timeslots}[LSP_{CA,CR}(g)] \right|$$

$$U_{admins} = \left[\sum_{a \in \text{Admin-Roles}} |\text{CA-Target-Roles} \cap \text{Get-Roles}[LSP_{CA,CR}(a)]| \cdot |\text{Get-Timeslots}[LSP_{CA,CR}(a)]| \right]$$

$$RE = \left| \text{CE-Target-Roles} \bigcap \left[\bigcup_{a \in \text{Admin-Roles}} \text{Get-Roles}[LSP_{CE,CD}(a)] \right] \right| \cdot \left| \bigcup_{a \in \text{Admin-Roles}} \text{Get-Timeslots}[LSP_{CE,CD}(a)] \right|$$

$$d_3 = 2^{U_{goal}+U_{admins}+RE}$$

Figure 3.4: Tightening 3, calculates a diameter for NuSMV by returning the longest simple path for a user to be assigned the goal roles and for all required administrative roles to be assigned and enabled. We calculate the possible simple paths by working backwards from the goal roles and using rule preconditions to add rules to the simple path. The function $LSP_{x,y}(r)$ returns a set of rules from the input ATRBAC policy's $t\_can\_assign/t\_can\_revoke$ $(x, y = CA, CR)$ or $t\_can\_enable/t\_can\_disable$ $(x, y = CE, CD)$ sections, that represents the longest simple path to the role r.

simple path from the initial state to the goal state as a means of measuring the maximum number of role/timeslots required by the target user, each admin user, and for the role enablement. Using these roles, we calculate our bound as the length of a simple path which visits every state.

To understand the bound in Figure 3.4, we split the required state size into three sections: (1) a target user obtaining the goal roles, (2) other users obtaining required administrator roles, and (3) having each administer role enabled at the right time. The longest simple path (LSP) is calculated by looking at all the rules that assign, or enable, a specific state (i.e. role a for timeslot t_3) and then working backwards, to collect all rules that a target user require and returning the longest path. This is similar to backwards pruning in Section 3.5.2. We identify rules that might be required based on the positive preconditions. We do not consider the assignment or enablement of admin role in the LSP algorithm, as they are specifically handled in U_{admin} and RE.

The intuition behind tightening 3 is to minimize the number of role/timeslots variables in our state representation for the upper bound on the minimum length path from initial to goal state. Tightening 1 reduced the number of users required to solve all ATRBAC policies. From tightening 2, we recognize that if there exists no $t\_can\_assign/t\_can\_enable$ rule for a particular role/timeslot, then we can remove it from our state regardless if there exists a t_can_revoke $t\_can\_disable$ rule. Tightening 3 utilizes these observations and applies them based on the safety query.

We notice that to achieve our goal state, we must have a user contain all goal roles for the goal timeslot. In order for this user to obtain the above, we must have administrator users able to execute rules on the user. In order for an administrator user to fire a rule, their administrative role must be enabled. This is how the three steps above were created. For each step we determined the longest shortest path (LSP) from the initial conditions to a state (i.e. user obtaining a single goal role for the goal timeslot, or an admin user obtaining their admin role for a specific timeslot). We use the LSP algorithm to determine the maximum number of role/timeslots to keep for that particular user. The exact roles/timeslots are not important, as we are only using this as a method of estimating the minimum path length from the initial state to the goal state, and not actually changing the underlining TRBAC state.

In the previous 2 tightenings, we assumed each administrator user required the same number of states. In this tightening we calculate the required state size for each administrator user and add the sizes together (see U_{admins} in Figure 3.4). In the worst case, we find that all administrative users require the same of role/timeslots and thus see no improvements from tightening 2. We modify the Role Enablement state size by limiting the number of roles to those required to enable the admin roles.

From Figure 3.4, below we explain the functions used. Get-Roles() is a polynomial time function that takes a set of rules and returns the set of roles that contains all target roles and positive preconditions. Get-Timeslots() is a polynomial time function that takes a set of rules and returns the set of timeslots that contains all target timeslots. LSP is a polynomial time algorithm that can be used on the $t\_can\_assign/t\_can\_revoke$ or the $t\_can\_enable/t\_can\_disable$ rulesets, it takes a target role l and returns the set of rules which is associated with the longest shortest path. The LSP functions works backwards from the state where a the role is assigned, or enabled (depends on the ruleset), starts with the set of rules with target role equal to l, and uses the preconditions to build a set of rule paths. Since we are dealing with rule sets, the set size is linearly limited by the number of rules, and the number of rule sets is limited linearly by the number of rules.

We have provided a full example of calculating each tightening in Section C.2.

3.5.5 Reduction to Model Checking

A correct reduction from ATRBAC-Safety to Model Checking must ensure: (1) that every state an ATRBAC policy can reach, is also reachable by the reduced model, and (2) every state that our reduced model can reach must be reachable by the ATRBAC policy. We chose to reduce to model checking because of the similarities between ATRBAC-Safety and

model checking, and to leverage the mature model checking community for their robust and highly optimized software.

Model checkers have 2 variable types: state variables and control variables. The state variables have controlled values, where state transitions dictate future values. Control variables are left to the model checker to choose a value for. This allows the model checker to perform optimizations to dictate which state transitions to pick when a choice is required.

Our reduced model has 2 state variables: (1) a $|users| \times |roles| \times |timeslots|$ binary array TUA that indicates if a user is assigned to a role in a specific timeslot, and (2) a $|roles| \times |timeslots|$ binary array RS that indicates if a role is enabled for a specific timeslot. The value for $|roles|$ is extracted from the list of all roles in the ATRBAC policy, $|timeslots|$ is extracted using the polynomial reduction algorithm to convert time intervals into non-overlapping timeslots (see Reduction 2 in [115]). We use tightening 1 to calculate the value for $|users|$. Both TUA and RS are initialized to false (ATRBAC-Safety initial conditions) and each variable has a state transition where the default case keeps the current state. TUA and RS accurately represent the underlining TRBAC state in an ATRBAC policy.

We have 3 control variables that we let the model checker control:

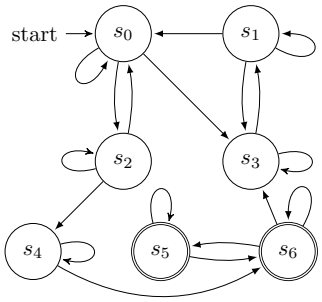
rule an enumerated integer that can take the value of any rule in the ATRBAC policy (ex: $rule : \{\text{CA01, CA02, } \ldots\text{, CR01, } \ldots\text{, CE01, } \ldots\text{, CD01, } \ldots\};$),

user an integer between 1 and the maximum number of users (|Admin-Roles| + 1, see Section 3.5.4), to represent a selected target user, and

admin which is modelled the same as *user* but is independently assigned a value, this represents a selected admin user.

For each $t\_\ can\_\ assign$/$t\_\ can\_\ revoke$ rule, we add a state transition to each affected TUA variable if the admin and target user satisfy the rule's condition in the current state. The model checker is able to change the state of TUA by assigning values to *rule*, *user*, and *admin* such that:

1. the *rule* is a $t\_\ can\_\ assign$ or $t\_\ can\_\ revoke$ rule,
2. the *user*'s values in TUA, in the current state, satisfy the *rule*'s preconditions C_t,
3. the *admin*'s values in TUA, in the current state, satisfy the *rule*'s administration condition, and
4. the *rule*'s administration condition is enabled for at least 1 timeslot in RS, in the current state.



(a) Diagram Model

```
1         // Query: t2, [r3, r4]
2         LTLSPEC G !(u[1][3][2] = TRUE & u[1][1][2] = TRUE)
3         // User 1, Role 3 Assign/Revoke
4         next(u[1][3][1]) := FALSE;
5         next(u[1][3][2]) := case
6         user=1 & rule=CA1 & ( (u[admin][1][1]&e[1][1]) |(u[
              admin][1][2]&e[1][2]) |(u[admin][1][3]&e[1][3]) ) &
                (u[user][2][2]&u[user][2][3]) & !(u[user][1][2]|u
              [user][1][3]) : TRUE;
7         TRUE : u[1][3][2];
8         esac;
9         next(u[1][3][3]) := case
10        user=1 & rule=CA1 & ( (u[admin][1][1]&e[1][1]) |(u[
              admin][1][2]&e[1][2]) |(u[admin][1][3]&e[1][3]) ) &
                (u[user][2][2]&u[user][2][3]) & !(u[user][1][2]|u
              [user][1][3]) : TRUE;
11        TRUE : u[1][3][3];
12        esac;
13        // Role 1 Enable/Disable
14        next(e[1][1]) := case
15        rule=CE2 & (e[4][1]) & !(e[2][1]) : TRUE;
16        TRUE : e[1][1];
17        esac;
18        next(e[1][2]) := FALSE;
19        next(e[1][3]) := FALSE;
```

(b) SMV Code

Figure 3.5: Figure (a) depicts the diagram view of model checking, where s_0 is the initial ATRBAC state and each state transition is effecting a rule r with an admin and target user that satisfy the rule's admin and target role conditions. States s_5 and s_6 are the accepting states, where a user in TUA satisfies the safety query. Figure (b) shows a version of (a) using a small snippet from the reduction of Figure 3.2 to NuSMV's model checking language. The TRBAC state is encoded as such: TUA is defined by u and RS is defined by e in out reduction above.

For each *t_ can_ enable*/*t_ can_ disable* rule, we add a state transition to each affected RS variable if the admin and current state satisfy the rule's condition. The model checker is able to change the state of RS by assigning values to *rule*, *user*, and *admin* such that:

1. the *rule* is a *t_ can_ enable* or *t_ can_ disable* rule,
2. the current state in RS, satisfy the *rule*'s precondition,
3. the *admin*'s values in TUA, in the current state, satisfy the *rule*'s administration condition, and
4. the *rule*'s administration condition is enabled for at least 1 timeslot in RS, in the current state.

We can intuit the correctness of the reduction by comparing state transitions to rule firings. In an ATRBAC policy, to fire a rule r the target and admin users must both satisfy their conditions, and the admin role must be enabled for the current time of day. If these conditions are true then the TRBAC instance is updated for the TUA

($t\_\ can\_\ assign/t\_\ can\_\ revoke$) or the RS variable ($t\_\ can\_\ enable/t\_\ can\_\ disable$). The reduced model checks all conditions except for the check if the admin role is enabled during the current time. In Section 2.2.2, we define timeslots to be periodic, in our reduced model we utilize this to optimize the model by ignoring time. We are able to achieve this because we can stay in the current state until the current time reaches one of the timeslots required to enable the admin role. Without this optimization, we would require a time state variable to keep track of the current timeslot and state transitions to increment, and wrap, to the next timeslot.

In Figure 3.5, we show a small snippet of the reduction from Figure 3.2 to model checking. The state variables user/admin/rule are defined as above. To save space, TUA and RS were renamed to u and e respectively. We can understand u and e by their index values: $u[user][role][timeslot]$, $e[role][timeslot]$. Figure 3.5 does not show the variable declaration or initialization, and hides most of the rules for user1 and all rules for user2. The rules to dictate rule enablement are truncated.

The query on line 1 can be understood as: "Does the statement after G hold true for all future time instances?". We translate the query to model checking by negating the query "can any user ever obtain the goal roles for the goal timeslot?". An optimization to the query is that we force user1 to be our target user, by limiting our query to "can $u[1]$ ever obtain the goal roles for the goal timeslot?".

In Figure 3.5 each state variable must have a state transition statement (`next` or `case`). Without these statements the model checker would be able to change the value between states, thus obtaining states that are unreachable to the ATRBAC policy. For role$\times$timeslots that are not touched by a $t\_\ can\_\ assign$ rule, we set the next value in u to **FALSE**, for all users, as the value cannot change from the initial value $FALSE$. We do the same with e and $t\_\ can\_\ enable$ rules.

The format for state transitions for the TUA variable u are: (correct user) $\wedge$ (correct rule) $\wedge$ (admin satisfies condition for any timeslot in interval) $\wedge$ (user satisfies condition for all timeslots) : TRUE. The format for state transitions for the RS variable e are: (correct rule) $\wedge$ (admin satisfies condition for any timeslot in interval) $\wedge$ (e satisfies condition for all timeslots) : TRUE. Admin and target condition which are TRUE are satisfied in all states. The last line of each case statement is the default route, and this prevents state changes outside of the rules in the ATRBAC policy.

3.5.6 Performance Considerations

The performance techniques described above are able to give a performance increase in the best case, and in the worst case add polynomially timed overhead. In this section we will consider each of the performance techniques and outline conditions where a performance increase or decrease is expected.

The Polynomial Time Solving module of Cree, is optimized to run very quickly because they are executed before each technique. We see a performance increase if the ATRBAC policy is relatively simple or when running with abstraction refinement on. We have found that the initial abstraction refinement steps are able to be solved almost exclusively using Polynomial Time Solving. We also found that we do not see any noticeable decrease in performance if this technique is not able to solve the policy.

Forward pruning is effective when the ATRBAC policy contains rules that cannot fire. There are a few different ways that this can be the case for a rule. Forward pruning has been shown to greatly reduce the policies created by Uzun et al. [133]. We see an increase in performance for backwards pruning if there exists low coupling between the set of goal roles and the rules which reference them. Backwards pruning decreases performance if there is high coupling within the policy, as this results in very few roles/rules/timeslots from being removed.

A performance increase from abstraction refinement is only possible for UNSAFE ATRBAC-Safety policies. This due to the one sided error that abstraction refinement implements. If the policy is assumed SAFE, then this feature should be turned off to increase performance. Parallelizing abstraction refinement would mitigate the negative effects on performance for SAFE policies, but this feature was not implemented in Cree due to time constraints.

The performance considerations for bound estimation rely on the optimizations of the ATRBAC-Safety solver being used. Our bound estimation produces an upper bound which can be exponential in the size of the input. The performance effects of bound estimation relies on the ingenuity of the solver. Empirically, we have noticed that NuSMV [78], when running in bounded model checking, is much faster than its symbolic model checking variant only when given our upper bound. If only using Tightening 1, we have found that symbolic model checking was quicker on average.

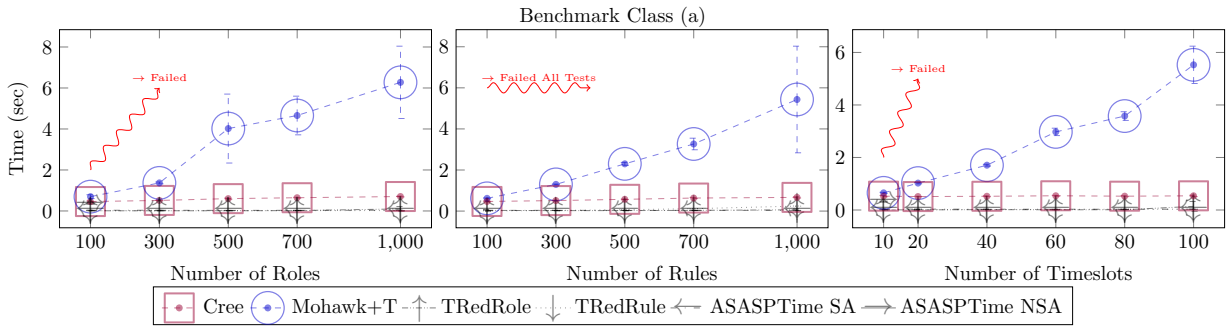


Figure 3.6: Results from all tools for Benchmark Class (a). It comprises random input instances from a generator from Uzun et al. [133]. The curves interpolate averages, and the error-bars show the standard deviation. Figure D.1 contains no Mohawk+T.

3.6 Empirical Assessment

We have implemented Cree in Java, and made it available for public download [103]. In this section, we discuss an empirical assessment we have conducted of Cree, in comparison to five prior tools for ATRBAC-Safety to which we have access. Our intent with such an empirical assessment is to ask whether Cree's design and implementation does indeed result in a performant tool when compared to prior tools.

We have conducted empirical assessments on the three benchmark classes from prior work: Uzun et al. [133], Ranise et al. [93], and Jayaraman et al. [56]. Our results are shown in Figure 3.6, Figure 3.7, and Figure 3.8. Results without Mohawk+T are shown in Figure D.1, Figure D.2, and Figure D.3. The curves interpolate the average of 5 runs. The error-bars show the standard deviation from the average. The red wavy lines represent the point where a tool is unable to solve the rest policies due to timing out or crashing.

Benchmark Class (a) in Figure 3.6, first presented by Uzun et al. [133], but altered here, are randomized test-cases where:

- Roles subplot: Rules are fixed at 200 and Timeslots at 20.
- Rules subplot: Roles are fixed at 200 and Timeslots at 20.
- Timeslots subplot: Rules and Roles are fixed at 200.

Everything about the rules created in the Benchmark Class (a) rules are randomized:

- Random start and end times for the administrator time-interval, where start $\leq$ end time.
- For every role that exists there is a 1/5 chance that it will be added to the rule's precondition as a positive role condition, and 1/5 chance for a negative precondition.

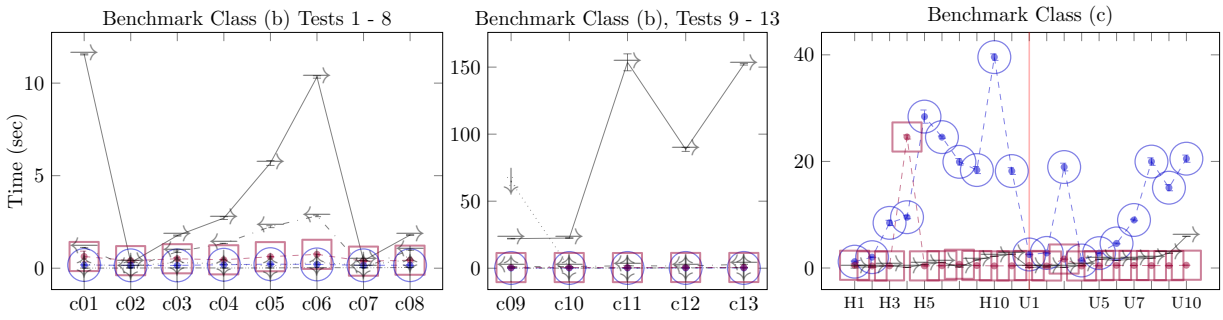


Figure 3.7: Results from Benchmark Class (b) (two graphs to the left), and Benchmark Class (c) (right). These comprise input instances from the work of Ranise et al. [93]. Figure D.2 contains no Mohawk+T.

These factors differ from the original code in [133] where there was an equal probability of 1/3 for each case. The change is to reduce the number of role preconditions is to allow for more rules that have zero preconditions, and thus are allowed to be executed. Without rules with empty preconditions the query will always be unreachable and thus a safe system.

- The target role is randomized.
- The target time-interval is a set of time-slots and there is a 1/2 probability of a time-slot being added to the "role-schedule".
- The type of rule is randomized with a 1/2 probability for $t\_\ can\_\ assign$ or $t\_\ can\_\ revoke$.
- The administrator is "TRUE".

Benchmark Class (b) in Figure 3.7, presented by Ranise et al. [93], are ARBAC policies that have "temporality" randomly added to them. We would like to thank Ranise et al. for providing these test-cases for us to use. This set of 13 policies all have "TRUE" as the administrator and only contain $t\_\ can\_\ assign$ and $t\_\ can\_\ revoke$ rules.

Benchmark Class (c) in Figure 3.7, presented by Ranise et al. [93], is generated similarly to Benchmark Class (b) but these rules allow for arbitrary administrator roles.

In Figure 3.8, we present test cases taken from [56], which are ARBAC policies and we convert them to ATRBAC policies by introducing 1 time-slot and having every rule associate with that time slot. We first converted this set in our prior work [115]. Given an example ARBAC rule: $\langle a, C, t\rangle$, we can convert it with a single time-slot ts such that the policy still reflects it's original guarantees on safety: $\langle a, ts, C, ts, t\rangle$. The Mohawk test-cases are split into 3 complexity classes: polynomial time, NP-Complete, and PSPACE-Complete. This reflects what is contained within the test-cases:

- Polynomial Time: *can_ assign* and *can_ revoke* rules where the administrator is "TRUE", only positive preconditions for *can_ assign* rules or "TRUE", and *can_ revoke* rule's preconditions are "TRUE".
- NP-Complete: *can_ assign* rules where the administrator is "TRUE" and preconditions can include positive or negative roles, or be "TRUE".
- PSPACE-Complete: *can_ assign* and *can_ revoke* rules where the administrator is "TRUE" and preconditions can include positive or negative roles, or be "TRUE".

For Benchmark Class (a), Figure 3.6, Cree is within 0.5 seconds of the tools TredRole, TredRule, and AsaspTime-Sa and outperforms our prior tool Mohawk+T by almost 6 seconds for the bigger policies. AsaspTime-Nsa was unable to run most these tests. Cree's Polynomial Time Solving when possible is the factor that reduces the run time down to comparable times to the other software as we are able to skip the expensive overhead of running the model checker.

For Benchmark Class (b), Figure 3.7, all tools, except AsaspTime-Nsa and Asasp Time-Sa, solve the policies within 1 second. This is the only instance where Mohawk+T outperforms Cree, but the amount of the order of 10 milliseconds.

For Benchmark Class (c), Figure Figure 3.7, Cree either performs the quickest or is tied with the other fastest tool AsaspTime-Nsa. The only exception to this is the policy Hospital 4, where Cree performs the worst overall. Note that as in prior work [93], we did not try this Benchmark Class on the TredRole and TredRule tools.

For the Mohawk inputs, Figure 3.8, Cree significantly outperforms all the existing tools. Furthermore, besides Mohawk+T, the existing tools are unable to withstand the input instances from Mohawk [56] beyond a certain threshold. For the polynomial-time verifiable sub-class, for example, which is Test Suite 1, none of the existing tools were able to handle inputs beyond 20,000 roles and 80,000 rules.

Cree is no worse than any of the prior tools for any input we tried. Furthermore, for each of the prior tools, there exists an input for which Cree is strictly better. Mohawk+T is strictly worse than Cree for Benchmark (a), Benchmark(c), and the Mohawk inputs. Asasp Time-Sa is strictly worse than Cree for input Benchmark (b). For the tests 9-13, Asasp Time-Sa performs 0.5 to 3 seconds slower than Cree. AsaspTime-Nsa is strictly worse than Cree for the Mohawk inputs. TredRole and TredRule perform much faster than Cree in almost all instances it was able to handle, but Cree is able to solve much larger policies without crashing or timing out.

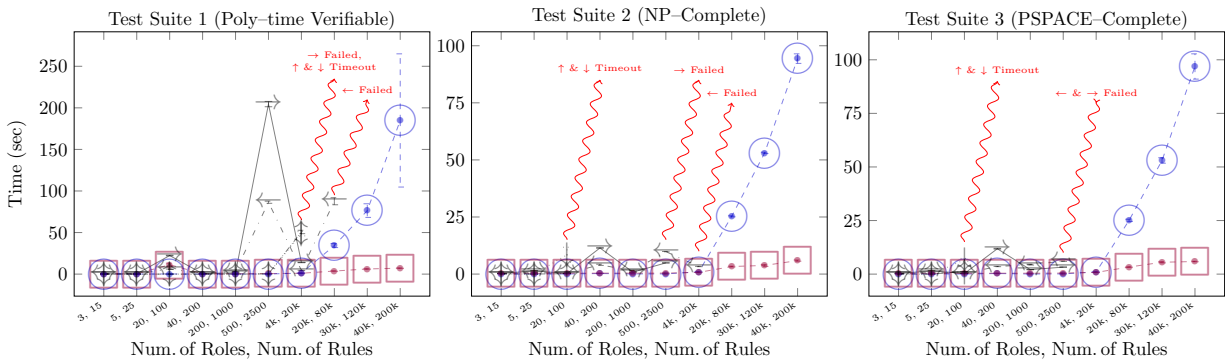


Figure 3.8: Mohawk inputs [56] converted to ATRBAC-Safety instances using one time-slot. Test Suite 1 are inputs with non-negated preconditions only. Test Suite 2 are inputs with no revoke rules. Test Suite 3 are both positive and negated preconditions, and assign/revoke rules. Red wavy lines are tools that crashed/timed-out during testing for certain input sizes. Figure D.3 contains no Mohawk+T.

3.7 Conclusions

We have proposed a new approach and corresponding tool which we call Cree, for addressing safety analysis in the context of Administrative Temporal Role-Based Access Control (ATRBAC). ATRBAC introduces new features, and therefore technical challenges for safety analysis: support for time-intervals in policies, and rules for enabling and disabling roles. In Cree, we reduce the problem to model checking and then leverage an existing model checker, NuSMV. In addition, we incorporate several heuristics for performance: Polynomial Time Solving when possible, Forward and Backward Pruning, Abstraction Refinement, and Bound Estimation. While these heuristics are inspired by prior work, our algorithms are customized for the specific technical challenges that ATRBAC-Safety introduces. We have conducted a thorough empirical assessment in which we compare Cree to five prior tools. Cree is no worse than any of the other tools across all inputs we tried, and outperforms every other tool for at least one of the input cases. In addition to comparable completion time, Cree is also able to solve many more test cases without timing out or failing than all prior tools.

Acknowledgements

We thank the creators of the prior tools [93, 133] for making their tools and benchmarks available to us and helping us with their use. This work was partially supported by the National Science Foundation (NSF) grants NSF CNS-0964710, NSF CNS-1748109, and NSF Award #1736209; awarded to Jianwei Niu.

Chapter 4

Generating Hard Instances of ATRBAC-Safety

Contents

Declaration of Contributions

I am the sole author of this chapter.

4.1 Introduction

Access control is vital for the security of computers connected to a network. Access control policies define the privileges each user has to each of the resources controlled by the system. ATRBAC is an access control model that is typically used in large organizations that have a large user base and who often deal with offices spanning multiple timezones.

Safety analysis of Access Control policies, is a decision problem which verifies if every reachable state, from the initial state, satisfies a Safety Query. The safety query in ATRBAC-Safety ask whether any user can obtain a set of roles, for a specific time, from the initial state (no users have roles assigned to them and no roles are enabled). This type of query is often used to ensure that users, who have a certain access level, are unable to obtain sensitive/compromising privileges within an ATRBAC-policy.

Schools, Hospitals, and Governments can utilize ATRBAC to handle their access control needs, and can use ATRBAC-Safety to ensure that manual and automatic policy generation conforms to business security logic. An example safety query within a school environment: can a student in class A also obtain the privileges to update marks for class A. An example for a hospital setting: does the current ATRBAC policy conform to the HIPPA and PHIPA laws. This last example would require potentially many safety queries to test.

Prior work in Access Control, ARBAC, ATRBAC, and Safety Analysis is located in Section 3.3. Shahen et al. [115] show that solving the above safety analysis queries is a **PSPACE**-complete problem. In Chapter 3, we created an ATRBAC-Safety solver, and tested the effectiveness against datasets produced by: Uzun et al. [133], Ranise et al. [93], and modified policies from Jayaraman et al. [56].

The contributions of this chapter are: an analysis of ATRBAC-Safety, identifying some "sources of complexity", an analysis of previous datasets, and the creation of 2 new datasets that demonstrate increased solver difficulty.

The rest of this chapter is organized as follows. In Section 4.2, we provide background on ATRBAC-Safety. Then in Section 4.3, we outline some of aspects of ATRBAC-Safety policies which cause the instances to be difficult to solve, called "sources of complexity". In Section 4.4, we analyze the datasets created by [133, 93, 115] and show which sources of complexity are included. In Section 4.5, we analyze 4 general ATRBAC-Safety performance techniques that were seen in previous empirical solutions to ATRBAC-Safety. We then propose 2 new dataset generation techniques, in Section 4.6. In Section 4.7 we provide empirical analysis of the 2 new datasets and test the performance of Cree using multiple operational modes. The chapter ends with Future Work and Conclusions.

4.2 ATRBAC-Safety Background

The ATRBAC-Safety background has been moved to Section 2.2.

4.3 ATRBAC-Safety Sources of Complexity

ATRBAC-Safety was shown to be **PSPACE**-complete in [115]. In this section we will attempt to enumerate some of the aspects within an instance of ATRBAC-Safety that make the problem difficult to solve. We will call these aspects the "sources of complexity", as was done by Jayaraman et al. [56] and Stoller et al. [126] with ARBAC-Safety. In this section we will describe sources of complexity and provide intuition as to why it makes ATRBAC-Safety more difficult. In Section 4.5 we describe general techniques that address some of these sources of complexity to show empirical improvements with the datasets provided. As well, in Section 4.6 we use these sources of complexity to create "hard" ATRBAC-Safety policies.

Sources of complexity for ATRBAC-Safety can be defined in terms of a state-search algorithm, similar to model checking. Sources of complexity are attributes of an ATRBAC-Safety policy that require more space in the state or more computation steps required to answer the safety query. Obvious sources of complexity are adding more roles, timeslots, and rules to a policy. Adding role/timeslots increase the size of the TRBAC state that is required to be kept track of. Adding more rules can increase the number of states that are reachable. If we fix the number of roles/timeslots/rules of an ATRBAC-Safety policy, we can then identify less obvious sources of complexity.

4.3.1 Solver Sources of Complexity

ATRBAC-Safety has many versions published in the literature. The ATRBAC-Safety version used by Cree, in Chapter 3, is a generalization of the previously published versions. Here we discuss the sources of complexity that come from implementation decisions made by a solver. Performance techniques used by Cree are discussed in Section 4.5.

Non-Separate Administration The separate administration restriction, in the context of safety analysis, excludes administrator roles from appearing in the precondition and target conditions. A separate administration policy is one where the set of roles that appear in the administrator condition does not overlap with the set of roles that appear in

the precondition and target roles. Jayaraman et al. [56] used the separate administration restriction for their ARBAC-Safety tool Mohawk. Versions of ATRBAC-Safety were created with the separate administration restriction in the works [133, 93]. We will focus on ATRBAC-Safety where a Non-Separate Administration is assumed, as it is a more general problem. The work of Ranise et al. [93] and Shahen et al. [115, 116] use non-separate administration in their works.

The non-separate administration restriction provides a source of complexity because the administration condition now must be considered. In a separate administration, all rules are satisfied if at least 1 user is within the admin role, so we can rewrite all of our rules with the administration role equal to TRUE. In non-separate administration we now are required to create a user that is able to satisfy the administration condition before firing a rule. This adds computation steps to safety analysis because we also need to find paths for potential administration roles before we can fire a rule on our target user. This can add space required in the TRBAC state, since the maximum required users is the number of admin roles plus one user to satisfy the safety query.

Time Progression In the ATRBAC-Safety described in [93, 115], time is an integer m which increases until it reaches a maximum value and then resets to 0 and repeats. The current time m is used to validate the administration time interval, as the m must be in the time interval in order for the rule to fire. Tools can greatly reduce the number of computation steps required if they do not encode the current time into the TRBAC state, but instead assume that given enough time m, will eventually be the correct value required to fire a rule.

TRBAC Encoding The method of encoding a solver uses for the TRBAC state can determine additional sources of complexity. We outlined a binary matrix method in Section 3.5.5, which is the method used by Cree. The size of the TRBAC state remains consistent throughout executions of rules. Another encoding method would be a list of sets for each user and a set for role enablement. Where a user U is enrolled in a the role r for the timeslot t if tuple $\langle r, t\rangle \in U$. A more space compact versions is where the tuple is $\langle r, [t_1, t_3, \ldots]\rangle$. Both versions have the minimum size at the initial size and increase in size when a CanAssign or CanEnable rule is invoked, decrease when a CanRevoke or CanDisable rule is invoked. The choice of encoding scheme can change the space/computational requirements an instance requires to solve.

4.3.2 Dataset Sources of Complexity

Jayaraman et al. [56] listed the following attributes as sources of complexity for ARBAC:

Disjunctions: multiple CanAssign rules with the same target role,
Irrevocable Roles: a role that has a CanAssign rule but no CanRevoke rule,
Positive Precondition Roles: roles that appear without negation in CanAssign rules.
Mixed Roles: roles that appear with and without negation in CanAssign rules,

We will first adapt the above observation to ATRBAC-Safety and then we will make additional observations for new sources of complexity.

Disjunctions An ATRBAC-Safety policy that contains disjunctions means it contains multiple CanAssign rules with the same target rule. This adds a source of complexity because we have now multiple paths to obtain a required role. Disjunctions form a problem in NP, the correct path to a target role can be verified using polynomial time, which path to choose is non-obvious and requires a polynomial set of non-deterministic decisions to find. Disjunctions can also exists in the role-enabled table, if there exists multiple CanEnable rules with the same target role. ATRBAC includes a precondition for all rule types, thus disjunctions now exists for the Can Revoke and Can Disable rules if the target role appears as negated in a precondition. Timeslots alter the "disjunctions" source of complexity in the following way: a disjunction exists in an ATRBAC-Safety policy if there exists multiple rules where they share the same target role and at least 1 target timeslot.

Irrevocable/Un-Disableable Roles A role is irrevocable if it exists as a target role in a CanAssign rule with a target time-slot array $T = [t_1, t_2, \ldots, t_x]$ and does not have corresponding CanRevoke rules for the set of timeslots in T. If there exists disjunctions then T is the union of all CanAssign rules with the same target role. Irrevocable roles also exists in the Enabled domain, where the above is altered to CanEnable and CanDisable rules. For algorithms that are using a "forward-search" technique, it is non-obvious when a user should obtain an irrevocable role and non-obvious when a un-disableable role should be enabled, as a further rule condition might require the user to not have the role for a particular timeslot or for a role to be disable for a particular timeslot. For an irrevocable or un-disableable role to cause difficulty when solving, the role must also appear as a positive and negated role in a precondition of CanAssign/CanRevoke (irrevocable) and CanEnable/CanDisable (un-disableable).

Positive Precondition Roles A positive precondition role is a role that appears in a rule's precondition without negation. We can see that the more positive precondition roles there are the more computational steps are required to satisfy all possible conditions.

Mixed Roles A mixed role in ATRBAC-Safety is a role that appears in the precondition of CanAssign/CanRevoke rules or CanEnable/CanDisable rules with negation and without for at least 1 shared timeslot. The source of complexity that mixed roles add is that a user

my need to enroll in a role for a particular timeslot to obtain another role and then be required to unenroll in order to satisfy another condition, same for role enablement. This adds more computation steps to the safety analysis.

Number of Administration Roles The number of roles in the set administrator roles is a source of complexity as stated above in Non-Separate Administration. For each administration role we can be required to create a unique user to obtain that role. This new user increases the size of the TRBAC state. In separate administration we only required 2 users, an administrator and a target user. In non-separate administration we require the number of administration roles to act as potential users plus 1 user to be the target user attempting to reach the goal state.

Minimum Number of Steps to Goal State For ATRBAC-Safety policies where the Safety Query is Reachable, the limit on the minimum number of computation steps required is dictated by how far the goal state is from the initial state. If an ATRBAC-Safety policy has 1000 roles, rules, and timeslots, but the goal state is reachable by applying 1 rule to a user, then the number of computation steps required is constant in the number of rules.

Rules Types and Properties We list a few of the complexities derived from rule type and properties below. Given no Can Enable or Can Disable rules, we can remove the role enablement table from the TRBAC state. All rules which have admin roles but no Can Enable rules are impossible from running since no user can satisfy the admin condition from the initial conditions. This can be applied to Can Assign/Can Revoke rule, for roles which appear as a positive precondition, if there exists no Can Assign rule for that role for the required timeslot, then that rule can never fire. Similar for Can Enable and Can Disable rules. We explore rule types and properties in more detail in Section 4.4.2.

4.4 Analysis of ATRBAC-Safety Datasets

There are 3 main ATRBAC-Safety datasets from the literature. We refer to them by there first authors surname: Uzun, Ranise, and Jayaraman. Table 4.1 provides an overview of the notable features from each dataset. All policies used in our analysis can be found here [104].

Uzun et al. [133] presented the first dataset in the literature. They created a program to randomly create an ATRBAC-Safety policy given 3 numbers as input: number of rules, roles, and timeslots. In order to have an accurate comparison, Shahen et al. [116] created a repository where they randomly generated 10 policies for each of the role/rule/timeslots

Table 4.1: Summary of Previous ATRBAC-Safety Datasets.

	Uzun	**Ranise**	**Jayaraman**
Number of Policies:	150	33	30
Number of Reachable/Unreachable:	0/150	25/8	30/0
Longest Pre-Condition:	711	5	4
Longest Timeslot Array:	549	1	1
Longest Reachable Path:	0	5	4

combinations that were presented in the original paper. We used those policies for our analysis. Table 4.1 shows that the random nature of generation produces reachable safety queries in very unlikely cases. This is due, in part, to the very long preconditions and time slot arrays.

Ranise et al. [93] created 3 datasets, which we have combined into a single dataset for analysis. The first dataset was created with the assumption that there exists a separate administrator role, and thus the admin condition for every rule in the set is TRUE. The second and third datasets assume non-separate administrator roles. The second and third dataset are created from hospital and university access control policies, respectively. These datasets produce the longest reachable path, and contain both reachable and unreachable safety policies.

Jayaraman et al.[56] created an 3 ARBAC-Safety datasets, where each dataset introduces additional features to increase the complexity required to solve. Shahen et al. [115] updated the ARBAC-Safety policies to ATRBAC-Safety, using the version of ATRBAC-Safety presented in [115]. The conversion introduced a single timeslot which all admin intervals and target time slot arrays are set to.

These datasets have been used to show relative performance of many ATRBAC-Safety solvers. We are analyzing the datasets to see what sources of complexity exists within the combined datasets. This analysis will show where these datasets lack in terms of "sources of complexity", and why Cree was able to solve all tests cases in under 25 seconds when the complexity class of the problem is **PSPACE**-complete.

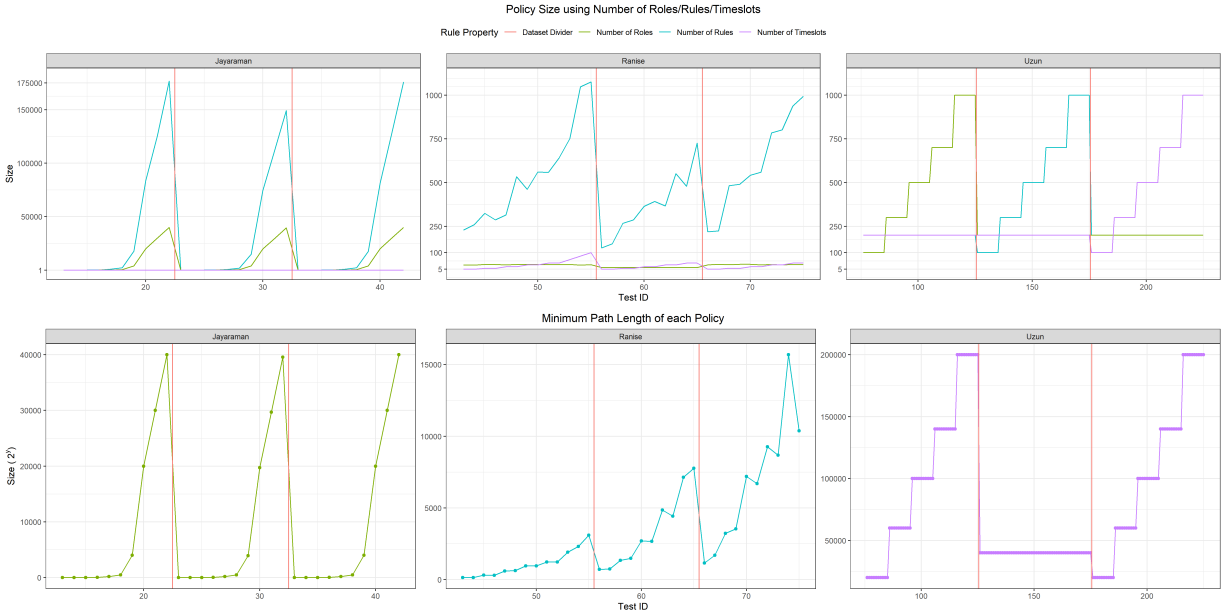


Figure 4.1: Comparison of two methods for measuring policy size. Top figure defines policy size using the number of Roles/Rules/Timeslots. Bottom figure show the new proposal for measuring the size of an ATRBAC-Safety policy, by determining the minimum simple path required to visit all possibly required roles.

4.4.1 Measuring Size of a Policy

In prior work [133, 93, 115, 116], ATRBAC-Safety policies were described in terms of 3 numbers: the number of unique roles, rules, and timeslots that appear within the policy. This definition can be used to compare policies where only 1 of these values differ, but when 2 or all 3 of the numbers are different, the expected hardness for solving policies of those sizes becomes much harder to determine. Figure 4.1 shows the policy size of all prior dataset using this definition for policy size.

We can compute the values in Figure 4.1, top figure, using a linearly timed algorithm. Using the policy size can be used to predict how difficult a policy would be to perform safety analysis. The Jayaraman dataset has the greatest number of roles and rules, but there is is only a single timeslot. We see that Uzun and Ranise have comparable number of rules, but that Uzun has a larger number of roles and timeslots. This definition of policy size provides predictions on the size of the TRBAC state required to perform safety analysis. This definition relies upon three values which makes comparisons between 2 policies more

Figure 4.2: The percentage of rules by type.

difficult.

In Figure 4.1, bottom figure, we create a single value representation of the size of the policy. This value can be computed using a polynomially timed algorithm. The size is calculated using the formula in Section 4.5.4. This definition calculates the maximum length of a minimum simple reachable path for the safety query and the rules in the policy. This allows for comparison of computational steps required and incorporates the safety query into the measurement. From the graph we see that the datasets have switched positions. Uzun has the largest size, then Jayaraman, and last Ranise. While the number is useful to compare abstract policies of a fixed size, the number of operations to solve each test case varies. In the case of the Uzun dataset, due to how the policies are randomly created, it is near impossible for a reachable policy to be created (as can be seen in Table 4.1).

4.4.2 Rule Types and Properties

There are 4 rule types defined in an ATRBAC-Safety policy: CanAssign, CanRevoke, CanEnable, and CanDisable. All rules are a tuple of the following: *Admin Role, Admin Time-Interval, Pre-Condition, Target Timeslot Array, Target Role*. The CanAssign/CanRevoke rules effect changes in the *TUA* state, and CanEnable/CanDisable rules effect changes in the *RS* state.

Satisfying the safety query requires any user to obtain a set of roles for a specified timeslot. A minimum requirement requires that there exists CanAssign rules with a target role and the timeslot in timeslot array for all roles in the safety query. From the set of CanAssign rules above, we can extract conditions which these rules require to fire. We perform this analysis during backwards pruning in Cree (see Section 3.5.2. A summary of additional conditions:

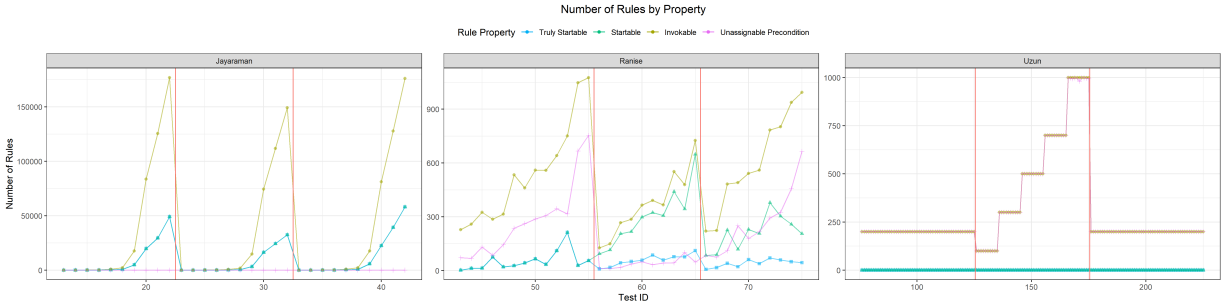


Figure 4.3: Number rules by Rule Property: Truly Startable, Startable, Invokable, and Unassignable Precondition. If no truly startable rules exists, and we assume a empty initial condition, then *TUA* and *RS* are unable to change.

- Positive roles in the precondition require extra CanAssign Rules,
- Roles that appear as positive and negative in the precondition require CanRevoke rules,
- Administrator roles that are not TRUE require CanAssign and CanEnable rules, and
- CanDisable rules are required if any role appears as positive and negative in a CanEnable rule.

In Figure 4.2, we show the percentage of each rule type for each dataset. Jayaraman and Uzun assumed separate administrator, thus no CanEnable/CanDisable rules. Ranise contains CanEnable/CanDisable rules for 2 of the test suites. The middle test suite for Jayaraman contains only CanAssign rules and the other test suites contain very few Can Revoke rules. Uzun datasets randomly select the rule type, thus the number of rules per type is nearly equal. Ranise has the most evenly spread of rule types.

Rules can be classified in many ways, one way of classification is looking at the conditions that might restrict a rule from firing. Within the safety analysis, we assume that if an administrator exists then they will give a user a role if they ask for it. Thus there exists 2 conditions that restrict a target user from obtaining a role in a given state:

1. The target user does not currently satisfy the Pre-Condition and Target Timeslot Array, and
2. No user is able to satisfy the Admin Role and Admin Time-Interval.

The number of rules that are able to fire for a given state, is a source of complexity as it determines the number of states that are reachable.

In Figure 4.3, we show the number of rules that with the following properties:

Truly Startable A rule that can be fired during the initial state.
Startable A rule that can be fired only if an admin is available and enabled.

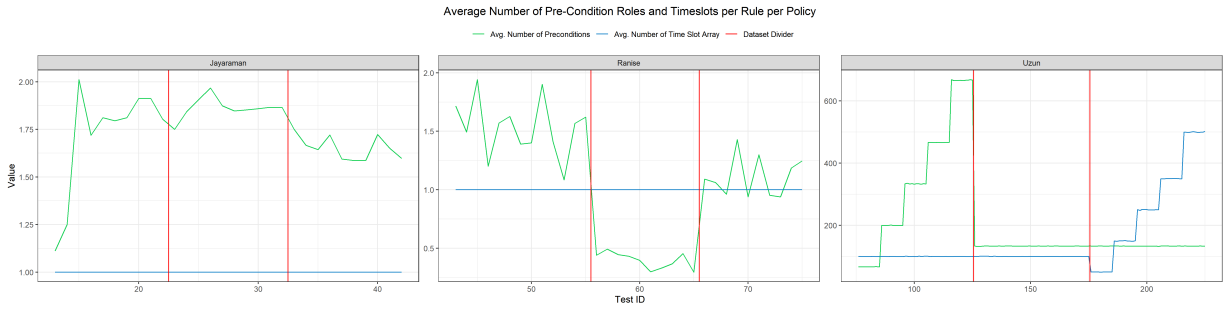


Figure 4.4: Average number of roles in the precondition and average number of time slots in the target time slot array. Maximum number in Table 4.1.

Invokable A rule that has the possibility of firing some time in the future.

Unassignable Precondition A rule where it is impossible for any user to satisfy the precondition.

The set of truly startable rules dictate the number of paths that exists from the initial state, and are able to fire in almost every state afterwards. The set of startable rules is able to fire once a single user obtains the necessary administrator role and that role is enabled. A truly startable rule is also a startable rule. The set of Invokable rules is able to fire only when the admin condition is satisfied by an admin user, and the precondition is satisfied by a target user. It is hard to determine the set of rules required to satisfied invokable rules. It is harder to determine the sequence of steps necessary to fire an invokable rule.

In Figure 4.3, we see that the Uzun dataset contains only invokable and unassignable preconditions rules. Thus safety analysis is very easy for these policies, since there are no branches from the initial state. Jayaraman contains no unassignable precondition rules. The number of truly startable rules is equal to the number of startable rules for Jayaraman. Ranise has a significant number of unassignable precondition rules. From this analysis, Jayaraman has the most rules that can be fired during the starting state and most rules which can add available branches for each state.

4.4.3 Precondition and Timeslot Array Lengths

The number of roles in a rule's precondition and the number of timeslots in a rule's timeslot array length is a source of complexity. Specifically, the more roles in a precondition, the more rules that are required to satisfy the precondition. This is similar for the timeslot array, as a user must have all positive precondition roles for all timeslots in the timeslot

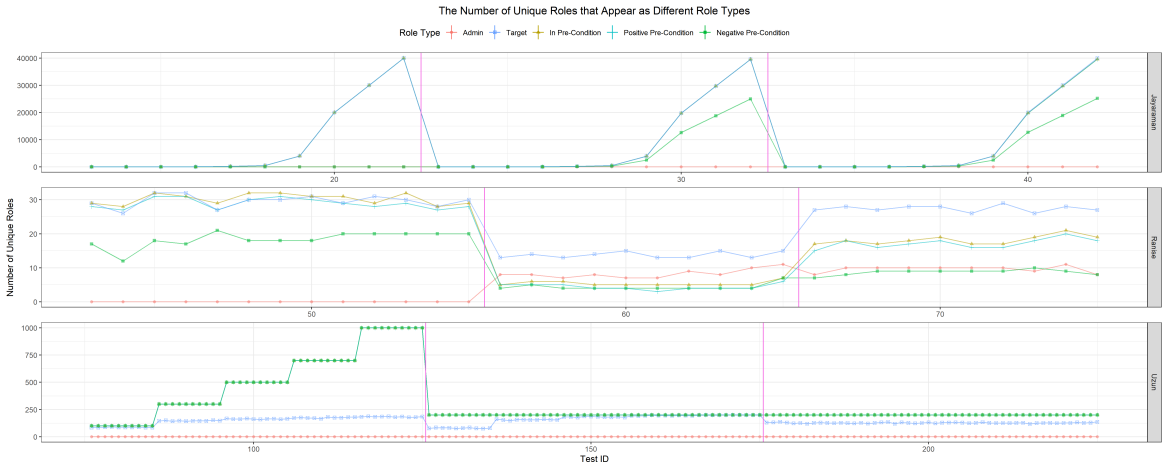


Figure 4.5: Number of roles that appear in unique roles within each policy for each of the prior datasets. The number is calculated by iterating through all rules and adding each role to a set for each role property, the number is the size of the set. Uzun and Jayaraman both assume a separate administrator scheme, and thus have no administrator roles. The set of positive, negative, and in precondition are all the same size for the Uzun dataset, this is because all roles have a probability of 2/3 of being in each rule.

array and not have any of the negative roles for any of the timeslots in the array. Thus longer precondition and timeslot arrays increase the work required for rules to be fired, thus adding to the source of complexity.

In Figure 4.4, we plot the average length of the precondition and timeslot arrays. Table 4.1 shows the maximum length precondition and timeslot array. The datasets for Jayaraman and Ranise are quite similar in that they have low averages for the precondition and only associated with 1 timeslot. The dataset from Uzun takes the opposite approach and saturates each rule with many precondition roles and timeslots. Unfortunately, due to the randomness of Uzun's dataset generation, the probability is very low that any of these rules will have assignable preconditions/timeslot arrays.

4.4.4 Role Types

Figure 4.5 shows a breakdown of each policy and how many unique roles appear in the role types: admin, target, precondition, positive precondition, and negative precondition. From our description of Jayaraman and Uzun's dataset, we know that they utilize separate administer, and thus have no roles in the admin slot. From Section 4.3.2, we know that

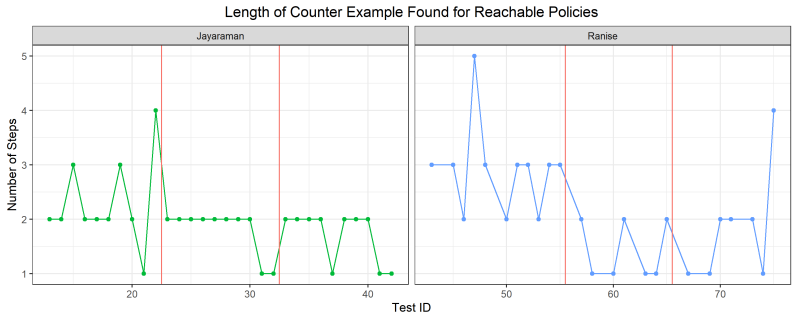


Figure 4.6: Length of the counter example returned by Cree for all reachable policies. Uzun is not present due to no reachable policies. The longest path length is 5 rule firings.

the number of administrator roles is a source of complexity. Only the Ranise's University (Test ID ∼70) and Hospital (Test ID ∼60) policies contain non-zero administrator roles. The largest number of administrator roles is 12 unique roles.

The size of the target role set determines the number of bits that are required for a safety analysis tool to store per user in the TUA for Can Assign and Can Revoke rules, and per role in the RS for Can Enable and Can Disable rules. Jayaraman has the largest set of target roles, there are no roles that appear in preconditions that do not appear in a target role. The same cannot be said for Ranise first dataset (left side) or Uzun; where more roles appear in the precondition than do in the target role.

Positive-Only and Negative-Only precondition roles are not a source of complexity, due to the fact that they will never need to be acquired and then un-acquired. We see that Jayaraman have the largest set of roles that appear as both positive and negative in preconditions; this only occurs in their "mixed" (Test ID ∼40) and "mixednocr" (Test ID ∼30) datasets, their "positive" dataset (Test ID ∼20) only contains positive roles in the preconditions. The datasets from Uzun have an equal number of positive and negative roles in the preconditions, but the length of the positive preconditions is so large that most rules are impossible to fire. The Ranise datasets have an even spread of role types, but the quantity is small.

4.4.5 Path Length of Reachable Policies

For safety queries where a satisfying state is reachable, the minimum number of steps required to reach that state is a source of complexity. In Figure 4.6, we show the minimum number of steps required to solve the reachable policies from Jayaraman and Ranise. Uzun

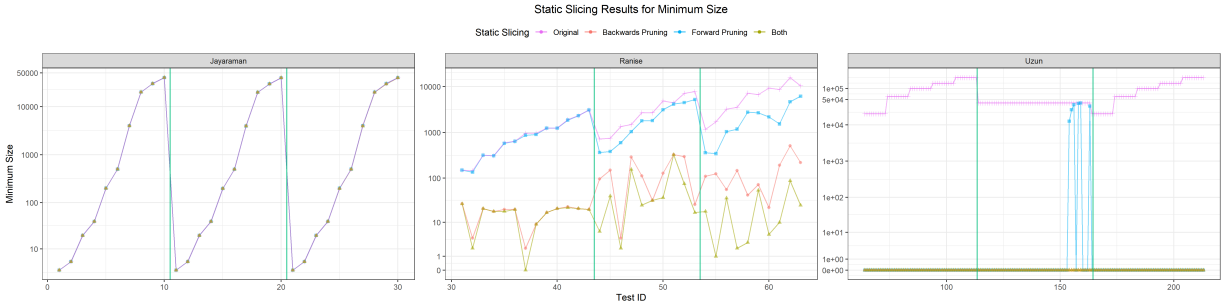


Figure 4.7: The Policy Size of each policy after applying static slicing techniques.

is not present because no test cases were reachable. From this figure we see that the maximum length path required to solve all policies from the 3 previous datasets is 5 steps. In Section 4.5.4, we test the performance gains from setting a fixed bound of 6 steps for the Cree solver.

4.5 Performance Techniques for ATRBAC-Safety

Here we address Cree's performance techniques used to combat the sources of complexity in ATRBAC-Safety. We can mitigate the negative performance effects from the sources of complexity by identifying instances where rules/roles/timeslots can be removed, alternative solvers can be used, and reduced limits for the solver's search space. The following techniques are a balance of performance improvements for the set of inputs it relates to and minimizing the overhead added to the policies outside of this set.

4.5.1 Static Slicing

Static slicing is a performance technique which attempts to reduce the size of the policy before executing the safety analysis. There are 2 methods to static slicing: forward pruning and backwards pruning.

Forward Pruning method of removing rules which are impossible to execute. This method of reduction takes rules that contain both a positive role r and a negative role r in the precondition. We also remove all rules where the administrative role can never be assigned and enabled for any of the timeslots in the admin time-interval. Forward slicing

is effective in the following situations: Ill-formed policies, especially randomly generated policies, and sub-policies created when excluding rules from a policy.

Backwards Pruning method of removing rules that are not required to answer the safety query. We perform backwards pruning by starting with the safety query and adding all CanAssign rules where the target role and the timeslot array appear in the safety query. From this initial set of rules, we can add any CanAssign and CanRevoke rules that could be used to satisfy a precondition in our set of rules. We also add CanAssign and CanEnable rules to satisfy any of the administrator role during the administrator time interval. For CanEnable and CanDisable rules we only add CanEnable/CanDisable rules to the set for positive/negative roles in the precondition and CanAssign/CanEnable rules for the administrator role. We continue iterating, adding rules, until we have added all rules that satisfy our set of rules or all rules have been processed.

Forward and backwards pruning are polynomial-time algorithms, and thus including them in the safety analysis does not change the complexity class. Increased performance only occurs if the reduced size of the policy translates to faster solution time minus the static slicing overhead. A smaller sized policy results in a smaller state space, and can reduce the size of the TRBAC state by removing roles/timeslots that were only mentioned in the set of rules that were removed. This method works well for unreachable ATRBAC-Safety policies.

In Figure 4.7, we show the minimum size (as defined in Section 4.4.1) for each policy after running each pruning technique. Static slicing is unable to reduce the size of any policies in the Jayaraman dataset. The Uzun dataset is able to reduce almost all policies to length zero with forward pruning only. Backwards pruning only is able to reduce all Uzun policies to zero. The Ranise dataset sees a small improvement from forward pruning, an even larger improvement from backwards pruning, and the best reduction in size from the combined pruning methods.

4.5.2 Abstraction Refinement

Safety queries often do not require all of the rules in the ATRBAC policy to determine if a policy is reachable. If we assume this, then we can gain a potentially large performance increase by testing sub-sets of the original policy until we find a sub-policy which is reachable or we test the original policy. Abstraction refinement has one sided errors, where if there exists a path from the initial state to the goal state in one of the sub-policies then it exists in the original policy, but the lack of a path in the sub-policy does not mean that a path does not exists in the original policy. This method is only increases performance

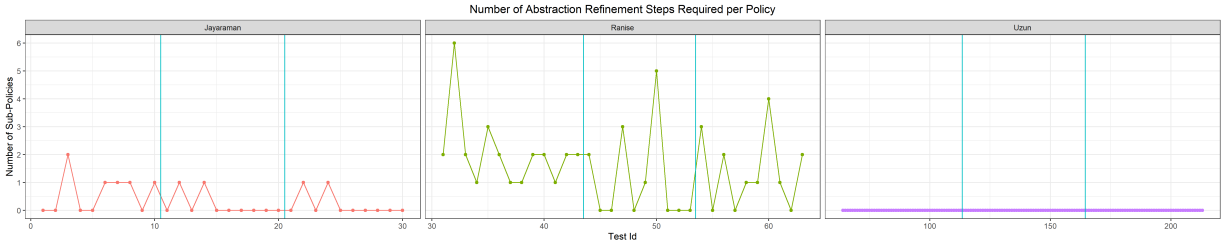


Figure 4.8: The number of required abstraction refinement steps used to solve each policy from the prior datasets. The more steps required, the more difficult the policy. 0 steps means the solver is called once.

for large policies that have a reachable safety query. This method supports parallelizing, where we can perform safety analysis of multiple sub-policies in parallel and stop if the safety query is satisfied or if the original policy finishes running. Given enough resources for parallelization, this method will take negligible amount of extra time than just running the original policy and potentially much faster.

The abstraction refinement algorithm (see Section 3.5.3) starts with the safety query and find all CanAssign rules where the target role is in the safety query role array and the query timeslot is in the CanAssign target role timeslot array. This is the initial sub-policy.

From our initial policy we can perform a refinement step and create our next sub-policy. Each refinement step uses the previous sub-policy to provide additional constraints that allow for more rules to be included. We build the new sub-policy by copying the old policy and including rules from the original policy that satisfy any of the following conditions:

- All CanAssign rules where the target role appears in the previous policy's CanAssign/-CanRevoke rules as a positive role in the precondition
- All CanDisable rules where the target role appears in the previous policy's CanAssign/-CanRevoke rules as a negative role in the precondition
- All CanEnable rules where the target role appears in the previous policy's CanEnable/-CanDisable rules as a positive role
- All CanAssign and CanEnable rules where the target role appears in the previous policy's CanAssign/CanRevoke/CanEnable/CanDisable rules as an administrative role

The abstraction refinement algorithm is similar to building a dependency graph, but we take intermediate snapshots to test in the hopes that one of those smaller policies will return with a satisfied safety query. Abstraction refinement performs a backwards pruning

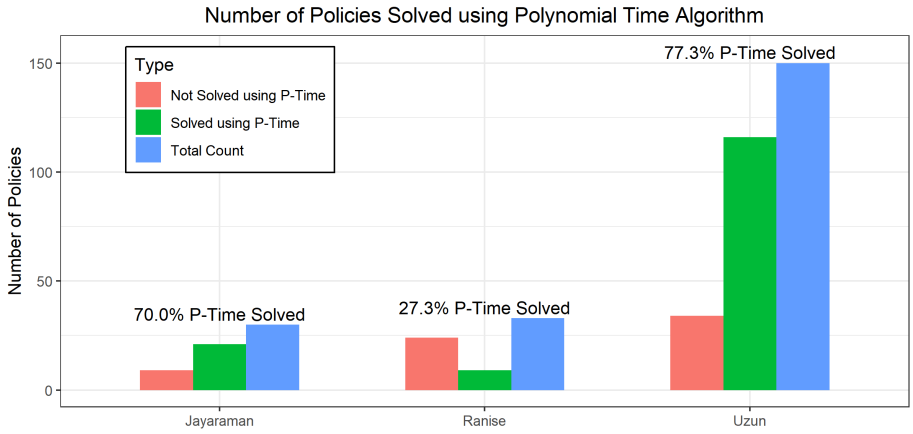


Figure 4.9: Number of policies solved using the Cree's polynomial time algorithm.

technique where the last sub-policy contains all rules that might be required to solve the safety query, and the number of rules is less than or equal to the original policy.

Figure 4.8 shows the number of abstraction refinement steps used to solve each of the prior datasets. From this we can determine how many abstraction steps is required to solve a policy. The more abstraction refinement steps required, the more difficult a policy is to solve when utilizing abstraction refinement. The Uzun dataset is able to be solved with no abstraction refinement steps, meaning one call to the solver. The Jayaraman dataset requires very few abstraction refinement steps. The Ranise dataset requires the most abstraction refinement steps, but the maximum number is 6 steps. Six abstraction refinement steps requires 7 calls to the solver, but we can see that the path length (from Figure 4.6) only requires 3 steps for the model checker to solve.

4.5.3 Polynomial Time Safety Solver (Quick Decisions)

In ATRBAC-Safety there exists policies which can be solved using polynomial time algorithms. This set of policies is vast and varied. The following sets have been identified in Section 3.5.1.

Empty Query Role Array A policy is insecure when no roles are provided in the safety query's role array. This result is true for all instances because all users satisfy this condition, irregardless of the roles they are enrolled in.

No CanAssign Rules for the Safety Query's Role Array and Timeslot The safety query is unreachable because the initial conditions have no users assigned to any roles. With no CanAssign rules to obtain the query's roles, for the query's timeslot, it is impossible for a user to obtain the goal roles.

No Truly-Startable CanAssign Rules A Truly Startable rule is any rule that has the precondition and the administrator condition set to TRUE. This means that a Truly-Startable CanAssign rule is satisfied in the initial conditions. If no Truly-Startable CanAssign rules exists then the initial user-role assignment TUA can never be changed. Thus the policy is secure.

Query Truly-Startable CanAssign Rules The safety query is reachable if there exists a set of truly-startable rules that can satisfy the safety query.

Figure 4.9 shows the effectiveness of the current polynomial time algorithms, where it is able to solve: 77.3% of the Uzun dataset, 70% of the Jayaraman dataset, and 27.3% of the Ranise dataset. The results for Jayaraman and Uzun are quite impressive, as it shows the very large sizes each of these datasets have, does not require complex, nor involved, algorithms to solve them. From Section 4.3, Sources of Complexity, we can identify more classes of input which are polynomially solvable.

4.5.4 Bounded Search of State Space

ATRBAC-Safety is a state search, where a state is represented by a TRBAC policy and the state transitions are the rules from the ATRBAC policy. An ATRBAC policy has a reachable safety query if there exists a path from the initial conditions to a state where the safety query is satisfied.

Performing ATRBAC-Safety using state reachability, it is useful to know the minimum number of state transitions, from the initial state, that a satisfying state can be found. Estimating this value allows for a bound to be placed on the state search algorithm.

In Section 3.5.4, we create an algorithm to estimate the bound for any ATRBAC-Safety instance. This algorithm returns the length of the longest simple path that traverses every state that *might* be required to reach the goal state. This algorithm assumes that the current time is not encoded in each state, thus any rule can be applied to a state. This assumption on time greatly reduces the bound and complexity of the estimation algorithm.

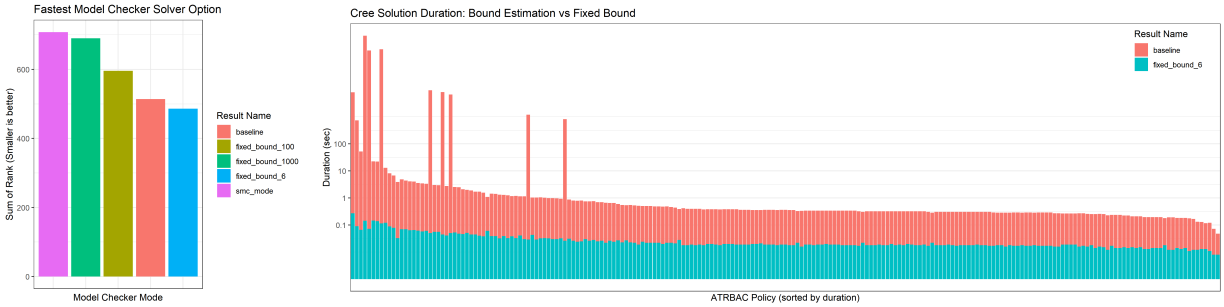


Figure 4.10: The left figure ranks each model checker option is terms of duration and displays the sum of those ranks over all policies. The smaller the value, the faster that option is overall. The fastest option is using a fixed bound set to 6 steps (maximum 5 steps required from Figure 4.6). The right figure shows the durations for all policies for the fixed bound at 6 steps and Cree's Bound Estimation algorithm (baseline).

Figure 4.10, the left figure, shows the relative speeds of the following model checker options.

Cree's Bound Estimation (baseline) Running the model checker in bounded model checking mode and using Cree's bound estimation.

Fixed Bound (fixed_bound_X) Running the model checker in bounded model checking mode and using a fixed bound of X. The value 6 is used from the analysis in Section 4.4.5.

Symbolic Mode (smc_mode) Running the model checker in symbolic model checking mode and supplying no bound.

From Figure 4.10, we see that the fastest model checking option is using the fixed bound of 6, which is the smallest value that can satisfy all reachable policies. The second fastest option is using Cree's bound estimation. As the fixed bound gets larger, the speed reduces. The slowest option is the symbolic mode.

We plot the difference in duration between the fastest and second fastest options, in Figure 4.10. For the majority of test cases, using Cree's bound estimation is 1 second or faster. This shows that the bound estimation helps, but improvement can still be made for specific policies where an unfavourable bound is provided.

In Figure 4.11 the bound estimation for each policy is provided.We can see that the bound estimation can produce quite a large upper bound for non-trivial ATRBAC-Safety policies. The number can be so large that it is actually more performant to run in an

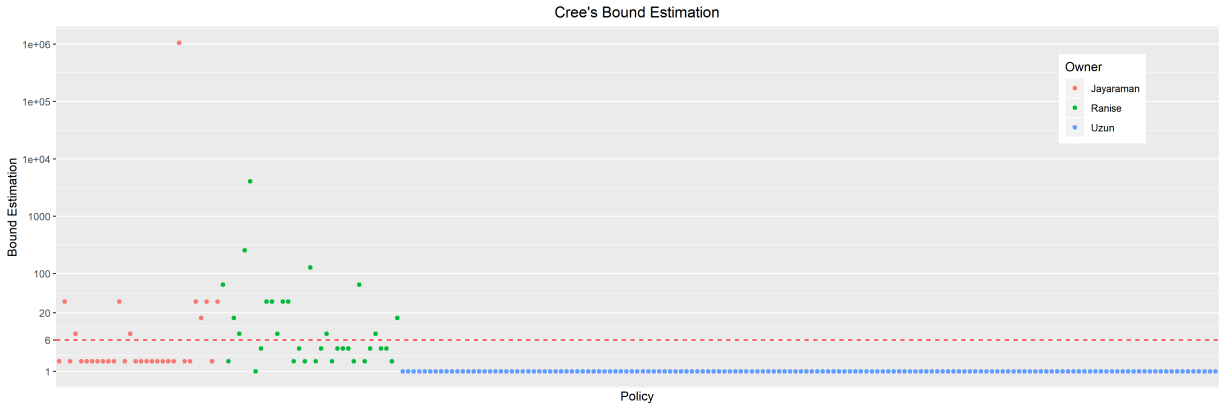


Figure 4.11: Cree's Bound Estimation for all previous datasets.

infinite bound setting or using symbolic model checking mode. These cases are rare in the prior datasets, as most of the estimations occur below the fixed bound of 6. Many times the bound is 1, which is the smallest bound.

4.6 Generating New Hard ATRBAC Instances

This section briefly describes two new techniques for generating hard instances of ATRBAC-Safety. Both techniques from below focus on creating hard instances by creating long reachable paths. From Table 4.1, we know that the longest reachable path is 5 rule executions. I plan to create hard instances by forcing the shortest reachable path to be much larger than 5.

Each dataset is generated with a script that accepts 1 input: *Input Policy Size*. Both datasets can be found in [104], and the source code is publically hosted here [113].

Version 1 Creates a dependency list, where for an input policy size of n, it requires the safety query from input policy size $n-1$ for a new admin role. That new admin role is the only admin role able to satisfy the safety query using a Can Assign rule on the target user. This new admin role cannot be assigned to any user who is a member of the previous admin roles. This new admin role is the only role allowed to assign the goal role to a user. Only Can Assign and Can Enable rules are used. All roles are un-revokable and un-disableable. There is no randomization used in the generation of these policies.

Version 2 Similar premise to Version 1, but this version introduces mixed roles into the

admin dependency graph. Each input policy size n is built using the input policy size $n-1$. Admin users must obtain the previous admin role in order to assign the new admin role, but this chain forces roles to be obtained and then revoked to obtain another role. This introduces an intermediary role that is required to reach a satisfying state but must be removed to allow for enrollment of un-revokable roles. Utilizes the rule types Can Assign, Can Revoke, and Can Enable. There is no randomization used in the generation of these policies.

4.7 Empirical Analysis

Here we will analyze the newly formed datasets, where the *input policy sizes* used to create the policies are: 1-10, 25, 50, 100, and 200. To minimize the size of this chapter, all figures were moved to Appendix E.

Policy Size Each policy is created using an input policy size. This value is used in an iterative process to create new policy. The actual size of the policies created is shown in Figure E.1. Each version has a linear growth in the number of rules/roles that are used for each policy size. The minimum path policy size shows a polynomial growth with respect to the input policy size. The size of these datasets is similar to all 3 prior datasets in terms of number of rules and the minimum path policy size.

Rule Types Figure E.2 shows the percentage distribution of rule types for each of the new datasets. Version 1 has only Can Assign and Can Enable rules, with Can Assign being the majority. The lack of Can Revoke and Can Disable rules show that all roles are un-revokable and un-disableable. Version 2 has Can Assign, Can Revoke, and Can Enable rules, with Can Assign as the majority and equal parts Can Revoke and Can Enable. The introduction of Can Revoke rules brings revokable roles, and this allows for temporarily assigned roles.

Rule Properties Figure E.3 shows the number of rules that fall within well defined properties. Version 1 and 2 contain only Invokable rules and no rules with Unassignable Preconditions. Both datasets have equal number of Truly Startable and Startable rules, and these grow linearly with the input policy size (All Truly Startable rules are Startable rules, thus these sets are identical). Both datasets have more Invokable rules than Startable Rules, thus we know that the reachable states are at minimum more than 1 steps from the initial conditions. Both datasets have the number of rules grow linearly, but given the

same input policy size, Version 2 creates more rules.

Precondition and Timeslot Array Lengths The average precondition and timeslot array lengths are show in Figure E.4. The average time slot array for Version 1 and Version 2 is 1, as there is only one time slot for each dataset. Version 1 has an average precondition length of 1 (`TRUE` as a precondition has length 0), thus we have fairly simple rules similar to Jayaraman and Ranise datasets. Version 2 has an average precondition length that grows linearly with the input policy size, these produces complex rules and can make manual analysis very difficult. Version 2 has a maximum precondition length of 80, which makes this dataset fit between Ranise/Jayaraman and Uzun.

Role Types The number of unique roles that appear in specific role types is shown in Figure E.5. Both datasets have a linear growth of roles for each of the role types for the given input policy size. Version 1 has equal number of roles that appear as positive and negative preconditions. These sets are not equal, as the set of roles which appear as a precondition role is double the size of the positive and negative precondition role sets. The number of roles that appear as an admin role and as a target role are equal (target role set has 1 extra: the safety query role). Version 2 has the smallest number of roles that appear as administer roles, followed by double the number of roles that appear as negative preconditions. The number of roles for that appear as target, positive precondition, and in the precondition is 3 times the number of administrator roles. These datasets are very different from the prior datasets for distribution of role types.

Expected Path Length Without validation from a solver, we do not know the minimum path length, thus the expected path length is provided in Figure E.6. Each dataset generator builds a counter example path as it builds the policy. This is used to show the steps required to reach a satisfying state. Almost all policies in the new datasets, have an expected path length much greater than 5 (the previous longest path). For an input policy size of 200, the expected path length for Version 1 is 20498 steps and Version 2 is 20900. The growth of the expected path grows quadratically with the input policy size.

Static Slicing The results for each Static Slicing technique can be found in Figure E.7. The pruning techniques are unable to reduce the size of any of the policies. This is similar to the Jayaraman dataset.

Abstraction Refinement Every rule is required in the new datasets to reach a satisfying state, thus only the last abstraction refinement step will produce a result. Figure E.8 shows the number of abstraction refinement steps required to solve each policy. For Version 1, the number of abstraction refinement steps grow linearly with the input policy size. Each

of these refinement steps is wasted computation due to the design of these datasets. This is much greater than all previous datasets, where the maximum abstraction refinement steps was previously 6. For Version 2 the number abstraction refinement steps is fixed at 2, thus only 1 abstraction refinement step is wasted. This is very similar to the Jayaraman dataset.

Polynomial Time Algorithms Only the smallest sized policy for Version 1 was able to be solved using the polynomial timed algorithm. The results for this experiment can be found in Figure E.9. This is much better than all prior datasets.

Bounded Search Cree's bound estimation algorithm produced bounds that exceed the maximum integer value for both new datasets. For Version 1 there are 5 usable bounds, and Version 2 has 3 usable bounds. The results for this experiment can be found in Figure E.10. The lack of usable bound will reduce the speed of solution since infinite bounded model checking is required to be used. The bound estimation algorithm uses the longest simple path to estimation the max-min path required to solve the policy (see Section 3.5.4), since both datasets have very long path lengths, which require every rule, this creates an exponential bound estimation. The estimation for these datasets is far larger than all previous datasets.

Cree Solver Duration This experiment runs the Cree ATRBAC-Safety solver, using multiple modes, against the new datasets for a 1hr time limit. From the results above, we note the very large bound estimation for both new datasets, and for Version 1 the number of abstraction refinement steps grows linearly with the input policy size. Thus we tested Cree using the following combination of modes: Symbolic Model Checker/Bounded Model Checking and Abstraction Refinement/No Abstraction Refinement. The results for this experiment can be found in Figure E.11.

From Figure E.11 we notice the same result as with Section 4.5.4, where bounded model checking gets slower as the bound is increased to very large values. Here we see that the bound is so high that symbolic model checking is the faster option. Both SMC modes are faster than either BMC modes for the majority of tests.

The effects of Abstraction Refinement, for both datasets, is seen best when comparing the BMC mode. For Version 1, the non-Abstraction Refinement BMC mode is faster and able to solve 1 additional policy compared to Abstraction Refinement in BMC mode. Since the abstraction refinement steps for Version 2 is fixed at 2 steps, the timing difference is not as pronounced, but non-Abstraction Refinement is still faster.

When comparing the policy sizes, using Figure E.1 and Figure 4.1, we see that the prior datasets include larger policies in terms of number of rules, roles, timeslots, and

minimum path size, but all of those policies were able to be solved within the 1 hr time limit. The maximum input policy size able to solve by Cree within the 1hr limit: 6 for Version 1 and 4 for Version 2. This shows that the new datasets are indeed hard instances of ATRBAC-Safety.

4.8 Future Work

The new datasets provide much in the way of missing sources of complexity, but future work is needed for utilizing time slots and adding more difficult Can Enable and Can Disable rules. Future work is required into creating better bound estimation. Static slicing had no effect on the new datasets because all rules are required to obtain the reachable state. Future work into adding noise, in a intelligent method, to the new datasets.

The new datasets all have reachable safety queries. Random rule dropout would require inserting another rule, otherwise the polynomial time solver could be used to show no path exists. Future work into an intelligent, and random, method for creating unreachable policies.

4.9 Conclusions

Section 4.3 identified multiple sources of complexity that exists within ATRBAC-Safety. These sources were split into two categories: ATRBAC-Safety Solver sources and Sources that exists within an instance of ATRBAC-Safety. From these sources, in Section 4.4 we analyzed the previous datasets released by Jayaraman et al. [56], Ranise et al. [93], and Uzun et al. [133]. A new singular value for measuring policy size is introduced in Section 4.4.1. Important rule and role properties are defined in Section 4.4.2 and Section 4.4.4. From our analysis, we found that the longest reachable path from all previous datasets was 5 steps.

Cree introduced 4 generic performance techniques that can be used by all ATRBAC-Solvers. In Section 4.5, we explore how effective these techniques are with the previous datasets.

From our analysis, in Section 4.6 we created 2 techniques for generating new ATRBAC-Safety policies. These new policies all contain reachable safety queries, which require a large number of steps to reach. In Section 4.7, we analyze these new datasets and compare them to the prior datasets. For similar instance sizes to the prior datasets, we were able to

achieve very hard instances of ATRBAC-Safety. Given a 1 hour time limit, the largest policy solved by Cree for Version 1 had an input policy size of 6 and for Version 2 had an input policy size 4. All other test cases were unable to be solved within the time limit. The size of the smallest unsolvable policy is much smaller than the hardest policies from the prior datasets.

All source code and the new datasets are available as open source [113, 104].

Chapter 5

Safety Analysis for Ethereum Smart Contracts

Contents

Declaration of Contributions

I am the sole author of the content within this chapter. The exception to this is the implementation of: state rollback and require statements, function modifiers, and the Enumeration data type. These were created by Alireza Takami. These implementations were code reviewed and modified by myself.

5.1 Introduction

The Ethereum Blockchain is a relatively new technology. In July 2015 the first block on the Ethereum blockchain was mined. Ethereum combines the security, immutability, and public record keeping of block chain technology with distributed computing. This combination allows for the creation of distributed applications where the code is in the block chain, called Smart Contracts.

A prominent use for smart contracts is as a virtual, public, and cheap escrow for contracts between 2 or more parties. The ability to act as a trusted escrow, works through the use of converting currency into Ethereum's Ether currency, transferring it to a smart contract's balance, and then the other party digitizes their asset (if able), once both parties agree, then the smart contract sends the balance and asset to the correct parties. Every step of the transaction is publicly available and is confirmed by a majority of the nodes in the network. This public record can act as a receipt for both parties. The smart contract is able to remove banks and notaries that are usually required to hold the assets and to witness the signing of the agreement.

The ability for smart contracts to act as escrows is important and a very useful task, but there are many other uses that have been created for smart contracts. Here is a list of the most popular smart contract categories:

- **Games** – Popular games revolve around trading cards/assets. Other game companies utilize the Ethereum network to accept payments and handle asset delivery (i.e. custom skins, in game currency, downloadable content).
- **Initial Coin Offering (ICO)** – A fund raising effort for new businesses where coin shares have the ability to be given special meaning (i.e. voting power, dividends).
- **Buy/Sell Stores and Auctions** – The ability to replace e-commerce websites by transferring ether for access tokens or receipts of sale.
- **Tokens/Gift Cards** – The ability to exchange real money/ether for unique tokens that have arbitrary meaning and worth. These tokens can have a one-to-one worth of Ether, but are restricted to only be used in a particular store (gift card replacement).

5.1.1 Prior Work

Prior work which is relevant to ours can be categorized into the following: Blockchain, Cryptocurrencies, the Ethereum Network, Safety Analysis, Vulnerability analysis, and For-

mal Verification. A comprehensive survey of these is beyond the scope of this work. In this section we will specifically discuss work into safety and vulnerability analysis for Ethereum Smart Contracts.

Ethereum Smart Contracts are typically written in a higher level language (ex: Solidity, vyper, etc) and then compiled into EVM byte code. The EVM byte code is what is stored on the Ethereum network and it what the is executed in the EVM. Work has been done to reverse the public EVM byte code stored on the Ethereum network to Solidity/Vyper code (Panoramix [36] and EthRays [22]). Panoramix uses symbolic execution as a means of decompilation, and is the decompiler found on EtherScan. EthRays is a EVM byte code decompiler created for the DEFCON Capture the Flag (CTF) 2018 game. The following tools can be be used to convert EVM byte code to assembly [58, 30, 29]. Erays [161] reverse engineers the EVM byte code into pseudo code

Work has been proposed which performs analysis looking for specific vulnerabilities in [132].

Prior work using Satisfiability Modulo Theories (SMT) based symbolic execution tools are found in Oyente [72], Mythril [27], MAIAN [77], a tool created by Park et al. [83], VeriSmart [124]. Oyente utilizes assert statements for encoding safety queries, whereas the other tools check for specific known vulnerabilities in smart contracts. Each tool simulates the Ethereum Virtual Machine (EVM). Formal verification tool SMTChecker [3], is a new experimental tool that utilizes `require` as assumptions and `assert` statements for specifications. The tool created in [83], by Park et. al., uses the formal semantics of KEVM (see [50]) and performs analysis on the ERC20 token, Ethereum Casper, and DappHub MakerDAO contracts. Solc-Verify, by Hajdu et. al. [46], is similar to the above, except verifies based on a Solidity smart contract instead of the underlining EVM byte code.

Other reductions can be found to Why3 [39], F* [13], LLVM [61], and Event-B [163]. F* and LLVM reduction are only able to verify specific vulnerabilities, whereas the Why3 reduction allows for user provided assertions (similar to Oyente). The reduction to Event-B utilizes a similar approach of reducing the grammar of Solidity and then performing their reduction to Event-B.

VerX, by Permenev et. al. [85], creates a closed source tool which is able to verify temporal properties. A reduced set of features from Solidity is used: loop are restricted, no recursion, no direct storage via assembly, no execution of external code, and no creation/destruction of contracts. Symbolic execution is performed using Z3.

ETHBMC, by Frank et al. [40], creates a bounded model checker which models the Ethereum virtual machine (EVM) using precise memory model. ETHBMC models the

EVM as an Abstract State Machine and utilizes SMT to verify that execution paths are feasible. This work differs from ours as they are creating a model checker and this work is creating a reduction to a model checker. Another difference is that our work use Solidity as the input language, and ETHBMC use EVM byte code. Our work is general safety analysis, which allows for more general analysis than ETHBMC, which limits the attacker model only the following options: Attacker stealing/extracting Ether, Attacker redirection execution flow, and Attacker self destructing the smart contract.

Solidifier, by Antonino et. al. [5], creates an intermediate language called Solid and from Solid reduce to model checking. VeriSol, by Wang et. al. [139], is very similar to Solidifier. Solidifier provides an empirical analysis comparing against VeriSol [139], solc-verify [46], and Mythril [27].

5.2 Ethereum Blockchain Background

The Ethereum Blockchain background has been moved to Section 2.3.

5.3 Reduced-Solidity

Ethereum's Solidity language, and corresponding EVM byte code, is a very powerful language. We introduce a subset of the language, called Reduced-Solidity. Safety analysis of this language will be shown to be **PSPACE**-complete. Figure 5.1 is smart contract using only the features available in Reduced-Solidity. Below we outline the supported features in Reduced-Solidity.

Data Types We limit the data type to fixed size data types only. The list of fixed size data types includes: Boolean (1 byte), signed/unsigned integers (32 bytes), signed/unsigned fixed point numbers (42 bytes), and address (20 bytes). Fixed sized arrays for all data types above and ENUM (converted to integer) are supported. Instances of other Reduced-Solidity Smart Contracts are allowed as a data type. Mappings and variable-sized arrays (`bytes` and `string`) are not supported due to their ability to create exponential sized states.

Delegate Calls Delegate calls are not supported in reduced-Solidity. All delegate calls can be replaced with a function call, and copying over the delegate code into the contract.

```
contract SimpleContract {
  enum State {locked,unlocked}
  State public lock = State.locked;
  uint callCount = 0; uint maxCount = 5; uint b = 0;
  /*!QUERY: lock > 0; UNREACHABLE*/
  /*!QUERY: callCount > 0; REACHABLE*/
  /*!QUERY: callCount > maxCount; UNREACHABLE*/
  /*!QUERY: b > 0; UNREACHABLE*/

  modifier modFunc1{
    require(lock == State.unlocked, "Must be unlocked");
    _;
  }
  modifier modFunc2{
    require(callCount < maxCount,"Reached the maximum allowed calls");
    lock = State.unlocked; _; lock = State.locked;
  }

  // This function can NEVER be called
  function mfTest1() public modFunc1 modFunc2 {
    b = b + 1;
    /*!QUERY: TRUE ; UNREACHABLE*/
  }
  // This function can be called at most maxCount Times
  function mfTest2() public modFunc2 modFunc1 {
    callCount = callCount + 1;
    /*!QUERY: TRUE ; REACHABLE*/
  }
}
```

Figure 5.1: Example Smart Contract, using Reduced Solidity and included embedded Safety Queries.

This allows for less complexity on the part of the solver, without loss of generality.

Inline Assembly Inline assembly is not supported to restrict state size, and the complexity of the solver. Allowing inline assembly would require adding: a stack, memory locations, registers, and support for the many EVM assembly commands.

Available Commands Many Smart Contracts can be made with only the following subset of Solidity commands:

- State Variable Creation, Initialization, and Modification Statements
- Function Parameters and Local Variables
- Functions and Constructors
- Function Modifiers
- If/Else Statements
- Loop Statements (for, while)
- Require Statements and Rolling Back State

- Address Balance and Transfer for only Addresses provided

Embedded Safety Queries In this chapter we choose to differentiate between: `require`, `assert`, and safety queries. We see each having a distinct meaning. We have thus created a custom comment to act as the safety query. This comment will not effect testing, and will not be deployed to the Ethereum network. Embedding the safety queries into the code allows for more complex queries to be posed. On lines 5-8, in Figure 5.1, we have safety queries that rely on information that is written to the blockchain. On lines 22 and 27, we have safety queries that can interact with transient and temporary information contained within a function.

5.4 Reduced-Solidity Safety Problem

Informally: given a Solidity DApp (set of Smart Contracts), with embedded Safety Queries, and a set of Accounts. For each Safety Query, determine if a path exists from the initial state to a state where the Safety Query is true. Below we formally define Reduced-Solidity Safety.

Inputs A list of Smart Contracts written in Reduced-Solidity, set of embedded Safety Queries, and set of Accounts. The size of the input is equal: size of the Solidity code + size of the deployed smart contracts on Ethereum network + the size of accounts on the Ethereum network.

Initial State The initial state is the blockchain state after deploying the smart contracts. Deploying a smart contract involves an Account and running the constructor. The set of Accounts and Balances are available on the blockchain.

Decision Problem For a single embedded Safety Query, s_q, we return TRUE if there exists a path from the initial state to a state where s_q is satisfied; FALSE otherwise.

Theorem 1. *Reduced-Solidity Safety requires to store, and have accessible, the set of provided Smart Contract state variables and each Account's balance. All other values on the Ethereum Network's block chain is not accessible.*

Proof. In order for the EVM, while running a Solidity smart contract, to access other parts of the block chain, one of the following actions must be performed:

1. Call another Smart Contract using a Call/Delegate Call function (Section 2.3.6).

2. Check balance or send money to an Account.

We only permit smart contract calls for smart contracts that are included in the input, thus those state variables are already included in the state. We limit the interactions with accounts to a set provided in the input, thus those account balances are included in the state. ■

Theorem 2. *Reduced-Solidity Safety is in* ***PSPACE****.*

Proof. We will show that Reduced-Solidity Safety is in **NPSPACE** and from the Savitch's theorem [102], **PSPACE** = **NPSPACE**, we can show that Reduced-Solidity Safety is in **PSPACE**.

Reduced-Solidity does not allow reading arbitrary values from the blockchain. Thus the only values from the blockchain that need to be stored in the state are:

- Account Balances,
- State Variables for each Smart Contract, and
- Function Parameters and Internal Variables

The account balance is stored as an unsigned integer, and all data types in Reduced-Solidity are a fixed size. The size of the above state is polynomial to the size of the input. Thus we can create a Turing machine, with size polynomial to the input, and where the alphabet simulates the available actions in Reduced-Safety.

The alphabet used to simulate the EVM, is required to simulate transactions and executing smart contract functions. All Solidity Smart Contracts functions halt, this is due to the gas limit and is described in Section 2.3.7. Transactions are only added to the blockchain if they are successful. A transaction can fail in 2 ways, a require statement fails or the transaction runs out of gas. Thus our Turing machine needs some method of reverting to the state before the transaction, in the event that a transaction fails. A simple method is to duplicate the state parameters, copy the state into this backup state before executing a transaction, and revert the state using the backup values in the case of a failed transaction, otherwise do nothing. Thus we correctly simulate the EVM, ensure we halt, and ensure the state is polynomial to the size of the input.

If a path exists from the initial state to a state which satisfies the safety query, then a non-deterministic Turing machine can create a finite path of non-deterministic actions to this state and return TRUE for the decision problem. If no path exists, the Turing machine will visit all, finite, states and halt and return FALSE for the decision problem. Thus Reduced-Solidity Safety is in **NPSPACE** and from Savitch's theorem [102], **NPSPACE** = **PSPACE**, we show that Reduced-Solidity Safety is in **PSPACE**. ■

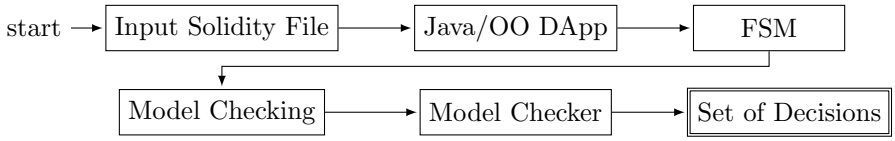


Figure 5.2: Flow diagram of the steps required for the reduction from Reduced-Solidity-Safety to Model Checking.

Theorem 3. *Reduced-Solidity Safety is* ***PSPACE****-complete.*

Proof. In Section 6.4, we reduce ATRBAC-Safety policies to Baked ATRBAC Smart Contracts. We know from the work in Chapter 3 that the number of users required for ATRBAC-Safety analysis is equal to the number of administrator role + 1. The state of the reduced smart contract is described in Section 6.4.1. From this we can convert the `mapping` data type to a fixed array for the address, we can also convert the mapping for role id and timeslot id to fixed arrays. This new baked smart contract is now in Reduced-Solidity formatting. In Section 6.4.3 we convert the safety query to a "bug bounty" query. We can put a TRUE safety query (line 22 of Figure 5.1) at the end of this `withdraw` function to create an embedded safety query.

This is a correct reduction from ATRBAC-Safety to Reduced-Solidity Safety ATRBAC-Safety is **PSPACE**-complete and Reduced-Solidity Safety is in **PSPACE**. Thus Reduced-Solidity Safety is **PSPACE**-complete. ■

5.5 Reduction to Model Checking

The reduction from Reduced-Solidity Safety to Model Checking is outlined by the steps shown in Figure 5.2. The input Solidity file contains all Smart Contracts related to the safety problem. This file is converted into an internal Java object oriented (OO) representation. This representation is converted into a finite state machine (FSM). The FSM is converted to model checking, with all safety queries embedded. The model checker runs and returns the solutions for each model checking specification.

5.5.1 Experimental Reduced Solidity Features

The experimental implementation in this chapter only contains a subset of Reduced-Solidity's supported features. Below is a list of the available features.

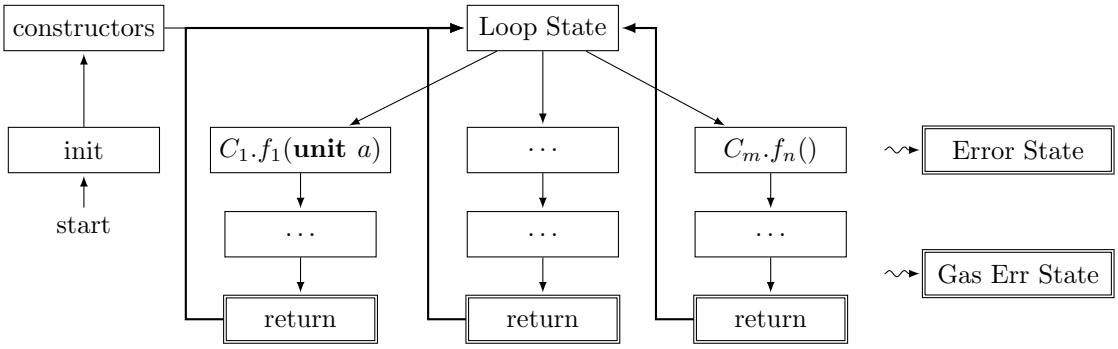


Figure 5.3: Diagram of the Finite State Machine that is created from a single DApp. The "..." state contains multiple states to simulate the function, all other states are added to simulate Ethereum transactions.

- Multiple Contracts in a single file
- State Variables, Function Parameters, and Local Variables
- Data Types: integer, unsigned integer, Boolean, Enum
- Data Change Statements (ex: `a = a + 5`, `b=a & c & d`)
- Constructors and Functions
- Function Modifiers
- If Statements
- Require Statements
- Rolling Back State
- Optional Parameters: Gas Limit, INT/UINT limits

Using preprocessing, we can remove static loops through unrolling. Using state variables, we can simulate transferring ether from one Account to another.

5.5.2 Finite State Machine

From the Java OO DApp we create a Finite State Machine (FSM), which is used to simulate the DApp and the Ethereum network in model checking. The outline of the FSM is shown in Figure 5.3. The *init* state is used to start the FSM. The *constructor* states run the constructor for all Smart Contracts in the DApp in the order that they are presented in the Solidity file. Once all constructors are sequentially created, the FSM enters the Loop State. The *Loop State* represents the time between mined blocks on the Ethereum Network, when returning to the loop state this symbolizes a new block has been mined. The loop state is the special state that allows the model checker to use its internal heuristics to select

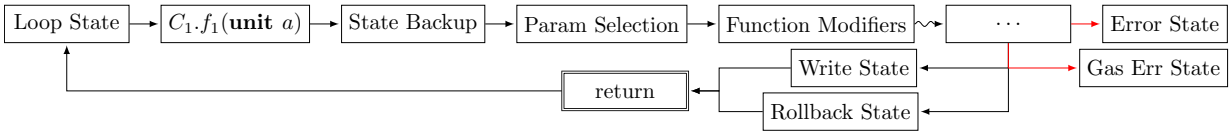


Figure 5.4: Expanded view of a single function from Figure 5.3. The "..." state contains many states that represent the actual logic of the function.

the next function to run. Once a function has finished running, it will return the output and return the Loop State.

Figure 5.4 shows an expanded view of the states that are called after the model checker selects a function from the Loop State. The first state is called the Function Entry State, shown as $C_1.f_1(\textbf{uint } a)$. This state locks the model checker's heuristic choice. *State Backup* copies the value of any state variables used in this function to temporary variables. The temporary variables are used in place of state variables for all operations. The *Parameter Selection* state allows the model checker to select any function parameter values. The *Function Modifiers* states sequentially perform the function modifiers. Function modifiers can run code before and after a function is called, thus there is an implicit *Function Modifiers* states at the end of "..." state.

The states denoted using ..., simulate the function and any function modifiers that have code after the function completes. During these states, if a require statement fails, then the next state is Rollback State. The *Rollback State* does not do anything, since we only used temporary variables, but this state allows for safety queries to incorporate the rollback. (called the Gas Limit Section 2.3.5). If the function runs out of gas, the next function is the *Gas Error State*.

In Ethereum, integers automatically overflow and underflow for arthritic operations, we provide an optional parameter to simulate overflow/underflow on a limited ranged integer or to go to the *Error State*. All transaction in Ethereum are provided with a Gas Limit (see Section 2.3.5), where each operation costs gas, and if the gas limit is reached then the transaction will fail, and all of the gas will be paid to the miner. We have provided an optional parameter, in Section 5.5.6, to simulate a gas limit on the Attacker. If the sum total of gas, over all states and transactions, is more than the attacker's gas limit then the next state is *Gas Error State*. If no errors occur, then the function ends with the *Write State*. In this state the temporary variables are written to the state variables. The *Return State* is where the return value is passed to the calling function (if this is not called from the loop state).

5.5.3 Available Queries

Due to the ability for functions to rollback, and for state variables to have multiple values during a function call, but only the last one is saved to the block chain, several safety queries are required to capture these scenarios.

State Reachability Does a path exist from the initial state to query state s_q (corresponds to a line of code) and then Loop State without rolling back. The State Reachability query is denoted by `/*!QUERY: TRUE ; UNREACHABLE*/`. This query is placed on any line within a Smart Contract function. This appears as a comment to the EVM compiler, and `/*QUERY:` is our symbol to show it is a safety query. "`TRUE`" shows this query is a state reachability query and "`UNREACHABLE`" is where the expected result is printed. The *Expected Result* can be empty, but if provided with either `REACHABLE` or `UNREACHABLE`, it allows for more visible error messages by our solver.

Boolean Expression Reachability Does a path exist from the initial state to the Loop State such that the contract's state satisfies the Boolean expression. This is denoted by `/*!QUERY: <bool_expr> ; UNREACHABLE*/`, in a Smart Contract but outside of a function. Lines 5-8 in Figure 5.1.

State and Boolean Expression Reachability Does a path exist from the initial state to query state s_q, where the contract's state satisfies the Boolean expression, and then back to the Loop State without rolling back (failing a require statement). This Safety Query appears in the Smart Contract, inside of a function. This is denoted by `/*!QUERY: <bool_expr> ; UNREACHABLE*/`. The Boolean expression uses the values in the temporary state variables.

5.5.4 Model Checking Specifics

The reduction to model checking creates a main module, which handles the FSM and the attack gas limit, and a module for each of the smart contracts inside of the DApp.

The main module, outline shown in Figure 5.5, is the entry point for the model checker. There are 3 sections defined in the main module:

- *VAR* – This section is where state variables are defined.
  - `_State` – ENUM type of all states in the FSM.

  - `_QueryName` – ENUM type of all safety queries and their internal names. Used for back tracing the model checking output for specifications to Reduced-Solidity Safety Queries.
  - `_GasLimit` – Integer type, with a range of 0 to the user specified gas limit. This is optional. Defined in Section 5.5.6.
  - `contract1` – All Smart Contracts are formatted as modules, here is where they are instantiated.
- *IVAR* – This section is where the input variables are defined.
  - `nextFunc` – Stores all functions that can be called from the Loop State. This stores the Function Entry State for each function.
- *ASSIGN* – This section is where state variables are initialized and the state transitions are defined.
  - `_State` – The FSM is initialized to the `start` state. The FSM is encoded in the `next` function. When in the *Loop State*, the model checker uses the input variable `nextFunc` to select the next function to run (line 13).
  - `_GasLimit` – The gas limit is initialized to the amount of total gas available. For states that correspond to Solidity code, the `_GasLimit` will be reduced by an estimate cost for that state; other states will not reduce the gas limit. The limit is reduced until 0, then it forces the FSM to go into an error state (line 12).
  - `_QueryName` – The query name has no defined initial value, nor are there any restrictions on the next value it can take. This means that for any state it can take any of the available query names.

Figure 5.6 outlines the features contained in a smart contract module. Each smart contract module is passed a reference to the current state and the query name. The smart contract is not able to change the state, but it uses the current state to change internal state variables and to accurately define the safety queries. The `_QueryName` is indirectly controlled by the smart contracts, when a query is successful, the `_QueryName` must equal the Java internal query name. Since there are no restrictions on the state transition for `_QueryName`, this variable assumes that value to make the model checking specification satisfied.

The *DEFINE* section, in Figure 5.6, allows for all model checking specifications to be defined. The reduction for these queries is defined below. In the *VAR* section the smart contract state variables are defined and limited to this module's scope. This is convenient due to similar scope rules in Solidity. A copy of all state variables, prefix

```
1  MODULE main
2  VAR
3  _State : {start,LoopState, }; -- ... all states
4  _QueryName : {BoolExprReachabilityQuery_0, }; -- ... all query variable names
5  _GasLimit : 0 .. 50000; -- Optional Attacker Gas Limit
6  contract1 : contract_SimCon(_State,_QueryName); -- Each Smart Contract
7  IVAR
8  nextFunc : {LoopState, }; -- ... all starting states for functions
9  ASSIGN
10 init(_State) := start;
11 next(_State) := case
12     _GasLimit = 0 : OutOfGasError;
13     _State = LoopState : nextFunc;
14     _State = start : LoopState;
15     -- ... the Finite State Machine
16     TRUE: _State;
17 esac;
18 init(_GasLimit) := 50000;
19 next(_GasLimit) := case
20     _State = SimCon_mfTest1_FunctionEntryState_ln20 : max(0,_GasLimit - 1);
21     -- ...
22     TRUE : _GasLimit;
23 esac;
```

Figure 5.5: Outline of the Main module for the model checking reduction.

`_Temp_`, is used to accurately simulate rollback for failed transactions. Function parameters and internal variables are also declared here, but do not require any temporary variables. Query and require variables are defined as required. Using modules allows for multiple instantiations and provides scoping rules similar to Solidity; which reduces processing during the reduction.

NuSMV supports the specifications: Computational Tree Logic (CTL) and Linear Temporal Logic (LTL) (see Appendix B for details). Our reduction utilizes CTL specifications for the safety queries. In Figure 5.7, we have the queries from lines 8 and 22 from Figure 5.1. The first query, named `BoolExprReachabilityQuery_3`, verifies if a path from the initial state to the Loop State exists, such that $b > 0$ is TRUE during the Loop State. The second query, name `QueryState_4`, verifies if a path from the initial state to the state represented by line 22, where the Boolean expression `TRUE` is satisfied and then the function returns to the Loop State without rolling back.

The first query is a Boolean Expression Reachability safety query. The reduction creates the following conjunction Boolean expression:

- `_QueryName = BoolExprReachabilityQuery_3` – there is no restriction on `_QueryName`, thus is can take any value. This is used for back tracing NuSMV output.
- `_State = LoopState` – Safety query is only valid during the Loop State.

```
MODULE contract_SimCon(_State,_QueryName)
DEFINE
-- Define all Smart Contract queries (include query name for backreferencing)
CTLSPEC NAME BoolExprReachabilityQuery_0 := AG !(_QueryName = BoolExprReachabilityQuery_0
    & _State = LoopState & StateVar_lock>0);
VAR
-- Define query variables
mfTest1_QueryVar_query_c1f1_QS_ln22 : {not_reached,cond_reached,cond_confirmed};
-- Define all Smart contract State Variables and their Temporary/Working variables for
    rollingback
StateVar_b : 0 .. 200;
_Temp_StateVar_b : 0 .. 200;
-- Define all extra require statement variables
REQUIRE_SimCon_mfTest1_RequireState_ln15 : boolean;
ASSIGN
-- Initialize all variables
init(REQUIRE_SimCon_mfTest1_RequireState_ln15) :=  FALSE;
-- Assign next values for Require statements/State/query/
next(REQUIRE_SimCon_mfTest1_RequireState_ln15) := _Temp_StateVar_callCount<
    _Temp_StateVar_maxCount;
```

Figure 5.6: Outline of the Smart Contract module for the model checking reduction.

- `StateVar_b` > 0 – The state variable (not the temporary variable) satisfies the Boolean condition.

This Boolean expression is negated and the CTL `AG` operator is used. The AG operator verifies that the condition p must be true in all states that can be reached from the initial state. We negate the Boolean expression to return a counter example if the state is reachable.

The second query is a State and Boolean Expression Reachability safety query. This reduction requires an additional state variable: `mfTest1_QueryVar_query_c1f1_QS_ln22`. The transition table for this new variable is:

1. Initialized to `not_reached`.
2. Every Function Entry State resets the value to `not_reached`.
3. When the FSM reaches the Query State, and the Boolean expression is satisfied using the temporary state variables, then the value is changed to `cond_reached`; otherwise it stays `not_reached`.
4. If the function reaches the Function Success State (after the *Write State* in Figure 5.4) and the variable equals `cond_reached`, then it confirms that no roll back has occurred and the query has been satisfied.

The NuSMV specification is similar to the first query, except we use the additional variable to determine if the query is satisfied and we do not need to be in the Loop State.

```
1  MODULE contract_SimCon(_State,_QueryName)
2  DEFINE
3  CTLSPEC NAME BoolExprReachabilityQuery_3 :=
4      AG !(_QueryName = BoolExprReachabilityQuery_3 & _State = LoopState & StateVar_b>0);
5  CTLSPEC NAME QueryState_4 :=
6      AG !(_QueryName = QueryState_4 & mfTest1_QueryVar_query_c1f1_QS_ln22 = cond_confirmed)
           ;
7  VAR
8  mfTest1_QueryVar_query_c1f1_QS_ln22 : {not_reached,cond_reached,cond_confirmed};
9  ASSIGN
10 init(mfTest1_QueryVar_query_c1f1_QS_ln22) := not_reached;
11 next(mfTest1_QueryVar_query_c1f1_QS_ln22) := case
12     _State = SimCon_mfTest1_EmptyState_ln19 : not_reached;
13     _State = SimCon_mfTest1_QueryState_ln3 & TRUE : cond_reached;
14     _State = SimCon_mfTest1_FunctionSuccessState_ln32
15             & mfTest1_QueryVar_query_c1f1_QS_ln22 = cond_reached : cond_confirmed;
16     TRUE : mfTest1_QueryVar_query_c1f1_QS_ln22;
17 esac;
```

Figure 5.7: Reduction for Safety Queries on lines 8 and 22 of Figure 5.1.

In Figure 5.8, we show the reduction of the `require` Solidity statement. Simulating `require` statements need an additional Boolean variable to keep track of the `require` function condition. This variable, prefixed `REQUIRE_`, is initialized to FALSE. The variables state transition table always set the next value to the result of the of the Boolean condition. The Boolean condition uses temporary state variables. The FSMn in the main module, use this variable to decide if execution should continue or if the function needs to roll back. Roll back is simulated by not updating the state variables with the temporary variable values. If execution continues, then we reach the Function Success State and update all state variables with the temporary variable values.

5.5.5 Known Issue

Model Checking is well suited to small action spaces, due to the state explosion when too many actions are available. Currently this experimental version is unable to represent the full range for integer values. The signed and unsigned int data types have too many potential values (2^{256}). This creates a state explosion which makes simple safety queries impossibly long to complete if the model checker is given the opportunity to select one of these values for a function parameter. In the next section we provide a method of reducing the available options for `int` and `uint`.

```
MODULE main
VAR contract1 : contract_SimCon(_State,_QueryName);
ASSIGN
next(_State) := case
    _State = SimCon_mfTest1_RequireState_ln15 & contract1.
        REQUIRE_SimCon_mfTest1_RequireState_ln15 : SimCon_mfTest1_EmptyState_ln8;
    _State = SimCon_mfTest1_RequireState_ln15 & !contract1.
        REQUIRE_SimCon_mfTest1_RequireState_ln15 :
        SimCon_mfTest1_FunctionFailedEndState_ln9;
esac;

MODULE contract_SimCon(_State,_QueryName)
VAR
StateVar_b : 0 .. 200;
REQUIRE_SimCon_mfTest1_RequireState_ln15 : boolean;
ASSIGN
init(StateVar_b) := 0;
init(REQUIRE_SimCon_mfTest1_RequireState_ln15) :=  FALSE;
next(REQUIRE_SimCon_mfTest1_RequireState_ln15) := _Temp_StateVar_callCount<
    _Temp_StateVar_maxCount;
next(StateVar_b) := case
    _State = SimCon_mfTest1_FunctionSuccessState_ln32 : _Temp_StateVar_b;
    TRUE : StateVar_b;
esac;
```

Figure 5.8: Reduction showing Require Statements and State Rollback.

5.5.6 Parameters

The following security and performance parameters were created.

Attacker Gas Limit

We can potentially speed up long running safety analysis through the use of limiting the amount of gas the attacker is willing to spend. Every transaction, function call from the Loop State, must be paid for by the account who is creating the transaction. There can be multiple transactions from the same account inserted into the next block, and attacks can take place over many blocks. Thus we cannot rely upon the Block Gas Limit as a method for limiting the attacker.

In cryptography, factoring large products of primes is considered "secure" for classical computers and given large primes. Given unlimited time and resources an attacker can use brute force and classical computer algorithms to find the large primes that make the factors. We can use this relative security to define the security of a smart contract in terms of the attacker. Realistically, this attacker won't find the factors before the application that used them has changed to new factor or stopped working.

We imitate this by showing that "The Safety Query is unreachable for attackers for up to 100×10^9 gas" (for 10 gwei/gas this equals 1000 eth $\approx$ \$209 $\times 10^3$ USD). The cost in gas per operation is always the same, but the cost of gas fluctuates, so to does the conversion rate from Ether to other currencies (much more fluctuations than more established currencies).

This parameter is under active development and requires more accurate gas cost estimation. Use this parameter with the following command: `/*!AttackGasLimit: <int_gas>;*/`

Signed/Unsigned Integer Limits

Due to the state explosion that occurs when an integer's value need to be selected by the model checker (for function parameters), simple test cases become very hard to solve. To mitigate this issue, the ability to reduce the size of integers has been provided. There exists classes of inputs where the extreme range of integers is not required, i.e. the number of books someone would like to buy. Using this setting can reduce the state space and find reachable solutions faster. The current range for signed integers is -100 to 100, and unsigned integers is 0 to 200. Changes to these limits can be made through the special commands: `/*!UINT_MIN: <uint>;*/`, `/*!UINT_MAX: <uint>;*/`, `/*!INT_MIN: <int>;*/`, and `/*!INT_MAX: <int>;*/`.

Redefining the limits can allow for forcing focus of the model checking to interesting ranges. Ethereum allows for overflow and underflow for integer arithmetic operations, and this is correctly simulated in the reduction. Defining integer ranges that are too small, can result in incorrect safety analysis. Thus careful thought is required when defining the ranges for signed/unsigned integers.

Error on Rollback

Simulating the rollback for failed functions can be expensive for the model checker since it does not end a branch but creates a loop. We can disable the ability to simulate rollback by having failed require statements go to the Error State. This parameter is enabled with the command line option: `-noRollback`.

Error on Integer Overflow/Underflow

Solidity naturally overflow/underflows integer data types. This can cause bugs. This option allows for any underflow/overflow operations to result in the Error State. If a Smart Contract is not using a safe math library, then this option can be used to show how operations would change if they did utilize the Safe Math library.

```
1  contract NonObvious {
2    uint a=3, b = 5, c=0;
3    /*!QUERY: c = 17;REACHABLE*/
4    /*!QUERY: c = 18;REACHABLE*/
5    function func1() public {
6      c = a + b;
7      b = b - 1;
8      a = 2;
9      /*!QUERY: c<a & c<b;UNREACHABLE*/ // BUG
10   }
11   function func2() public {
12     b = b + a;
13   }
14   function func3() public {
15     a = 10;
16   }
17 }
```

Figure 5.9: Non-obvious simple Smart Contract.

5.6 Experimental Results

We have created over 30 Reduced-Solidity Safety test cases to show each supported feature working. These smart contracts will be published along with the source code once it can be made available. Below are 2 select examples which have interesting results.

The first experiment, in Figure 5.9, shows a simple obfuscated contract but where the answers to the safety queries are non-obvious.

The safety query on line 3 is indeed reachable (as the expected result suggests). The counter example below shows the initialization and all functions the model checker selected from `next_func`. The Loop State is removed, but occurs between all functions.

```
start -> constructor -> func2 -> func2 -> func2 -> func1 -> GOAL_STATE
```

The counter example for the safety query on line 4:

```
start -> constructor -> func2 -> func3 -> func1 -> GOAL_STATE
```

The counter example for the safety query on line 9:

```
start -> constructor -> func2 -> func2 -> func3 -> func2 -> func2 -> func2 -> func2
-> func2 -> func2 -> func2 -> func2 -> func2 -> func2 -> func2 -> func2 -> func2
-> func2 -> func2 -> func2 -> func2 -> func2 -> func1 -> GOAL_STATE
```

From the counter examples, it is possible for the safety queries on lines 3 and 4 to be manually validated. The safety query on line 9 is harder to manually validate. This safety query is reachable due to overflowing the value in `c`. The limits of the unsigned integer are set 0 to 200, thus the length of this path is much smaller than it would be if the full 2^{256} unsigned integer range was used.

The second experiment, Figure 5.10, converts a `Purchase` Smart Contract (created by the Ethereum company) into a Reduced-Solidity Safety smart contract. The original Smart

```
// unit mvalue (replaces msg.value), Accounts sender (replaces msg.sender)
contract PurchaseConverted {
  // Removed all code that did not change from original Purchase Smart Contract
  enum Accounts {Zero, Seller, Buyer, Other}
  Accounts public seller, buyer;
  /*!QUERY: buyer = seller; UNREACHABLE */ // POTENTIAL BUG
  /*!QUERY: seller = Accounts.Other; UNREACHABLE */
  constructor(uint mvalue) public payable {
    seller = Accounts.Seller;
    val = mvalue / 2;
    require((2 * val) == mvalue, "Value has to be even.");
  }
  // Abort the purchase and reclaim the ether.
  function abort(Accounts sender) public onlySeller(sender) inState(State.Created) {
    state = State.Inactive;
    // seller.transfer(address(this).balance);
    /*!QUERY: TRUE; REACHABLE */
    /*!QUERY: sender != Accounts.Seller; UNREACHABLE */
  }
  // Confirm the purchase as buyer. Transaction has to include `2 * val` ether.
  function confirmPurchase(Accounts sender, uint mvalue) public payable
      inState(State.Created) condition(mvalue == (2 * val)) {
    buyer = sender;
    state = State.Locked;
    /*!QUERY: TRUE; REACHABLE */
  }
  // Confirm that you (the buyer) received the item. This will release the locked ether.
  function confirmReceived(Accounts sender) public onlyBuyer(sender) inState(State.Locked)
  {
    state = State.Inactive;
    // buyer.transfer(val);
    // seller.transfer(address(this).balance);
    /*!QUERY: TRUE; REACHABLE */
  }
}
```

Figure 5.10: Converting Smart Contract from Section H.2 to current supported features.

Contract has been included for reference in Section H.2. The purpose of this contract is to enable the selling of a product. The Seller deploys the contract and any potential Buyer can start the purchase process. The smart contract in Section H.2 is a fully functioning, Figure 5.10 accurately simulates this contract. Risk is mitigated in this contract by forcing the buyer to send 2 times the cost of the product, they get a refund when they confirm that they have received the product. The seller locks up 2 times the value of the product to show "good will", and the intention of shipping the product. The refund of the extra ether occurs by the buyer in `confirmReceived`.

Converting Section H.2 to Figure 5.10 requires the following steps. Replace addresses with a limited enumeration called Accounts. The model checker now only has to work with 4 accounts: Zero, Seller, Buyer, Other. The Zero address is the default value for address data types. The Seller account is set during the smart contract constructor. Ac-

counts initiating a transaction are now being sent via function parameter instead of using `msg.sender`. The model checker has the option of calling any function with any of the Accounts.

When running this experiment through the reduction and solver, we find that there is an interesting bug for the query on line 9, where the `seller` and the `buyer` can be the same person. Buying your own product might seem innocent, but this can be a type of fraud where buying your own products is used to show shareholders and the public that the company is successful, thus inflating the companies perceived worth.

5.7 Future Work

The following items are scheduled for future work on this project:

- Ethereum Accounts: Address data type, support for `msg.sender` and `msg.value`
- Account Transfer and Balance Queries: `address(0xab12).balance`, `address(0xab12).transfer(amount)`, `address(this)`
- Static Arrays and Mapping data type
- Loop Statements: `while` and `for`, with support of `continue` and `break`
- Intelligent Integer Selection: allow for full integer range but model checker selection is more selective. Similar to the FSM `_State` and `nextFunc`.
- Support more CTL Specifications for more varied Queries. Specifically safety queries concerning account balances given 2 Boolean expressions (`A [p U q]` from Section B.1).

A survey of previous Ethereum attack is written by Atzei et. al. [7] and a quantitative analysis of smart contracts is provided by Chatterjee et. al. [24]. Future work will convert the Smart Contracts used in previous attacks to Reduced-Solidity and then verifying that our approach is able to verify the attack vectors used.

5.8 Conclusions

In this chapter we introduced Reduced-Solidity Safety and showed that it is **PSPACE**-complete. We developed an experimental setup which parses Solidity files, with embedded safety queries as comments, creates a representation in Java, converts to a Finite State Machine, reduces to Model Checking, and solves using a model checker. We outlined each

step in the reduction to show it is in polynomial time and presented 2 example Smart Contracts for experiments. The first contract is small, but has non-obvious results where manual analysis might not be feasible. The second experiment converts a Purchase smart contract to a Reduced-Solidity Safety smart contract, by removing unsupported features and adding security queries. We show that there is a bug in the contract, where the seller and buyer can be the same person. From this work we have shown that safety analysis of Solidity Smart Contracts is possible with the use of a model checker.

Chapter 6

Converting Administrative Temporal Role Based Access Control (ATRBAC) Policies to Ethereum Smart Contracts

Contents

Declaration of Contributions

I am the sole author of this chapter.

6.1 Introduction

Access control systems are a complex and important infrastructure in computer networks. These systems deal with restricting access to many distributed assets, and are required to high a high availability and be very secure. Access control systems provide an attractive point of attack, since compromising these systems can provide access to all resources within a network. Access control systems can be centralized, but are often distributed. If access control is centralized, this provides a single point of failure. This is vulnerable to denial of service attacks, and any malicious changes to the policy are accepted without verification.

Access control in a distributed environment can mitigate the denial of service attacks. Access control on a blockchain can mitigate attacks on a specific node, as a majority of nodes is required to update the ledger in a blockchain. The immutability of blockchain also provides history of all changes and thus visibility to access control attacks. The blockchain makes all access control requests visible, and thus inappropriate and malicious requests can be mitigated by security aware employees.

There are many access control schemes, we will focus on Administrative Temporal Role Based Access Control (ATRBAC). This scheme generalizes the access matrix, RBAC, TRBAC, and ARBAC. In this chapter, we create 2 methods for creating ATRBAC Smart Contracts. The first method is called the Generic Smart Contract, which is a smart contract where the access control policy is built on the Ethereum network after deployment. The second method is called the Baked Smart Contract, where the ATRBAC policy is encoded into the smart contract before deployment and cannot be altered after. We discuss the benefits and costs with their deployment and use in Section 6.5.

6.2 Background

The background for Administrative Temporal Role Based Access Control (ATRBAC) policies is in Section 2.2, the background on the Ethereum network is in Section 2.3. The Cree tool is used to read and model the ATRBAC policies, Chapter 3 describes the tool.

6.3 Generic ATRBAC Smart Contract

The goal of the Generic ATRBAC Smart Contract is to have a single smart contract that can be deployed without modification and can be used to create any ATRBAC policy.

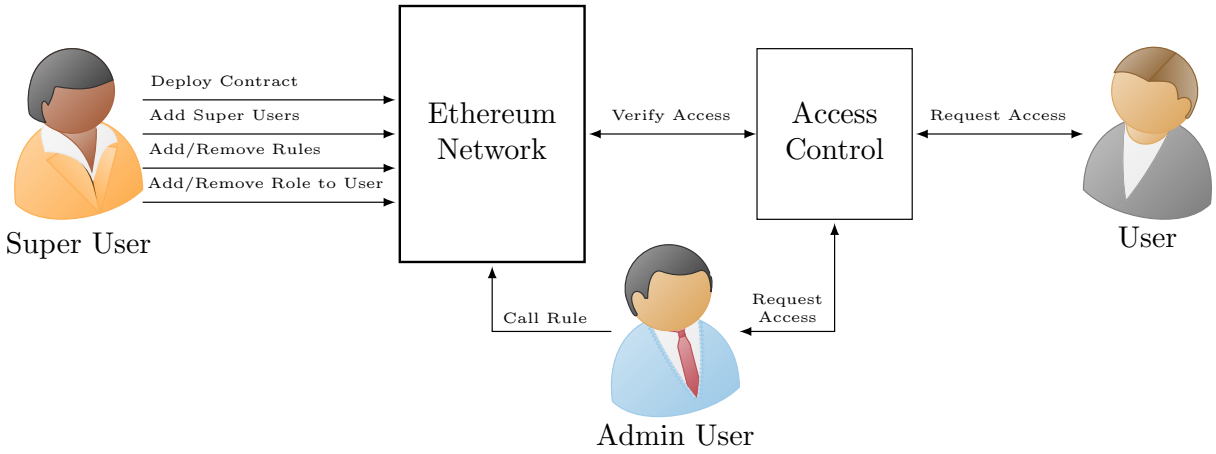


Figure 6.1: All actors and actions which can be used with the Generic ATRBAC Smart Contract.

This smart contract does not require any special tools, besides the tools to compile and deploy to the Ethereum blockchain. After the smart contract is deployed, the account who deployed is now a super user for that contract. This allows them to create the ATRBAC policy and to update it in the future.

The Generic ATRBAC Smart Contract was created during the 36 hour ETH-Waterloo hack-a-thon, held November 2019 in Waterloo Canada. It has since been modified for this chapter and is open source [110]. Another project was created for this chapter, which converts existing ATRBAC policies into `web3` commands to automatically deploy the generic ATRBAC contract and create the ATRBAC policy [112]. The full Generic ATRBAC Smart Contract has been provided in Appendix F.

Once the generic smart contract is deployed to the Ethereum blockchain it can be used to simulate the User-Role aspects of an ATRBAC policy. Figure 6.1 shows the actions that each user role can perform on the contract. Administrative users can fire Can Assign, Can Revoke, Can Enable, and Can Disable rules. The super user is able to deploy, create, and maintain the ATRBAC Smart Contract. All functionalities of ATRBAC policies, except linking role to permissions, is contained in this smart contract. The Role-Permission table is omitted due to storage space cost on the Ethereum blockchain, the Role-Permission table is usually much larger than the User-Role table.

Once an account has deployed the Generic ATRBAC Smart Contract, the account is saved as a Super User. Any Super User can add another account to the Super User role using the `suAddSuperUser(user)`. The purpose the Super User role is to build and update

```
contract ATRBAC {
  RoleTimeslots.Role[] public trbac_state;
  mapping(uint => mapping(uint => bool)) public role_enablement;
  Rule.rule[] ca_rules, cr_rules, ce_rules, cd_rules;
  Timeslot.timeslot[] public timeslots;
  mapping(address=>bool) public super_users;
  constructor() public {  }
  // USER Actions
  function hasAccess(address _user, uint _role, uint _timeslot) public view
    roleExists(_role) timeslotExists(_timeslot) returns(bool) {  }
  function getRoleName(uint _role) public view roleExists(_role) returns(string memory) {}
  function roleEnabled(uint _role, uint _timeslot) public view
    roleExists(_role) timeslotExists(_timeslot) returns(bool) {  }
  function fireCanAssignRule(uint _ruleIndex, address _targetUser) public {  }
  function fireCanRevokeRule(uint _ruleIndex, address _targetUser) public {  }
  function fireCanEnableRule(uint _ruleIndex) public {  }
  function fireCanDisableRule(uint _ruleIndex) public {  }
}
```

Figure 6.2: Outline of the User available data and functions from the ATRBAC Smart Contract. Full code in Appendix F and [110].

rules for the ATRBAC policy. This is how ATRBAC policies are usually administered, where the IT department has super user permissions over the policy and the access control system.

The super user must first create all of role and timeslots that the policy requires, using the contract functions `suAddNewRole(name)` and `suAddNewTimeSlot(start_hr,end_hr)`. These function are used to create ATRBAC rules: `suAddCARule(...)`, `suAddCRRule(...)`, `suAddCERule(...)`, and `suAddCDRule(...)`; the parameters is the same as denoted in Chapter 3 and are described in detail in Appendix F.

Once the ATRBAC rules are completed, the super users can add users to unreachable/restricted positions using the `suAddRoleTimeslotToUser(user,role,timeslot)`. They can also enable any roles that don't have rules attached to them using `suEnableRole(-role,timeslot)`. There are revoke and disable function to remove a role from a user and to disable a role.

The policy is now setup and ready to be connected to an access control centre. Users are identified by their Ethereum wallet (public/private key), which can be generated free of charge and a person can have as many wallets as they wish. When a user makes a request for certain access to an asset, the access control system can check if they have access by checking the current access control state (which is publically viewable) and grant/deny the request.

All non-super user available actions are defined in Figure 6.2. Users who wish to fire a

```
Query : t1 , [ goalRole ]
Expected : REACHABLE
// Path: CE-1 -> CE-2 -> CD-1 -> CE-3 -> CA-1 -> CA-2 -> CR-1 -> CA-3
CanAssign {
/* CA-1 */ < TRUE, t1, TRUE, t1, role3>
/* CA-2 */ <role3 , t1, role3 , t1, role2>
/* CA-3 */ <role3 , t1, role2 & NOT ~ role3 , t1, goalRole>
}
CanRevoke {
/* CR-1 */ <TRUE, t1 , TRUE, t1, role3>
}
CanEnable {
/* CE-1 */ <TRUE, t1 , TRUE, t1, role1>
/* CE-2 */ <TRUE, t1 , role1 , t1, role2>
/* CE-3 */ <TRUE, t1 , role1 & NOT ~ role2 , t1, role3>
}
CanDisable {
/* CD-1 */ <TRUE, t1 , TRUE, t1, role2>
}
```

Figure 6.3: Example ATRBAC Policy where the Safety Query is Reachable.

rule in the ATRBAC policy do so by sending a transaction to the Smart Contract for the functions: `fireCanAssignRule(rule,targetUser)`, `fireCanRevokeRule(rule,targetUser)`, `fireCanEnableRule(rule)`, and `fireCanDisableRule(rule)`.

We created a tool [112] to automatically generate the `web3` commands to deploy the Generic ATRBAC Smart Contract, create all the required roles/timeslots, and create each rule. We `web3` commands to convert the ATRBAC policy in Figure 6.3 into a Generic ATRBAC Smart Contract, are found in Appendix G.

6.4 Baked ATRBAC Smart Contract

An alternative method to the Generic ATRBAC Smart Contract, is to "bake" an existing ATRBAC-policy into a Smart Contract. Where the creation of the smart contract occurs before deployment, and each smart contract only simulates a single instance of ATRBAC policy. This shifts the costs away from users invoking rules to the organization who deploys the contract. Baked smart contracts are immutable and cannot be altered after deployment. The baked contract has no Super Users and only stores the TRBAC state table and Role Enablement table. This project is open source [112].

In the following section we will "bake" the ATRBAC-Safety policy in Figure 6.3 into a Smart contract. This will show a reduction from ATRBAC-Safety to Smart Contract.

```
mapping (address => mapping (uint => mapping (uint => bool))) public trbac_state;
mapping(uint => mapping(uint => bool)) public role_enablement;

modifier noZeroAddress(address account) {
  require(account != address(0), "Address cannot be the zero address");
  _;
}
modifier target(boolean cond) {
  require(cond, "Target User does not satisfy the preconditions for the rule.");
  _;
}
modifier sender(boolean cond) {
  require(cond, "Sender is not authorized to fire this rule at this time.");
  _;
}
modifier role(boolean cond) {
  require(cond, "Target Role does not satisfy the preconditions for the rule.");
  _;
}
```

Figure 6.4: The state variables and function modifiers used in Baked ATRBAC Smart Contracts.

6.4.1 State and Modifiers

The Baked ATRBAC Smart Contract has a smaller state, shown in Figure 6.4, compared to the Generic ATRBAC Smart Contract. The "baked" smart contract's state only stores the TRBAC and Role Enablement tables. The role enablement variable is defined the same as in Figure 6.2. Which is a `mapping` from the id of the role to a mapping from the id of the timeslot to a Boolean value. By default, if a value does not exists in a mapping, it will return the default value of that data type (Boolean = false). This agrees with the initial conditions assumed by ATRBAC-Safety. The TRBAC State is redefined since all roles and time slots are known before hand. This becomes a mapping from the user's account address to a mapping similar to the role enablement variable. This reduces the size of the smart contract from the Generic version, but it removes the ability to reverse lookup roles based on names.

In solidity, a function modifier wraps the statements of a function, which allows for code to be run before and after a function executes. The modifier is provided a callback function (`_;`), which it can choose to call the callback function, force the transaction to fail, or return without calling. In the Baked Smart Contract, we utilize function modifiers to reduce the size each function. In Ethereum, the zero address can never represent a real account, but it is the default value for the address data type, thus it should never be used to indicate a user; this constraint is covered with `noZeroAddress` modifier. In order for an ATRBAC rule to fire, the admin user (represented by `msg.sender`) must satisfy the admin condition,

```
// /* CA-3 */  <role3, t1, role2 & NOT ~ role3, t1, goalRole>
function fireCA3Rule(address _targetUser) public noZeroAddress(_targetUser)
  sender(trbac_state[msg.sender][0][0] == true)
  target(trbac_state[_targetUser][3][0]==true && trbac_state[_targetUser][0][0] == false)
{
  trbac_state[_targetUser][2][0] = true;
}
// /* CE-3 */  <TRUE, t1, role1 & NOT ~ role2, t1, role3>
function fireCE3Rule() public
  role(role_enablement[1][0] == true && role_enablement[3][0] == false)
{
  role_enablement[0][0] = true;
}
```

Figure 6.5: Converting rules for Baked ATRBAC Smart Contracts.

for Can Assign/Can Revoke the target user must satisfy the precondition, and for Can Enable/Can Disable the target role must satisfy the precondition. These constraints are covered with `sender`, `target`, and `role` function modifiers.

6.4.2 Convert Rules

Within a Baked Smart Contract, each converted ATRBAC rule is given its own function. In Figure 6.5, we show the converted Can Assign rule `CA-3` and Can Enable rule `CE-3` from Figure 6.3. The constraints for each rule are converted into function modifier calls. Any constraint that is `TRUE`, will be excluded from the list of constraints. Within the function, the TRBAC or Role Enablement states are directly altered. Can Assign and Can Revoke rules use the TRBAC state for the admin and target user conditions. Can Enable and Can Disable rules use the TRBAC state for the admin conditions and the Role Enablement state for the target role conditions. The above accurately simulates ATRBAC policies.

6.4.3 Convert Safety Query

Performing safety analysis on the Ethereum network would be too expensive, slow, and currently impossible for non-trivial safety queries due to the gas block limit. Thus we convert the safety query into a bug bounty, show in Figure 6.6. Bug bounties have proven very useful for software companies, Google paid $6.5 million USD in bug bounty rewards in 2019 [81]. This bug bounty allows anyone to send Ether to the contract, but only someone who satisfies the safety query can withdraw from the smart contract's balance. This allows the company, or any stakeholder, to put money into the bug bounty. If anyone is able to retrieve the balance, then the list of steps to get to the safety query state is saved in the

```
// Query : t1 , [ goalRole ]
function withdraw(uint256 amount) public {
require(amount <= address(this).balance, "Not Enough Funds");
require(trbac_state[msg.sender][2][0]==true, "User does not satisfy the Safety Query")
    ;
msg.sender.transfer(amount);
}
```

Figure 6.6: Converting the Safety Query into a Bug Bounty for Baked ATRBAC Smart Contracts.

Ethereum blockchain. From this, security administrators can alter their polices to remove the safety vulnerability and reward and security researcher for their efforts and results.

6.5 Costs

Ethereum is an attractive option for companies: account creation is free, fees for transactions are small, reading/viewing data is free, and the fees are directly related to the amount of gas/work executed by the miners. For most small to medium sized companies, the cost to recreate the Ethereum network is too significant to be practical.

In Table 6.1, we outline the costs of converting the ATRBAC policies from Ranise et al. [93]. These ATRBAC policies represent real University ATRBAC policies and thus can provide a good estimation of the cost for deploying a Generic or Baked ATRBAC Smart Contract.

The first column in Table 6.1 is the ATRBAC policy file. The next 3 columns provide the number of unique roles/timeslots/rules in the policy. This indicates how large the TRBAC State and Role Enablement table are. The *Generic Gas* column represents the total cost, in gas, for deploying and setting up the Generic ATRBAC Smart Contract. This includes the steps: deploy contracts and libraries, all transactions to add roles/timeslots, all transactions to create rules (see Appendix G). The *Baked Gas* column represents to deployment costs, in gas, for each Baked ATRBAC Smart Contract.

Once deployed, all access control changes made by the administrators cost gas per transaction. It costs approximately 75 000 gas $\approx$ \$0.31 USD to execute one `fireCanAssignRule` function in the Generic ATRBAC Smart Contract. Functions that have the keyword `view`, like `hasAccess` and `getRoleName` in Figure 6.3, do not cost any gas to run locally. If a `view` function gets called during a transaction, then the statements inside that function do cost gas. To call a can assign rule in a Baked ATRBAC Smart Contract the cost is approximately 45 000 gas $\approx$ \$0.19 USD.

Table 6.1: Costs for Converting Ranise et al. [93] "University" ATRBAC policies to Generic and Baked Smart Contracts. 10×10^6 gas $\approx$ \$41.6 USD, 150×10^6 gas $\approx$ \$624 USD (using 20 gwei/gas).

	Roles	Timeslots	Rules	Generic Gas	Baked Gas
AGTUniv01	30	5	220	10,680,294 gas	42,291,721 gas
AGTUniv02	33	5	224	10,831,662 gas	32,917,668 gas
AGTUniv03	31	11	483	16,766,814 gas	75,101,757 gas
AGTUniv04	33	10	491	17,047,898 gas	76,342,997 gas
AGTUniv05	33	20	542	18,322,046 gas	79,908,514 gas
AGTUniv06	30	20	560	18,751,430 gas	90,026,032 gas
AGTUniv07	32	30	784	23,997,078 gas	147,290,341 gas
AGTUniv08	30	30	802	24,326,650 gas	148,317,445 gas
AGTUniv09	32	40	938	27,732,874 gas	191,105,053 gas
AGTUniv10	31	40	994	28,994,414 gas	150,517,292 gas

From the from the deployment costs in Table 6.1 we see that the Generic ATRBAC Smart Contract costs much less than the Baked ATRBAC Smart Contract. This is because new functions have a high gas cost to create. The higher deployment costs for the Baked ATRBAC Smart Contract reduces the transactional cost. The Generic Smart Contract is able to update over time, at a relatively low cost to add/remove a rule. Whereas a new Baked Smart Contract needs to be deployed for any alteration to the policy. The limited features of the Baked Smart Contract is ideal for well designed ATRBAC policies where expansion is not required. The Generic Smart Contract is ideal for organically written ATRBAC policies, where they are updated as needs arise. This versatility of the Generic ATRBAC Smart Contract is at the cost of a new attack vector.

6.6 EIP 170 - Contract Code Size Limit

Ethereum Improvement Proposals (EIP), are proposals for change to the Ethereum network. Each EIP is given a number, title, and description. EIPs are debated, and those which are implemented are known as Accepted EIPs. The Accepted Ethereum Improvement Proposals (EIP) 170 [23] proposed to limit the maximum smart contract code size. This EIP effects the size of ATRBAC policies which can be baked. EIP 170 was implemented at Block Number 2675000, and it limits the maximum size of smart contract to

24576 bytes. This limit only effects the size during deployment, the smart contract can grow after this. All ATRBAC policies can be converted to the Generic ATRBAC Smart Contract within this limit. ATRBAC policies with more than 400 rules cannot be converted to Baked ATRBAC Smart Contracts.

By splitting the Baked functions into new smart contracts and utilizing delegate calls, we can convert any ATRBAC policy into a Baked ATRBAC Smart Contract. EIP 2535 [74], the Diamond Standard, provides a method of splitting contracts and utilizing delegate calls.

EIP 170 has found critism within the Ethereum community and might not stay implemented. EIP 1662 [44], called "Removing Contract Size Limit", proposes removing the maximum deployment contract size which EIP 170 implemented.

In Section 6.5, EIP 170 prevented gas estimates for the baked smart contracts. We changed the version of the solidity compiler to v0.4.24, which was created before EIP 170 was implemented. This change required running Ganache [43] (locally hosted Ethereum Blockchain) with the `-allowUnlimitedContractSize` flag and set the block gas limit (`-gasLimit`) equal to `0x1FFFFFFFFFFFFF`.

6.7 Conclusions

In this chapter we have created two methods for automatically creating ATRBAC Smart Contracts. The Generic ATRBAC Smart Contract needs no modification before deploying it to the Ethereum network. Once deployed it can be updated with new roles/timeslots and rules. The list of super users can be expanded to include more people and accounts can be removed from super user access. Setting up the ATRBAC policy requires transactions for creating roles/timeslots and for each rule. The second method Bakes the ATRBAC policy into a smart contract. This setup occurs locally, and only 1 transaction is required to deploy the ATRBAC Smart Contract.

The deployment costs are much higher Baked ATRBAC Smart Contracts. The costs are higher to fire rules in the Generic Smart Contract. Thus it is possible over the life time of an ATRBAC policy that the costs of Generic can exceed the costs of the Baked smart contract due to frequent transactions.

Due to the accepted EIP 170 [23], the maximum size of a Baked ATRBAC Smart Contract is limited to 24576 bytes. There are methods to take a large smart contract and, using delegate calls, to create multiple smart contracts which are all below EIP 170's

threshold. One process to perform this is outlined in EIP 2535 [74]. If EIP 1662 [44] is accepted, the limit imposed by EIP 170 will be removed.

If customizability is required in the ATRBAC policy, then the Generic method is better. If immutability is required, then the Baked method would be preferred. The Generic method can be made immutable by removing all super users. If reduced transaction costs are preferred then the Baked method is preferred. Generic ATRBAC Smart Contract have much smaller deployment costs.

Part II

Problems Reduced to Integer Linear Programming

Chapter 7

Vagabond: Using VM Scheduling to Combat Side-Channels in Cloud Systems

Contents

Declaration of Contributions

This chapter is based off the paper [60], written by: Nahid Juma, Jonathan Shahen (thesis author), Khalid Bijon, and Mahesh Tripunitara. The contributions included in this chapter are my contributions to that paper. All sections, except for Section 7.5, are copied from [60], with modifications to fit within this thesis. Section 7.5 was created after [60] and solely my work.

7.1 Introduction

Cloud computing has become an important paradigm by offering flexibility and cost-effectiveness to both providers of infrastructure and services, and their clients. An aspect of cloud computing is virtualization, which enables providers to host virtual machines (VMs) of multiple clients on the same physical infrastructure. Multiple VMs may reside on the same server, which we call co-resident. While co-residency has benefits, such as economies of scale, it gives rise to a serious security threat. Information may leak from a victim client's VM to a malicious co-resident VM via a side-channel attack. In such an attack, the attacker runs a VM on the same server as the victim's VM and takes advantage of a shared physical component in order to extract information about the victim. For example, the attacker may retrieve the victim's cryptographic key by observing the activity of the processor cache. Prior work exposes several such co-residency side-channels in shared cloud environments [54, 55, 71, 80, 96, 128, 148, 155, 156].

A straightforward solution to this security issue is hard-isolation: preclude co-residency and give each client dedicated hardware. This is unfavourable because it reduces efficient use of resources. Several other defences that do not involve the complete elimination of co-residency have been suggested by prior work [136, 87, 84]. Many of these defences suffer from at least one of the following drawbacks. Firstly, they entail changes to existing deployments and applications (see Section 7.1.1 and [73] for a discussion). Secondly, they are catered specifically to known cross-VM side-channel attacks. Therefore, as new attacker capabilities are unveiled, more changes may have to be made. It is desirable to have defences that are general across a broad spectrum of side-channel attacks and immediately deployable with no or only few modifications to existing cloud hardware and software.

With this objective in mind, scheduler-based defences have been proposed in recent work by Moon et al. [73]. These can be thought of as a realization of the moving target defence philosophy to mitigate side-channels [35]. The mindset is that security is a factor which should be considered when deciding how VMs are placed on servers, rather than as an after-thought.

A scheduler has several mechanisms it can employ to maximize security. Two that have been explored in recent prior work by Moon et al. [73] are the following. A scheduler can be careful with where it places a newly arrived VM thereby limiting information leakage from or to it. Migration of existing VMs is yet another mechanism, with the intent that a victim VM does not leak too much information to a co-resident malicious VM. In Figure 7.1, we illustrate a scheduler-based defence. In the figure, a scheduler places and migrates client VMs in a manner that minimizes information leakage between them.

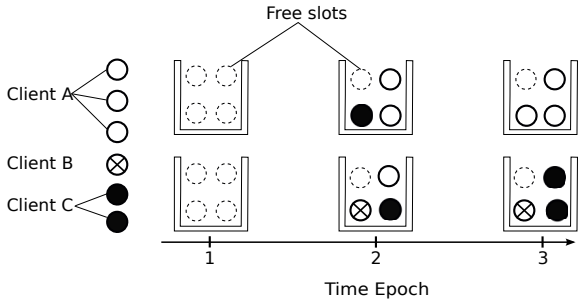


Figure 7.1: Scheduling actions that can reduce information leakage. Time is discretized into periods called *epochs*. A scheduler places VMs, such as the ones that arrive in Epoch 1 and are placed in Epoch 2. It also migrates VMs, which results in a new placement for Epoch 3. We use this example in several sections of the chapter. In particular, the placement in Epoch 2 may or may not be considered to suffer from more information leakage than the one in Epoch 3, depending on the model for leakage that we adopt.

Scheduler-based defences appear to have promise as they offer several advantages. Firstly, they focus on the root cause of side channels, i.e. co-residency, and are agnostic to the specific side-channel attack used. This makes them robust against unforeseen side-channels that meet certain conditions [73]. Secondly, they require no changes to the cloud provider's hardware, client applications, and hypervisors and can be deployed "out of the box" as they require only changing the VM placement and/or scheduling algorithm deployed by the cloud provider. While we acknowledge that there may be other defences, e.g. shielded execution [10], that also offer these advantages to some extent, our focus in this work is on scheduler-based defences.

An important contribution of Moon et al. [73] is four models that quantify information leakage in co-resident settings. These allow for the security-sensitive scheduling problem to be posed precisely as an optimization problem — given a migration budget, we seek a placement of VMs on servers that minimizes information leakage. That work explores three approaches to solve the problem, namely a mapping to ILP, an initial greedy algorithm called Baseline Greedy, and a final greedy algorithm whose associated software is called Nomad.

7.1.1 Related Work

Cross-VM side-channel attacks Ristenpart et al. [96] were among the first to show how an attacker can map the internal cloud infrastructure using network probing in order

to achieve co-residency with a victim VM, and how the co-residency can be exploited to extract coarse-grained information such as keystroke patterns.

This was followed up by the work of Zhang et al. [155] who showed how a Prime+Probe side-channel attack on upper level cache can be used to extract more fine-grained information such as an ElGamal decryption key. In the prime stage, the attacker fills a portion of cache and then waits to allow the victim to access cache. In the probe stage, the attacker reloads the primed data and uses the probe time to decide whether the victim accessed that portion of cache. Prime+Probe attacks on last level caches that make use of huge page-sizes have also been studied [71, 54, 52].

Research has also shown how fine-grain information can be extracted using Flush+Reload techniques [156, 55, 148]. In the flush stage, the attacker flushes out certain lines in memory and waits to allow the victim to access them. In the reload stage, the attacker accesses the flushed lines. If any such line does not take long to load, the attacker knows the line was accessed by the victim.

Other works have shown that sharing of same-content memory pages among co-resident clients poses the threat of a memory disclosure attack [80, 128]. The attacker can identify which pages it shares with a victim and take advantage of differences in write times to gain information about the victim's applications and OS.

Of course, other kinds of side-channels may exist. For example, the work of Xu et al. [145] considers the issue of excluding the operating system from the trusted computing base, and side-channels that can result as a consequence. As another example, the work of Shinde et al. [122] considers side-channels based on page-faults. Such attacks can certainly underlie cross-VM side-channel attacks. For example, if a benign VM runs a particular kind of application that allows a malicious VM that is co-resident to cause a page-fault, it may be vulnerable to the latter kind of attack.

Defences To counter such attacks, several defence strategies have been proposed. At a hypervisor level, Vattikonda et al. [136] proposed hiding a program's execution time by eliminating fine grained timers. Kim et al. [63] used statistical multiplexing to protect against cache-based side-channel attacks in the cloud. They proposed a set of locked cache lines per core that are efficiently multiplexed amongst co-resident VMs and never evicted from the cache. Raj et al. [87] identified Last Level Cache (LLC) sharing as one of the impediments to finer grain isolation, and proposed two resource management approaches to provide performance and security isolation - cache hierarchy aware core assignment and page colouring based cache partitioning. Shi et al. [121] proposed an approach that leverages dynamic cache colouring: when an application is doing security-sensitive operations, the Virtual Machine Manager (VMM) is notified to swap the associated data to a safe and

isolated cache line. Varadarajan et al. [135] suggested introducing a minimum run time (MRT) guarantee into the Xen scheduler, to limit the frequency of preemptions.

At a guest OS level, Zhang et al. [157] proposed a method by which a tenant can construct its VMs to automatically inject additional noise into the timings that an attacker might observe from caches. Pattuk et al. [84] suggested partitioning cryptographic keys into random shares and storing each share in a different VM. Zhou et al. [162] presented a software approach which dynamically manages physical memory pages shared between security domains to disable sharing of LLC lines.

At a hardware level, some works have proposed modifying the cache architecture to obtain access randomization [70, 140] while others have explored the partitioning of cache [82]. In recent work, Liu et al. [69] demonstrated how to combat LLC side-channel attacks by using the Intel Cache Allocation Technology (CAT) to partition the LLC into a hybrid hardware-software managed cache.

As many side-channel attacks are based on the observances of memory access, a promising defence strategy is to use Oblivious RAM (ORAM) to randomize data access patterns [42, 125]. When an adversary observes the physical storage locations accessed, the ORAM algorithm ensures that they have negligible probability of learning the true access pattern.

An approach that focuses on isolation is that of shielded execution, for example, by using Intel SGX enclaves [10]. Chen et al. [25] presented a software framework that enables a shielded execution to detect page-fault side-channel attacks.

Various scheduler-based defences have also been explored. Y. Zhang et al. [158] proposed periodically migrating VMs using methods based on game theory. Azar et al. [9] focused on mitigating co-location attacks, on the premise that they are a necessary first step to performing cross-VM attacks. They suggested assigning VMs to servers such that attack VMs are rarely co-located with target VMs. Their model is based on co-location rather than on information leakage. Han et al. [47] also studied how to minimize the attacker's possibility of co-locating their VMs with targets. They introduced a security game model to compare different VM allocation policies and advocated selecting an allocation policy statistically from a pool instead of having a fixed one. Bijon et al. [18] proposed an attribute-based constraints specification framework that enables clients to express essential properties of their resources as attributes. A constraints enforcement engine then schedules the VMs while respecting the conflicts specified by these attributes. Han et al. [48] proposed metrics which can be used to assess the security of a VM allocation policy.

The work that is most closely related to ours is by Moon et al. [73]. They characterized information leakage models which can be used to pose the security-sensitive scheduling problem precisely and explore various approaches to address the problem. Our work differs

from theirs in the following ways: (1) We analyze the problem's computational complexity (in [60]), (2) Our reduction to ILP is polynomial time, while theirs can be exponential time for a certain encoding of input, (3) We also investigate the approach of reducing to CNF-SAT and employing a SAT solver (in [60]), (4) Our approaches guarantee security whereas their final approach, Nomad, does not, and (5) We show a new attack which occurs in non-zero information leakage systems.

7.1.2 Summary of Conference Paper Contributions

This chapter is based off the work done in Juma et al. [60]. Here we will summarize the important contributions that paper made that are not part of this chapter.

1. We show that a decision version of the problem, to which the optimization version reduces polynomially, is in **NP**. We show also that the problem is indeed **NP**-hard and is therefore **NP**-complete.
2. We then establish that even the special case of the problem for closed systems, under the $\langle R, C \rangle$ model for information-leakage, is **NP**-hard.
3. Our second set of contributions is an analysis of the three approaches taken by Moon et al. to address security-sensitive scheduling [73]. We analyze their mapping and observe that while it does appear to be a reduction, it can be inefficient – there exists an encoding of placement for which the output of the mapping from that work is exponential in the size of its input.
4. A reduction from this problem to CNF-SAT was created and included as part of the testing.

7.1.3 Contributions

We reduce the problem to ILP with the intent of adopting an ILP solver as an oracle. Our reduction to ILP differs from that of prior work in that ours is computable in polynomial-time, and, as a consequence, the size of its output is polynomial in the size of the input.

We have implemented our ILP approach in this chapter and the CNF SAT approach in [60]. Also, we have conducted an empirical assessment of our approaches, and Nomad. We have validated some of our analytical observations on Nomad against its implementation. We implemented both reductions due, from practice, that SAT/ILP solvers may not demonstrate the same performance on all input instances. Thus, even though both

CNF-SAT and ILP are **NP**-complete, it is interesting to investigate whether one of those approaches outperforms the other for different classes of input instances.

Our ILP implementation is discussed in Section 7.3, and we make empirical observations in Section 7.4. Our empirical observations are consistent with the main themes of this work. We find that while approaches based on reduction to ILP and CNF-SAT do not scale as well as Nomad, their scalability appears to be much better than previously reported [73] — for example, for a cluster of 40 servers, an approach based on reduction to ILP takes 4 minutes, and not longer than a day as previously reported (see Section 7.4).

In Section 7.5, we show a new attack that occurs with the previous reductions and Nomad, in non-zero information systems and using Total Information Leakage. We propose an alternative reduction to ILP to solve this attack. We perform an empirical analysis on the 2 ILP reductions within this attack case in Section 7.5.1.

7.2 Minimizing Information Leakage

In this section, we describe the problem posed by Moon et al. [73]. The problem is: given (i) a set of clients, each having a number of VMs, (ii) a set of servers, each having a capacity expressed as a number of VMs, and (iii) a migration budget, that limits the number of VM migrations that can occur, what is a placement of VMs on servers that minimizes information leakage? Our exposition is similar to that of Moon et al. [73], with changes for clarity only.

Setup Time is discretized into periods called *epochs*. A *VM* is a virtual machine, which is containerized software, similar to a process in a conventional operating system. A VM's execution may last several epochs. A *client* is associated with a set of VMs; every VM belongs to a client. The set of VMs that belong to a client can change over time. A *server* is a machine on which a VM runs; the VM is said to be *hosted* by that server. A server is able to host a maximum number of VMs at any given time, this is called the *capacity*. VMs may be moved and hosted by a different server in a later epoch, this is called *migration*. VM migrations occur seamlessly, meaning the VM is not aware of the migration. A potential victim client has some secret information in her VMs. A malicious client attempts to steal the secret of a victim client via information-leakage across VMs that can occur when the VMs are *co-resident*, i.e., hosted by the same server. We do not know beforehand which VMs are potential victims, and which are malicious. In the leakage models we consider, clients do not share secrets, and a client's secret may be present in more than one of its VMs. Figure 7.1 shows three servers and clients, and six VMs across those three clients,

spanning three epochs. These notions are further explained by Table 7.1, which presents the symbols we use in our calculations of information-leakage.

We quantify information as a certain number of bits. We premise that merely owing to co-residency of a victim and a malicious VM, information leaks from the former to the latter. Information that leaks from a VM to another aggregates over the time that the two VMs are co-resident. In order to aggregate information leakage as a function of co-residency over time, we consider a *sliding window* of the most recent $\triangle$ epochs. This can model secrets, such as cryptographic keys, that are refreshed periodically; information the adversary gathered in previous sliding windows is no longer useful to her. We assume that during an epoch, a VM leaks 1 bit to another that is co-resident. This can be generalized to the case that the amount of leakage is some constant, k bits. The value k can then be used to model the time-length of an epoch. The amount of information leaked from a victim to an adversary is proportional to the time their VMs are co-resident with one another. For example, if a VM of Client A is co-resident with a VM of Client B for 2 epochs, then the total information leaked from Client A's VM to Client B's VM over 2 epochs is 2 bits.

Replication and Collusion *Replication* means that information is duplicated across a victim client's VMs. This is potentially advantageous for an attacker because when an attacker VM is co-resident with multiple victim VMs, under the assumption that the attacker VM extracts different bits of information from each victim VM, the rate of leakage to the attacker increases. For example, in Figure 7.1, for the placement in Epoch 2, with replication, Client C (black), on server 1, gains twice as many bits from Client A (white) than it would without replication.

Collusion means that a malicious client's VMs can collaborate with one another. This is potentially advantageous for the attacker because when multiple attacker VMs are co-resident with a victim VM, under the assumption that each attacker VM extracts different bits from the victim VM, the rate of leakage to the attacker, as a whole, increases. In Figure 7.1, for the placement in Epoch 2, with collusion, Client A (white), on server 1, gains twice as much information from Client C (black) than it would without. As in prior work, we denote the presence of replication and collusion by R and C respectively, and the absence by NR and NC respectively. This leads to four possible leakage models: $\langle NR, NC\rangle$, $\langle R, NC\rangle$, $\langle NR, C\rangle$, $\langle R, C\rangle$.

Information Leakage Models We now describe how client-to-client leakage is calculated under the four models, using the example in Figure 7.2.

For the $\langle NR, NC\rangle$ case, the information leakage from client c to client c' is the maximum per-VM-pair information leakage across all pairs of VMs. This is the information which

Table 7.1: Symbols used for calculating Total Information Leakage.

Symbol	Meaning	Symbol	Meaning
$v_{c,i}$	The i^{th} VM of client c.	X, x_i	$X = \{x_1, x_2, \ldots, x_n\}$ is a set of integers where x_i denotes the number of VMs that belong to the i^{th} client.
$Y_{c,i,c',i'}(e')$	Indicator variable $\{0, 1\}$. It takes the value of 1 if $v_{c,i}$ and $v_{c',i'}$ are co-resident on a server during an epoch e'. Otherwise 0.	S, s_j	$S = \{s_1, s_2, \ldots, s_k\}$ is a set of integers where s_j denotes the number of VMs the j^{th} server can host.
$I_{c,i,c',i'}(e, \triangle)$	The VM-to-VM information leakage from $v_{c,i}$ to $v_{c',i'}$ over the sliding window of epochs $[e - \triangle, e]$. This quantity is calculated as $\sum_{e'=e-\triangle}^{e} Y_{c,i,c',i'}(e')$	A, a_i	$A = \{a_1, a_2, \ldots, a_n\}$ is a set of integers where a_i denotes the number of VMs belonging to the i^{th} client which have arrived in this epoch.
$I_{c,c'}(e, \triangle)$	The client-to-client information leakage from c to c' over the sliding window of epochs $[e - \triangle, e]$. This quantity is dependent on the presence (R, C) or absence (NR, NC) of replication and collaboration.	F, f_e	$F = \{f_{e-1}, f_{e-2}, \ldots, f_{e-\triangle}\}$, where each f_e encodes the placement of VMs in servers in the e^{th} epoch. Entries in F can be used to express VMs that depart.
$I_{total}(e, \triangle)$	The total information leakage over the sliding window of epochs $[e - \triangle, e]$, under one of the four leakage models, $\{R, NR\} \times \{C, NC\}$. This quantity is calculated by aggregating the client-to-client leakage across all pairs of clients.	m	An integer that is the migration budget. Each kind of migration (e.g., swap VMs on two servers) has a cost associated with it. A special symbol, ∞, for m, is used to indicate that we are to be unconstrained by a migration budget.

leaks to the adversary VM that is co-resident with the same victim VM for the largest number of epochs:

$$\langle NR, NC\rangle: \qquad I_{c,c'}(e,\triangle) = \max_{i} \max_{i'} I_{c,i,c',i'}(e,\triangle)$$

In Figure 7.2, the maximum VM-to-VM leakage from the blue client (B) to the red client (R) is 2 bits, so the leakage from B to R under $\langle NR, NC\rangle$ is 2 bits over the three epochs.

For the $\langle R, NC\rangle$ case, there is a cumulative effect across victim VMs. The information leakage from client c to client c' is determined by the adversary VM that has extracted the most information:

$$\langle R, NC\rangle: \qquad I_{c,c'}(e,\triangle) = \max_{i'} \left[\sum_{i} I_{c,i,c',i'}(e,\triangle) \right]$$

In Figure 7.2, R1 obtains the most information, a total of 1+2=3 bits, so the leakage from B to R under $\langle R, NC\rangle$ is 3 bits.

For the $\langle NR, C\rangle$ case, there is a cumulative effect across adversary VMs. The information leakage from client c to client c' is determined by the victim VM that has leaked the most information:

$$\langle NR, C\rangle: \qquad I_{c,c'}(e,\triangle) = \max_{i} \left[\sum_{i'} I_{c,i,c',i'}(e,\triangle) \right]$$

In Figure 7.2, B1 and B2 each leak a total of 1+2=3 bits, so the leakage from B to R under $\langle NR, C\rangle$ is 3 bits.

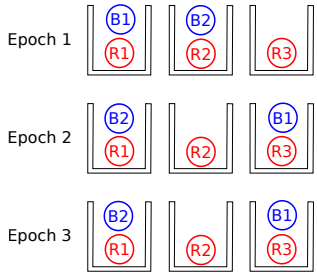
Finally for the $\langle R, C\rangle$ case, there are cumulative effects across both victim and adversary VMs:

$$\langle R, C\rangle: \qquad I_{c,c'}(e,\triangle) = \sum_{i} \sum_{i'} I_{c,i,c',i'}(e,\triangle)$$

In Figure 7.2, the sum of all bits leaked from B's VMs to R's VMs is 6, so the leakage from B to R under $\langle R, C\rangle$ is 6 bits.

Given the client-to-client information leakage in any one of the four models, the total information leakage that we seek to minimize is:

$$I_{total}(e,\triangle) = \sum_{c} \sum_{c'} I_{c,c'}(e,\triangle)$$



VM Pair	Leakage (bits)
B1 → R1	1
B1 → R2	0
B1 → R3	2
B2 → R1	2
B2 → R2	1
B2 → R3	0

Figure 7.2: Example showing how client-to-client leakage is calculated under all four models. The leakage from B1 to R1 is 1 bit over the three epochs because B1 is co-resident with R1 for one out of the three epochs.

Minimizing $I_{total}(e, \triangle)$ is equivalent to minimizing the average leakage across client pairs. We acknowledge that there may be other ways to represent the security of the system, for example, by minimizing the inter-client leakage. Although we chose the above representation for ease of comparison with prior work [73], the approaches we propose in Section 7.3 are general and can be used for such representations of security by modifying the objective function.

Problem Statement We now pose the scheduling problem that is solved at the beginning of each epoch with the intent of minimizing information-leakage. We first choose and fix a particular information leakage model from amongst $\{R, NR\} \times \{C, NC\}$.

Inputs: The sets X, S, A, F, and an integer, m — see Table 7.1.

Output: A placement for the current epoch that corresponds to the minimum total information leakage subject to the constraints specified as input.

As an example, suppose that the input is as shown for Epoch 2 in Figure 7.1. Assume the leakage model is $\langle R, C \rangle$, and $m = 2$. Also assume that Client A corresponds to Client 1, Client B to Client 2 and so on. Then, the inputs are: $X = \{3, 1, 2\}$, $S = \{4, 4\}$, $A = \{0, 0, 0\}$ and a function $f_2(i, j)$ which indicates the number of VMs of client i on server j during Epoch 2, e.g. $f_2(1, 1) = 2$. The output is the placement shown for Epoch 3 in Figure 7.1. In this example, the total information leakage in Epoch 2 is 10 bits, and in Epoch 3 it is 14 bits (10+4). That is, migrating some VMs causes the increase in information-leakage to be 4 only, rather than 10, which would have been the case had we adopted the same placement for Epoch 3 as we have for Epoch 2.

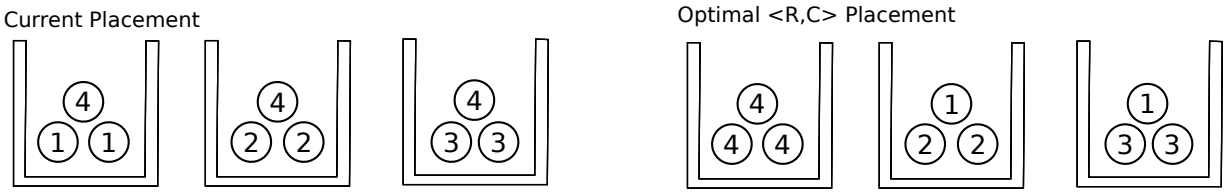


Figure 7.3: An example to show the sub-optimality of Nomad for the $\langle R, C \rangle$ model. The capacity of each server is at least 3 VMs. Clients 1, 2 and 3 have two VMs each, and Client 4 has three VMs. The placement in the left figure is the current placement. There is no free-insert or 2-way swap that results in a reduction in leakage. Hence Nomad makes no moves even when unconstrained by a migration budget. The optimal placement is shown in the right figure.

7.3 Reductions to ILP

In [60], we show that Nomad provides no decrease in information leakage for infinitely many inputs. An example placement map is shown in Figure 7.3. Thus we revert to our two reduction approaches. One is an efficient reduction to ILP (discussed here) and the other is an efficient reduction to CNF-SAT (discussed in [60]). Our primary intent is to identify whether we can indeed realize software that is efficient in practice, and provides the security guarantee of minimizing information leakage, notwithstanding the **NP**-hard worst-case for computational complexity. Both approaches are efficient algorithms given access to oracles for ILP and CNF-SAT, respectively. We chose to invest in both approaches because there can be differences in the performance of individual solvers for different classes of problem instances.

Reduction to ILP Figure 7.4 is our reduction to ILP for the $\langle R, C \rangle$ case. The other cases require minor modifications only, as we discuss in the following. The input to the reduction is an instance of the problem posed in Section 7.2, and the output is the optimal placement map. The unknown in the constraints and the optimization objective, is a new placement — values for the $n \times k$ variables $f_t(i, j)$, for each $i \in [1, n], j \in [1, k]$, for a given t, for the $\langle R, C \rangle$ case. These values are the number of VMs of client i on server j in the upcoming epoch, t. For the other three models of information leakage, the unknowns are $f_t(v, i, j)$, where $v \in [1, x_i]$, which indicates whether VM v of client i is placed on server j in epoch t.

In Figure 7.4, Line (7.1) provides the precomputed past total information leakage for

$$\textbf{input: } \Omega = \sum_{t'=t-\triangle}^{t-1} \sum_{c=1}^{n-1} \sum_{c'=c+1}^{n} \sum_{j=1}^{k} F[t'][c][j] \times F[t'][c'][j] \tag{7.1}$$

$$\textbf{minimize } \Omega + \sum_{j=1}^{k} \sum_{c=1}^{n-1} F[t][c][j] \times \left(\sum_{c'=c+1}^{n} F[t][c'][j] \right) \tag{7.2}$$

subject to

$$2 \times m \geq \sum_{c=1}^{n} \sum_{j=1}^{k} |F[t][c][j] - F[t-1][c][j]| \tag{7.3}$$

$$s_j \geq \sum_{c=1}^{n} F[t][c][j] \qquad \forall\, j \in [i,k] \tag{7.4}$$

$$x_i = \sum_{j=1}^{k} F[t][c][j] \qquad \forall\, c \in [1,n] \tag{7.5}$$

$$0 \leq F[t][c][j] \qquad \forall\, t,c,j \tag{7.6}$$

Figure 7.4: Reduction to ILP for the $\langle R, C \rangle$ model for total information leakage. Symbols are explained in Table 7.1. Constraint (1) ensures that the migration budget, m, is respected. Constraint (2) ensures that the capacity of a server, s_i, is not exceeded. Constraint (3) ensures that every VM of each client is placed on exactly one server. Constraint (4) ensures only positive numbers are allowed. The objective function minimizes the total information leakage. The source code for this reduction in CPLEX OPL is attach in Appendix I.

the $\langle R, C \rangle$ model. This is an optimization step to reduce overhead from the ILP model. The objective function, Line (7.2), ensures that we minimize the total information. Line (7.3) limits the number of migrations the current placement map is able to make. If m is specified as ∞ in the input, then this line is omitted. Line (7.4) is the constraint to ensure that the capacity of each server, s_j, is not exceeded in the new placement. Line (7.5) is the constraint to ensure that every VM of each client is placed on exactly one server. Line (7.6) ensures that the placement is formatted correctly, i.e. positive integers.

The only nuance here is that the objective function, for the $\langle R, C \rangle$ case, is not a set of linear constraints, but rather quadratic constraints. However, Quadratic Constraint Programming is known to be in **NP** as well [137]. We have designed and implemented a

reduction from quadratic to linear constraints via bit-wise multiplication.

Figure 7.4 runs in polynomial-time; its running-time is: $O\left(\binom{n}{2} \cdot \triangle \cdot k \cdot \log_2^2 \max_i x_i\right)$, where n is the number of clients, $\triangle$ is the number of epochs in the sliding-window, k is the number of servers, and $\log_2 \max_i x_i$ is the maximum number of bits we need to express the number of VMs per client. This bound is not tight; more efficient reductions can exist from, for example, more compact encodings for placement.

7.4 Empirical Assessment

We have conducted a limited empirical assessment of our two approaches, ILP and CNF-SAT, and Nomad. The intent of our empirical assessment is: (a) to understand the overhead of actually providing a security guarantee of minimizing leakage in practice, and, (b) validate empirically our analytical insights on the shortcomings of Nomad.

7.4.1 Tests

We designed and ran the following tests.

Test 1 A test for scalability. We report the response-time vs. cluster-sizes for the two approaches from the previous section, and Nomad.

Test 2 Demonstrate an example for which Nomad provides no decrease in information-leakage. We know that several such inputs exist, we demonstrate one in practice.

Test 3 Random input-cases that have a known, non-zero value for optimum information-leakage. The intent is to check whether each of the three approaches indeed converges to the optimum. This test is different from Test 2, in that we have randomly picked inputs, with no *a priori* knowledge that Nomad will provide no benefit to information leakage. Our inputs have a non-zero optimum because we anticipate that such inputs are more challenging for an approach.

Test 4 A test for how fast, in terms of epochs, each approach reduces total information leakage given limiting migration budgets. We test with random inputs, with three migration budgets, and measure the incremental improvements to the information leakage.

Test 5 Another test for scalability. Number of clients, servers and VM-slots per server, vs. response-time for 1 epoch. We choose the inputs such that a client has several VMs,

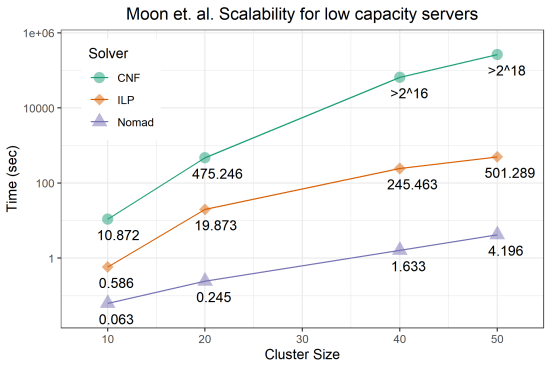


Figure 7.5: Scalability of Nomad, ILP, and CNF-SAT using the testing parameters presented in Moon et al. [73]. Cluster size of x means: x servers with capacity 4, and x clients with 2 VMs each. Total VMs in the cluster: $\sum VM = 2 \cdot x$.

and server-utilization is 50% only. We anticipate that this models reality in practice; cloud-servers are shown to be under-utilized [75].

Setup All tests were performed on a desktop with a 3.60 GHz Intel(R) Core(TM) i7-4790 CPU, has 24 GB RAM, and runs the 64-bit Windows 8.1 operating system. The code for the reductions is written in Java. Our ILP-solver was CPLEX, commercial version 12.6.1.0 [51], the same as the one used in prior work [73]. CPLEX provides a Java API, which we used. Our SAT-solver was Lingeling [2]. Lingeling does not have native binaries for Windows; we used Cygwin [1]. We modified Nomad to allow for arbitrary inputs, and for retrieval of the placement at each epoch. The performance of Nomad is not impacted by these modifications. Nomad is written in C++.

Design Choices We made the following design choices:

1. We ran tests for the $\langle R, C \rangle$ leakage model only. The $\langle R, C \rangle$ model has the strongest attacker. Also the implementation of Nomad we were provided works for $\langle R, C \rangle$ inputs only.
2. We ran tests for the closed migration version of the problem only. The reason is that the implementation of Nomad we were provided works for closed-migration only.
3. The sliding window for was fixed at 5 epochs (see Section 7.2). The reason is that the implementation of Nomad we were provided has the sliding window hard-coded at 5.

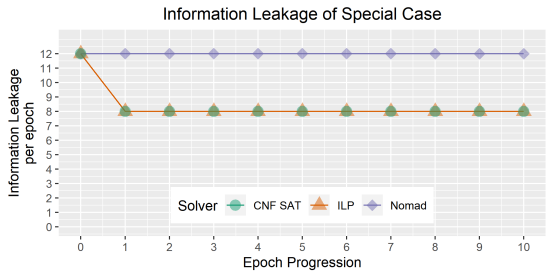


Figure 7.6: Information leakage for all three approaches when run for 10 epochs on the placement map shown in Figure 7.3. The CNF-SAT and ILP solvers were both able to obtain the optimal placement, while Nomad was unable to find any migrations that yield lower total information leakage.

7.4.2 Results

We now present and discuss our results.

Test 1 We show the results of this test in Figure 7.5, which is recreated from the work of Moon et al. [73]. The input instances comprised $x \in \{10, 20, 40, 50\}$ servers, each with a capacity of 4 VMs, and x clients each with 2 VMs. The initial placement is random. The optimal placement is for every client to be placed on a server of its own. Figure 7.5 shows the average time to compute the placement per epoch. We find that Nomad performs well, as reported in prior work [73].

An interesting observation is a difference in our results, and the results reported by Moon et al. [73], on the performance of the approach based on reduction to ILP. Moon et al. [73] report that their reduction to ILP takes longer than a day for cluster sizes of 40 and 50. In contrast, as we report in Figure 7.5, our ILP approach takes about 4 and 8 minutes, respectively. We suspect that this is a consequence of their inefficient (exponential-time) reduction to ILP.

Test 2 We show the results in Figure 7.6. In this test, we ran the example from Figure 7.3. We establish analytically in [60] that Nomad provides no decrease in information-leakage for such an input, even with an unlimited migration budget. Our empirical observations concur. The approaches based on reduction to CNF-SAT and ILP immediately converge to the optimum.

Test 3 We show the results for this test in Figure 7.7. Our input instance comprised 5 servers, each with a capacity of 4 VMs, and 6 clients, each with 3 VMs. We ran the test

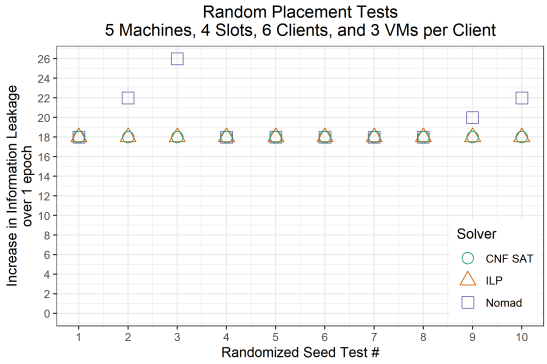


Figure 7.7: Total Information leakage for 1 epoch and 10 random initial placements. Input parameters: 5 servers with 4 VM slots each, and 6 clients with 3 VMs each. Each of the 10 tests were run over a duration of 1 epoch with an infinite migration budget. The optimal placement is the one resulting in leakage of 18 bits. ILP and CNF-SAT always converged to the optimal. Nomad's performance was inconsistent.

over 1 epoch for 10 different random initial placements. The optimal placement results in a total information leakage of 18 bits. We see that CNF-SAT and ILP were always able to converge to an optimal placement, whereas Nomad yields the optimum sometimes only.

Test 4 The results are shown in Figure 7.8. The input instance comprised 6 servers each with a capacity of 6 VMs, and 3 clients each having 6 VMs. We used a random initial placement, and provided three different migration budgets: 2, 5, and ∞. The results in Figure 7.8 show that the ILP and CNF-SAT solvers reached the optimal placement faster than Nomad. This is because Nomad makes locally optimal greedy choices; the other two approaches globally optimize for information leakage.

Test 5 In Figure 7.9, the input instances comprise $x \in [5, 20]$ servers, each with a capacity of x VMs. We had x clients each with $\lfloor x/2 \rfloor$ VMs. The number of total VMs range from 10 to 200. We adopted a time-out of 4 minutes per test. For the initial placement, we tried (i) random, where VMs are assigned randomly to a server that can accommodate it, and, (ii) spread, where each VM of a client is placed on a different server. That is, under the spread-placement, each client had its $\lfloor x/2 \rfloor$ VMs spread out on the first $\lfloor x/2 \rfloor$ servers; the other servers are empty. Both random- and spread-placements are used in practice [31]. Figure 7.9 shows that the ILP implementation scaled the best. This test demonstrates that an approach based on reduction to ILP can perform well; in some cases, even better than a polynomial-time algorithm such as Nomad.

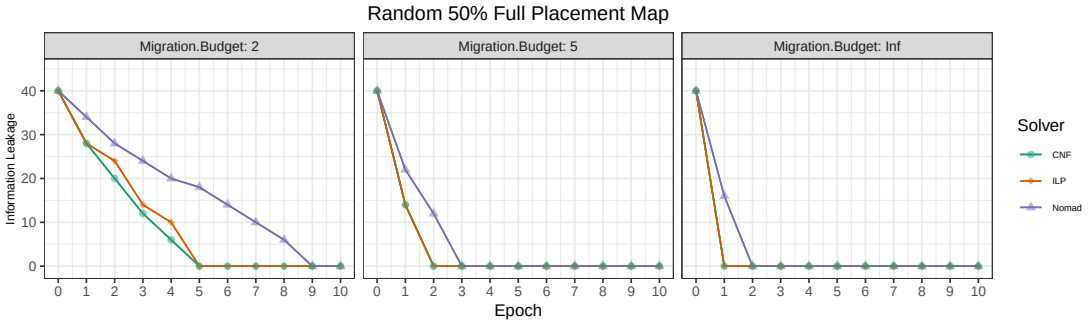


Figure 7.8: How the per-epoch information leakage varied over 10 epochs for all three approaches with a random initial placement and different migration budgets. The instance comprised 6 servers each having a capacity of 6 VMs, and 3 clients each having 6 VMs.

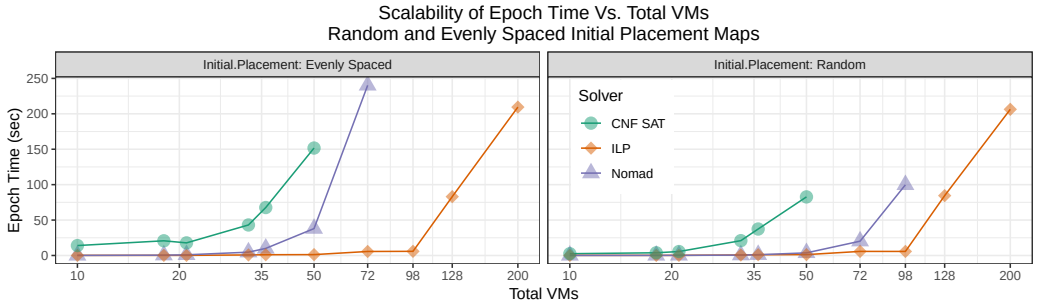


Figure 7.9: The average time it took to compute a placement for 1 epoch. The input instance comprised of x servers each having a capacity of x VMs, and x clients each having $\lfloor x/2 \rfloor$ VMs. The test was run for both random and even initial placements. Tests were not included if they did not complete in under 240 seconds (4 minutes).

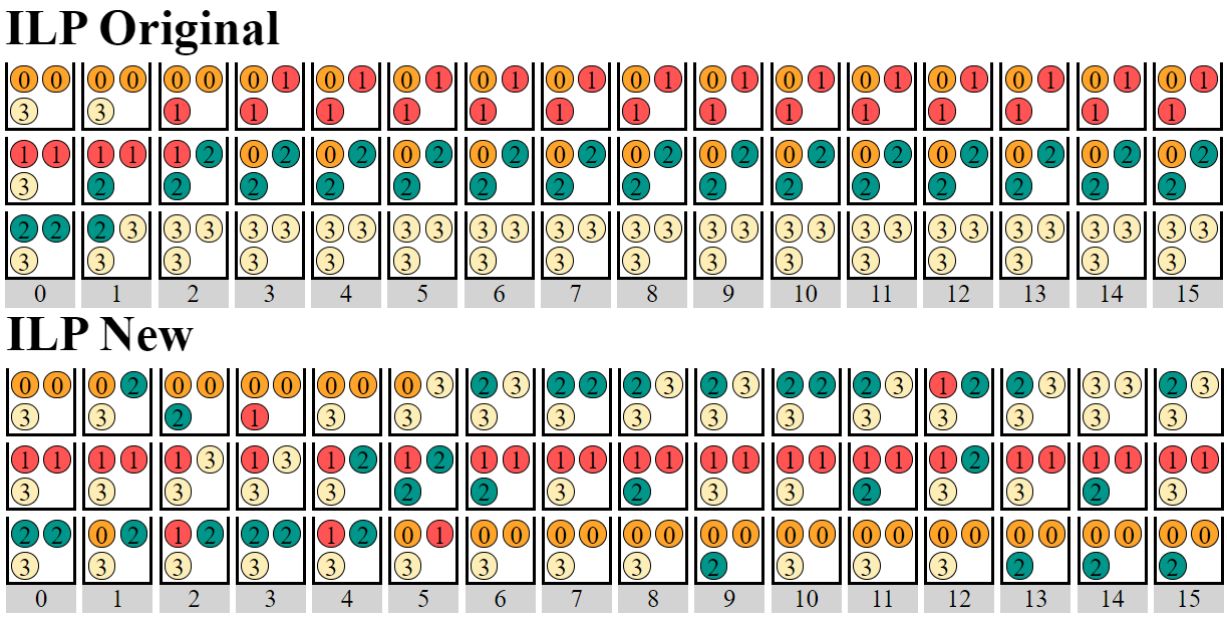


Figure 7.10: Comparison of the placement maps for the Original ILP method and the New ILP method. The original method quickly reduces to the lowest total information leakage state, but then remains there, whereas the new method never stays in one state.

7.5 Minimum Total Information Leakage Attack

Attackers can often create non-standard conditions to test the original programmer's assumptions to see if a vulnerability can be found in an untested region. If we assume an attacker is able to fill up a cloud computing cluster/region, then we explore the situation of whether the placement methods contain any flaws which would allow an attacker to operate their side-channel attacks.

In Figure 7.10, we outline a simple experiment where an attacker can take advantage of minimizing total information leakage method to attack a set of clients without fear of VM migrations. In this experiment we have the following settings:

- 3 Machines with 3 VM slots each,
- 4 Clients: Client $\{0, 1, 2\}$ have 2 VMs and Client 3 has 3 VMs,
- Migration Budget: $m = 5$, Sliding Window: $\Delta = 10$, and
- Initial Placement: Client $\{0, 1, 2\}$ have their own machine and Client 3 has 1 VM on each machine (see epoch 0 in Figure 7.10).

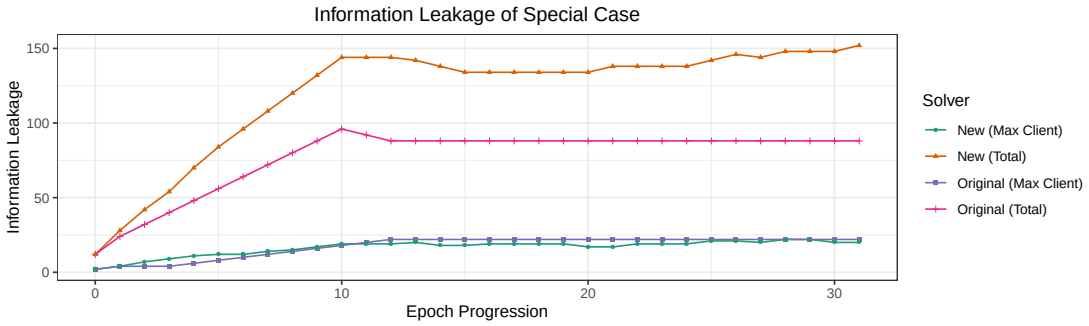


Figure 7.11: Comparison of the Total Information Leakage and the Max Client-to-Client Information Leakage across the original and new ILP reductions. The first 16 placement maps are shown in Figure 7.10. The absolute value of the Total and Max Client-to-Client information leakage is greatly effected by the sliding window. In this example the sliding window is set to 10.

Figure 7.3 shows an example where the method from Moon et al. [73] created a placement map which was sub-optimal in terms of total information leakage compared to our methods. That was an example where an attacker could utilize Nomad's sub-optimal, greedy, design to their advantage and under take a side-channel attack. We have created a similar example in Figure 7.10, to show a new attack vector for the static placement map that occur with minimizing the total information leakage of a system.

7.5.1 Max Client-to-Client Information Leakage

In Figure 7.10, the reason why the original solution is unable to change the placement map after epoch 3 is because it is minimizing the information leakage across all clients, where each client-to-client is weighted equally. But this situation allows for leaving client's in non-zero information leakage states with the same client for indefinite amount of time. This is the type of situation an attacker would like to create, in order to successfully attack a client and circumvent the security of the cloud infrastructure. We solve this issue by minimizing the maximum individual Client-to-Client information leakage. With this focus, we see that the placement maps from epoch 5 onwards, are changing so as to prevent the attacker's ability to steal too much information from a particular client.

In Figure 7.11, we plot the 2 types of information leakage: Total Information Leakage ("Original" – Figure 7.4) and Maximum Client-to-Client Information Leakage ("New" – Figure 7.12). From this graph we see that our "New" method is able to reduce the Max

$$\textbf{minimize} \max_{c_1 \in [1,n]} \left[\sum_{t'=t-\triangle}^{t-1} \left(\sum_{j=1}^{k} F[t][c_1][j] \times \left(\sum_{c_2 \neq c_1}^{n} F[t][c_2][j] \right) \right) \right] \tag{7.7}$$

subject to

$$2 \times m \geq \sum_{c=1}^{n} \sum_{j=1}^{k} |F[t][c][j] - F[t-1][c][j]| \tag{7.8}$$

$$s_j \geq \sum_{c=1}^{n} F[t][c][j] \qquad \forall j \in [i,k] \tag{7.9}$$

$$x_i = \sum_{j=1}^{k} F[t][c][j] \qquad \forall c \in [1,n] \tag{7.10}$$

$$0 \leq F[t][c][j] \qquad \forall c, j \tag{7.11}$$

Figure 7.12: New ILP problem for the $\langle R, C \rangle$ model for minimizing the Max Client-to-Client Information Leakage. This is different from the $\langle R, NC \rangle$ and $\langle NR, C \rangle$ models as it models. The source code for this reduction in CPLEX OPL is attach in Appendix I.

Client information leakage when compared to the "Original" method. We can also see that the New method has much more information leakage between all clients. This can be a problem if we assume a majority of the clients are attackers. This assumption might be unrealistic in some security models.

The choice of sliding window greatly effects how this Figure 7.11 looks, larger sliding windows produce larger numbers and a bigger separation between the New and Original methods for both Total and Max Client models.

This new method of information leakage prevents the attacker from exploiting the security flaw in single non-zero information leakage cloud environments, but at the cost of more total information leakage within the cloud system. This can be mitigated by increasing the frequency of epochs and by restricting the ability for the cloud to enter these single non-zero information leakage states.

7.6 Future Work

The Max Client-to-Client Information Leakage model solves the security vulnerability of the Total Information Leakage model but at the cost of increased total information leakage of the cloud environment. Future work into this field would combine the Max Client-to-Client and Total Information Leakage such that the security vulnerabilities are abated, the Total information leakage is lowered, and the effect on performance is still **NP**-hard (and decision problem is **NP**-complete).

7.7 Conclusions

We have designed and implemented an efficient reduction to ILP. We have conducted a limited empirical assessment with a focus on the main points of our work. Our empirical results suggest that the overhead imposed by an approach that indeed decreases information leakage is more than prior work suggests, but lower than previous work reported. For inputs that prior work suggests take longer than a day with an ILP approach, our implementation only requires a few minutes. This suggests that further research-investment in well-founded approaches can decrease the overhead further.

This chapter has provided the following contributions:

- Efficient reduction to ILP for minimizing Total Information Leakage in $\langle R, C \rangle$ mode,
- Matrix form of the ILP reduction in Figure 7.4 (this differs from the paper [60]),
- A new attack for Total Information Leakage,
- A new ILP method in Figure 7.12, which focuses on minimizing the maximum Client-to-Client information leakage, and addresses the attack on Total Information Leakage.

In this chapter the Max Client-to-Client Information Leakage was tested against the Total Information Leakage and shown to have a negligible effect on performance. We also were able to create an example which shows the security vulnerability in the Total Information Leakage model, which is solved by the new model. This example is not singular, but there exists infinitively many cloud environments that follow the same single non-zero information leakage format as shown in Figure 7.10.

Acknowledgements

We thank the authors of Nomad [73] for making a version of their implementation available to us.

Chapter 8

ILP Attack on Logic-Locked Circuits

Contents

Declaration of Contributions

This research was performed with the collaboration of Nahid Juma, Mahesh Tripunitara, and later on Mohamed El Massad. Section 8.1.1 is shared work with the above co-authors. Table J.1 contains reduction from Boolean to CNF SAT to ILP, the Boolean to CNF SAT v1 was created by Mahesh Tripunitara, and CNF SAT v2 is adapted from [28]. I created the reduction in Table 8.1, and I reduced from CNF SAT to ILP in Table J.1. All other sections are my sole work.

8.1 Introduction

In electronic manufacturing, companies can spend million of dollars in research and development for a new product. When that product is ready for production, they will often save money by hiring contractors to create pieces of the product and then assemble with another contractor. Utilizing contractors introduces the threat of counterfeiting, unauthorized overproduction, and stolen intellectual property.

Most sales of electronic items occur early in the life of the product, this is due to continual innovations in technology that can make past products obsolete or ineffective when compared to newer products. Thus any protection mechanism would only need to protect the circuit for 1-3 years after production begins. After this period, the company has recouped their losses from research and development and make a profit, if the product was successful. Protection mechanisms used for securing national security, i.e. nuclear and rocket control circuits, would require much longer time frames and much higher security guarantees.

One security measure proposed to protect integrated circuits (ICs) is called Logic Locking (alternatively called Logic Encryption [90]). Logic locking obfuscates a circuit by modifying the circuit gates, and adding input wires, known as the key inputs, where the incorrect values render the circuit ineffective. The key inputs need to integrated into the circuit in such a fashion that without the correct digital signals running on the key inputs, the locked circuit will not function identically to the original circuit. We show 2 basic forms of logic locking, XOR and MUX gates, in Section 2.4.2.

There are 2 metrics a logic locked circuit can be measured: how many new key inputs, and how integrated are the key inputs. If the key size is too small, then a brute force method can be used to find the correct key. Storing the key in a secure location, and transporting it to the circuit, is expensive and increases with the size of the key. If the circuit and key are loosely coupled, then brute and manual analysis can be used to remove the key. If the locked circuit integrates the key into every layer of the circuit multiple times, then the cost of the circuit increases. We measure the level of protection to the original circuit with encryption overhead. Where 5% encryption overhead means that the locked circuits number of gates is approximately 5% more than the original circuit.

This chapter recreates the successful SAT-Attack, from Subramanyan et al. [127], utilizing Integer Linear Programming (ILP) as the NP solver. ILP is not as natural a substitute to simulate circuits compared to SAT, but ILP solvers are quite powerful. Our contributions are the following:

1. Logic Locking ILP-Attack

2. 4 different Logic Gates to ILP reductions
3. Optimizations to the attack
4. Circuit optimization for known input/output pairs

In Section 8.1.1, we outline the related and prior works in logic locking protection mechanisms and attacks. In Section 8.2 we describe circuit representation and 2 basic logic locking techniques. In Section 8.3 we define the Logic Locking Problem. In Section 8.4 and Section 8.5 we describe the brute force and SAT attacks. In Section 8.6 we detail the reduction from Logic Locking to ILP. In Section 8.7 and Section 8.8 we outline the experiments and results obtained using the ILP Attack. Section 8.9 outlines future work, and we conclude in Section 8.10.

8.1.1 Related Works

Logic locking, to our knowledge, was first introduced in 2008 by Roy et al. [98]. In their scheme, EPIC, a set of XOR/XNOR key gates is randomly inserted into the original circuit, such that one input of each key gate connects to a key input and the other input is driven by a wire in the original circuit. Applying a correct key to the key inputs renders the locked circuit equivalent to the original. Rajendran et al. [88] observed that an adversary with access to a working IC could attack EPIC by using an automatic test pattern generation (ATPG) algorithm to identify a correct key. They proposed a scheme based on graph theory that seeks to be resilient to such an attack.

In 2015, El Massad et al. [33] and Subramanyan et al. [127] concurrently published attacks against Rajendran et al.'s scheme [88]. Their attacks are referred to in the literature as the "SAT attack" and it is based on the observation that a single input/output observation from a working chip can potentially eliminate a large number of incorrect keys. The discovery of the SAT attack was considered seminal because it defeated all locking schemes known at the time [11, 32, 89].

Much of the subsequent work focused on devising SAT-resilient schemes. These can broadly be categorized into three:

1. Formulate locking, and obfuscation, mechanisms that are not translatable to SAT,
2. Make the underlying SAT problems hard, and
3. Make the number of SAT problems the SAT attack must solve in order to be successful exponential.

Examples of defences in the first category are cyclic obfuscation [118] and behavioural locking of the logic [143]. In cyclic obfuscation, the key idea is to introduce logical loops

in the circuit so that it can no longer be represented as a directed acyclic graph. This forces a SAT attack to either be trapped in an infinite loop or to generate an incorrect key upon termination [97]. In behavioural locking, the key not only determines circuit's functionality, but also the timing profile. If the SAT model does not properly simulate the timing profile for each key, this will result in timing violations and thus making the circuit malfunction. Shortly after the introduction of these schemes, stronger attacks such as cycSAT [160] and the Satisfiability Module Theories (SMT) attack [8] were discovered that were able to model cyclic or behavioural locking into a SAT or SAT+theory solvable logic problems.

Defences in the second category include Cross-lock [119] and Full-lock [62]. In Cross-lock, a onetime programmable interconnect mesh is used to obfuscate the routing of the circuit, in order to substantially increase the runtime of each iteration of the SAT attack. In Full-lock a set of cascaded fully programmable logic and routing block (PLR) networks replace parts of the logic and routing in the desired circuit, resulting in harder SAT instances than Cross-lock. Other defences in this category are those that incorporate one-way functions to make the underlying SAT instances hard [152, 159].

Defences in the third category include those that are based on point functions [149, 142, 67]. For example in SARLock [149], every incorrect key is wrong for the single input that is identical to it. Similarly, the Anti-SAT block scheme [142] utilizes functions whose number of input vectors that result in an output of 1 is either very high or very low. Although these defences make the number of iterations involved in the SAT attack exponential, they are vulnerable to other attacks such as the Signal Probability Skew (SPS) attack [151], removal attacks [150], approximate attacks [117, 120], the bypass attack [144], and the Satisfiability Module Theories (SMT) attack [8]. Also in the third category is the more recent and state-of-the-art defence TTLock [154] and its subsequent generalization SFLL [153]. Key-recovery attacks against TTLock and SFLL include the Functional Analysis Attack [123], and an attack that exploits structural traces left behind in the circuit [146].

8.2 Background

The Logic Locking background has been moved to the Section 2.4.

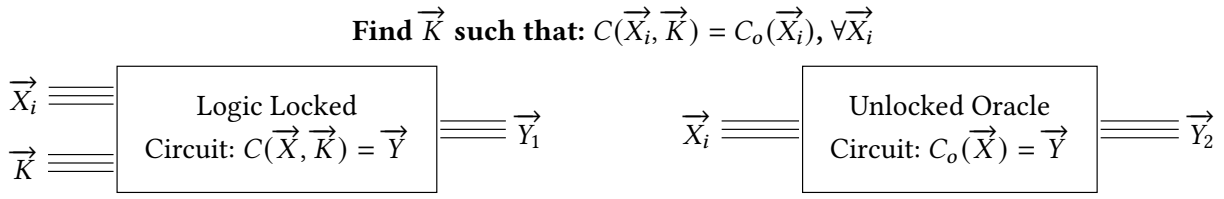


Figure 8.1: The Logic Locking Key Recovery Problem. Given a description of the locked circuit C ($\vec{X}$ and $\vec{K}$ inputs; $\vec{Y}$ outputs) and access to an unlocked black box circuit C_o. Find the values of $\vec{K}$ such that $C(\vec{X_i}, \vec{K}) == C_o(\vec{X_i})\ \forall \vec{X_i}$.

8.3 The Logic Locking Problem

In this section we formally define the Logic Locking problem and the attacker model used. In Figure 8.1, we provide a diagram of the key recovery problem in logic locking.

Notation Let the original, unsecured, circuit be C_o. Let the input wires to C_o be a Boolean vector $\vec{X}$. Let the output wires of C_o be $\vec{Y}$. The logic locked circuit corresponding to C_o is C. The inputs to C are the primary inputs of C_o and the key inputs $\vec{K}$. We will use function notation for circuits: $C_o(\vec{X_i}) = \vec{Y_i}$.

Problem Description For a given locked circuit C and an oracle C_o, find the set $\vec{K}$ such that $C_o(\vec{X_i}) = C(\vec{X_i}, \vec{K})\ \forall \vec{X_i}$. The attacker is provided a full description of the locked circuit C, experimentally this is provided as the circuit's netlist. The attacker has blackbox oracle access to the unlocked circuit and is able to perform arbitrary calls for any $\vec{X_i}$, i.e. $C_o(\vec{X_i}) = \vec{Y_i}$. Experimentally we provide the original circuit netlist for use to only perform these oracle requests. We are limiting our experimentations to full key recovery. An alternative logic locking problem is partial key recovery, where a certain percentage of the key is required for the attacker to be successful.

Key Space Some logic locking techniques can create a locked circuit C such that more than one key, $\vec{K_1}$ and $\vec{K_2}$, satisfy the logic locking problem above. The attacker only needs to return the first key which satisfies the problem.

Distinguishing Inputs Some logic locking techniques create locked circuits C such that for a well selected input, $\vec{X_1}$, this input and oracle output, $C_o(\vec{X_1})$, are able to invalidate many keys from the remaining key space. It is this property which the SAT-Attack uses to reduce the key space without requiring exponential time. Logic locking techniques have been proposed which reduce the ability for specific inputs to act as distinguishing inputs.

Algorithm 1 Logic Decryption Algorithm

Function: *decrypt*.
Inputs: C and *eval*.
Output: $\vec{\mathtt{K}_\mathtt{C}}$.

1: $i := 1$
2: $F_1 = C(\vec{X}, \vec{K}_1, \vec{Y}_1) \wedge C(\vec{X}, \vec{K}_2, \vec{Y}_2)$
3: **while** $sat[F_i \wedge (\vec{Y}_1 \neq \vec{Y}_2)]$ **do**
4: $\quad \vec{\mathtt{X}}_i^d := sat\_assignment_{\vec{X}}[F_i \wedge (\vec{Y}_1 \neq \vec{Y}_2)]$
5: $\quad \vec{\mathtt{Y}}_i^d := eval(\vec{\mathtt{X}}_i^d)$
6: $\quad F_{i+1} := F_i \wedge C(\vec{\mathtt{X}}_i^d, \vec{K}_1, \vec{\mathtt{Y}}_i^d) \wedge C(\vec{\mathtt{X}}_i^d, \vec{K}_2, \vec{\mathtt{Y}}_i^d)$
7: $\quad i := i + 1$
8: **end while**
9: $\vec{\mathtt{K}_\mathtt{C}} := sat\_assignment_{\vec{K}_1}(F_i)$

Figure 8.2: SAT-Attack Algorithm from Subramanyan et al.[127]

8.4 Brute Force Attack

Let the number of input bits be $|\vec{X}|$ and number of key bits be $|\vec{K}|$. The exhaustive brute force algorithm is $O(2^{|\vec{X}|+|\vec{K}|})$ in the worst case. Then the attacker tries each of the $2^{|\vec{K}|}$ keys on all $2^{|\vec{X}|}$ input/ouput pairs to determine the first key, which satisfies all input/output pairs.

Analysis of a locked circuit, and set of correct input/output pairs, can reveal extra propositional logic that can restrict the input and key space that is require to search. In Figure 2.6, we can determine extra information from the gate `po1`$= (\neg \text{keyinput2}) \oplus (\text{pi01} \wedge \text{pi02})$. If $\text{po1} = 1$, then the input and key set is reduced such that only input/key pairs that satisfy the condition:

$$(\text{pi01} \wedge \text{pi02} \wedge \text{keyinput2}) \vee ((\neg\text{pi01} \vee \neg\text{pi02}) \wedge \neg\text{keyinput2})$$

8.5 The SAT-Attack

In Figure 8.2 we have provided the SAT-Attack created in [127]. The notation for a locked circuit C differs from this thesis; these are equivalent $C(\vec{X}, \vec{K}) = \vec{Y} \equiv C(\vec{X}, \vec{K}, \vec{Y})$. On line 2, 2 circuits are created, they share the same input vector, but have different variables for the keys and outputs. In line 3, they loop their incrementally larger SAT problem until

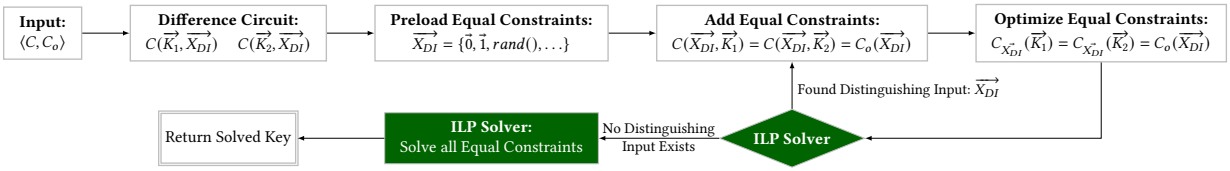


Figure 8.3: Flow diagram for the ILP Attack. The attack takes as input: locked circuit C and an oracle C_o. The attack repeatedly finds distinguishing inputs using an ILP Solver. When no more distinguishing inputs are found, the ILP Solver is used to solve for a key which satisfies all distinguishing inputs and matches the output from the Oracle.

it becomes unsatisfiable. The SAT problem they are creating in lines 2, 3, and 6, is: Does there exists an input $\vec{X}$ such that 2 keys can be found where $\vec{Y_1} \neq \vec{Y_2}$ and the keys are able to correctly produce all previous distinguishing input/output pairs $\vec{X_i^d}, \vec{Y_i^d}$. The SAT problem on line 4 returns the distinguishing input $\vec{X_i^d}$, and line 5 returns the correct output for that distinguishing input (which can be different from the sat assignment value). Line 6 adds the new constraint that both keys must match this input/output pair from the oracle. Line 9 returns the key that is able to correctly satisfy all distinguishing input/output pairs from the oracle.

8.6 Reduction To ILP

The ILP Attack, outlined in Figure 8.3, takes as input a locked circuit C and an oracle C_o. From this we convert C into an ILP representation, such that the input wires are unrestricted, but the output wires are conditional on which values the input wires take. Thus we create a simulation of C in ILP, where the input, output, and internal wires are all ILP integer variables restricted to the range $[0, 1]$. We go into details in Section 8.6.1.

The second step in the reduction is to create a difference circuit, where there are 2 circuits C_1 and C_2, which share the same input, but have different variables representing their keys. The ILP reduction forces $C_1(\overrightarrow{K_1}, \overrightarrow{X_{DI}}) \neq C_2(\overrightarrow{K_2}, \overrightarrow{X_{DI}})$. This constraint must be satisfied for the input to be considered a distinguishing input. The details are outlined in Section 8.6.2.

The first distinguishing inputs are often not special or specifically hard to compute. Thus the next step in Figure 8.3 is to preload the attack with specific and random distinguishing inputs. This component can increase performance. The amount of preload is user defined, with our empirical solution allowing for: User defined number of random

distinguishing inputs, or User provided list of distinguishing inputs (this can be externally calculated). Section 8.6.3 discusses the details.

Each preloaded distinguishing input is added to an Equal circuit, which forces the keys from the difference circuit to be valid keys for the previous found distinguishing inputs and their oracle output. The Reduction for the equal circuit is shown in Section 8.6.4. Each equal is optimized using the known input and the oracle output values. The process for optimizing these circuits is shown in Section 8.6.5.

The next step of the attack enters a loop, which can have an exponential number of iterations in the worst case $2^{|\vec{X}|}$. The SAT attack was successful against previous logic locking techniques because of the relatively low number of distinguishing inputs required to exit this loop. This loop can only exit if no new distinguishing input can be found. This occurs once all distinguishing inputs are exhaustively found, this set can be equal to the set of all possible inputs. The quality of the distinguishing inputs found, can reduce the size of this set.

After all distinguishing inputs are found, the attack creates a new ILP problem with all equal circuits for the shared key input $\vec{K_1}$. The solution to this is a key that produces the same output as the oracle for all found distinguishing inputs. This last step can fail to find a key if the circuit C is unable to be represented probably in ILP/SAT, or if the circuit contains internal saved state which changes output of circuit depending on the inputs previously received. Prior work has been proposed which implement such protection mechanisms (see Section 8.1.1).

8.6.1 Simulating Circuits in ILP

The first step of the ILP Reduction is to convert the locked circuit to an equivalent ILP program. In Table 8.1, we outline 2 of the 4 methods we used to convert circuit gates into ILP constraints. The other 2 ILP reductions are outlined in Table J.1.

When converting the circuit in Figure 2.6 into a ILP simulation, all inputs, outputs, and internal wires (left hand of equal sign) are declared as binary integers. The ILP objective function is empty. The order of constraints does not matter. All gates in the circuit are converted by using one of the reductions from Table 8.1 or Table J.1. The ILP reductions contain no assumptions, thus a mix of all the 4 reductions, even for the same gate, is possible.

During the reduction for each gate, extra variables are required for the (A) reduction for the NAND, NOR, XOR, and 2 are required for the XNOR gates. In reduction (B)

Table 8.1: Reduction from specific logic/circuit gate to ILP constraint. Reduced ILP problem has no objective function, only constraints. Reduction (A) requires extra variables for NAND, NOR, XOR, and 2 for XNOR. Reduction (B) removes these extra variables. All variables must be an integer in the range $[0, 1]$.

GATE	Boolean Expression	ILP Constraints (A)	Reduced ILP Constraints (B)
NOT	$y = \neg x_1$	$y = 1 - x_1$	$y = 1 - x_1$
BUF	$y = x_1$	$y = x_1$	$y = x_1$
AND	$y = x_1 \wedge x_2$	$0 \leq x_1 + x_2 - 2y \leq 1$	$0 \leq x_1 + x_2 - 2y \leq 1$
NAND	$y = \neg(x_1 \wedge x_2)$	$0 \leq x_1 + x_2 - 2z \leq 1$ $y = 1 - z$	$0 \leq x_1 + x_2 - 2 + 2 * y \leq 1$
OR	$y = x_1 \vee x_2$	$0 \leq 2y - x_1 - x_2 \leq 1$	$0 \leq 2y - x_1 - x_2 \leq 1$
NOR	$y = \neg(x_1 \vee x_2)$	$0 \leq 2z - x_1 - x_2 \leq 1$ $y = 1 - z$	$0 \leq 2 - 2 * y - x_1 - x_2 \leq 1$
XOR	$y = x_1 \oplus x_2$	$y == x_1 + x_2 - 2t$	$y == abs(x_1 - x_2)$
XNOR	$y = \neg(x_1 \oplus x_2)$	$z == x_1 + x_2 - 2t$ $y == 1 - z$	$y == 1 - abs(x_1 - x_2)$

we are able to remove these extra variables from the constraints. Reductions (C) and (D) do not require any additional variables. CPLEX [51] and Gurobi [45], the ILP solvers used for this project, converts the `abs` function to an extra range variable, which is not the same variable type as in reduction (A) and only one constraint for XNOR. CPLEX/Gurobi converts all dual inequalities into 2 inequalities: $0 \leq b \leq 1$ would be $0 \leq b$ and $b \leq 1$.

8.6.2 Difference Circuit

The difference circuit takes as input: a Circuit C, input wires $\overrightarrow{X_{DI}}$, 2 key wires sets $\overrightarrow{K_1}/\overrightarrow{K_2}$, and ensures that the output from those 2 circuits differ by at least one bit. The input wires are only constrained by this difference circuit. The 2 key wires are constrained in the difference circuit and all equal circuits created and appended to the ILP problem. The input wires that are solved for here is called a new distinguishing input.

In Figure 8.4, we show the difference circuit that can be used to ensure that the output of the circuit $C(X_{DI}, K_1)$ differs by at least 1 bit from $C(X_{DI}, K_2)$. We convert each of the XOR gates using Table 8.1, and then sum the result and force this to be greater than or equal to 1.

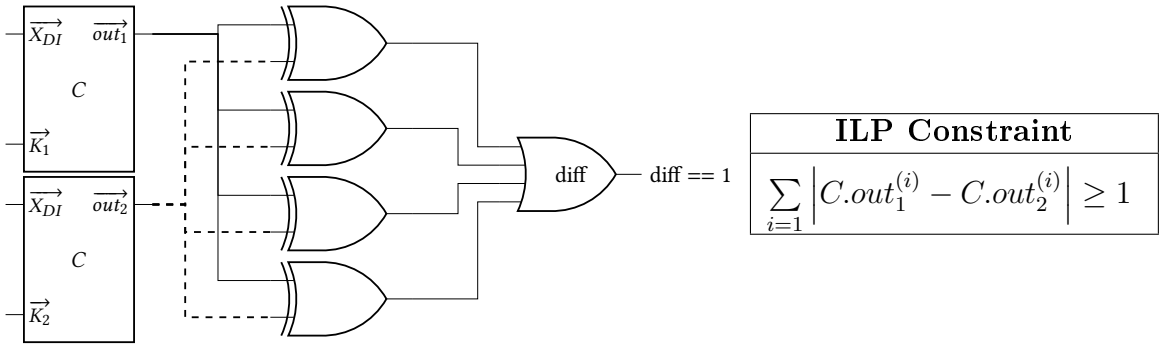


Figure 8.4: Circuit and ILP equivalent, of a constraint to force the output of 2 circuits to not equal. The ILP equivalent is condensed and easier to understand.

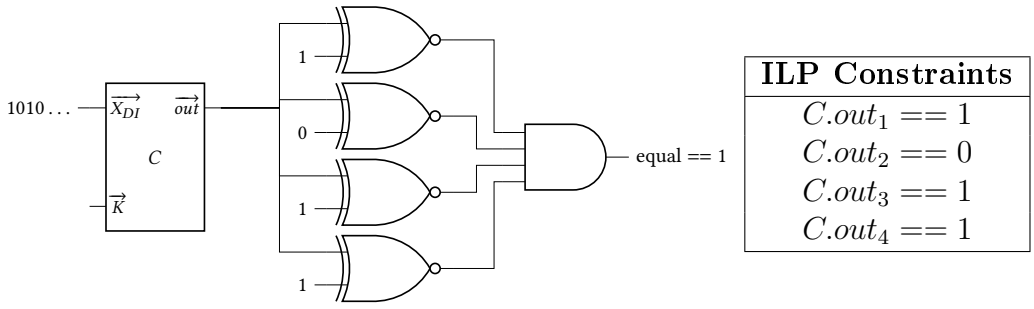


Figure 8.5: Circuit and ILP equivalent of the constraints used to force the output of a circuit to equal a set of bits. The ILP equivalent is condensed and easier to understand.

8.6.3 Preloading Distinguishing Inputs

We have empirically found, and through the work of [33], that randomly selecting inputs as distinguishing inputs can provide useful distinguishing inputs in the start of the attack. Preloading the circuit with selected distinguishing inputs can skip the overhead of running the ILP solver many times to produce similar quality distinguishing inputs. From the work of [33], they showed that utilizing only randomly generated distinguished inputs requires larger sets to complete the attack; we have found this to be true with the ILP attack as well.

We have provided 2 methods of preloading inputs into the attack: user defined number of randomly generated inputs, and user provided list of distinguishing inputs. The randomly generated inputs was empirically faster for smaller sizes (5 to 10 inputs), after this the attack would not see much improvement and only required more memory and

Gate	Pin Input: 0	Pin Input: 1
BUF	OFF	ON
NOT	ON	OFF
AND	OFF	BUF
NAND	ON	NOT
OR	BUF	ON
NOR	NOT	OFF
XOR	BUF	NOT
XNOR	NOT	BUF

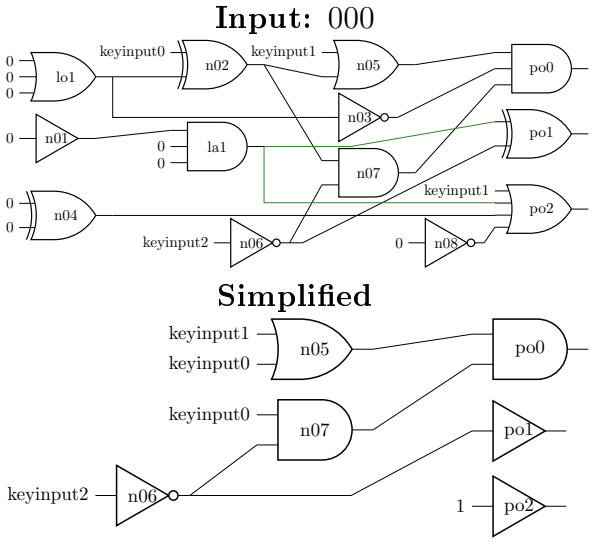


Figure 8.6: Simplifying each Equal Circuit by propagating the known Distinguishing Inputs through them. This simplifies the circuit and reduces the number of constraints.

computation time to complete iterations. The user provided list of distinguishing inputs allow for additional analysis to be performed on the circuit and the oracle that might yield higher quality distinguishing inputs, or similar quality inputs for less computation time. For repeatability, all experimental results were preloaded with the $\overrightarrow{0}$ and $\overrightarrow{1}$ distinguishing inputs and no randomly generated inputs.

8.6.4 Equal Circuit

For each distinguishing input, 2 new equal circuits are added to the previous ILP problem. These equal circuits force the 2 key wire sets, used in the difference circuit, to have the correct output for the new distinguishing input and oracle output. The equal circuit is used to reduce the key space by using the distinguishing input to invalidate hopefully many keys.

In Figure 8.5, we show the equal circuit that is used to ensure that the output of the circuit C, when given the specific input X_{DI}, exactly matches the output obtained from the oracle. Converting this circuit directly to ILP, usingTable 8.1, produces a more complex reduction than if we simply force each output wire to match the bit from the oracle. This is much easier to understand and contains less constraints.

Table 8.2: Benchmark circuits used to test the ILP Attack.

(a) ISCAS'85 Circuits [20]

Circuit	Input	Output	Gates
c432	36	7	160
c499	41	32	202
c880	60	26	383
c1355	41	32	546
c1908	33	25	880
c2670	233	64	1193
c3540	50	22	1669
c5315	178	123	2307
c7552	207	107	3512

(b) Select MCNC Circuits [147]

Circuit	Input	Output	Gates
i4	192	6	338
apex2	39	3	610
i9	88	63	1035
ex5	8	63	1055
i7	199	67	1315
k2	46	45	1815
dalu	75	16	2298
i8	133	81	2464
seq	41	35	3519
ex1010	10	10	5066
apex4	10	19	5360
des	256	245	6473

8.6.5 Simplifying Equals Circuits

Before the equal circuits get added to the ILP problem, we simplify the circuits to reduce computation required by the ILP solver. In Figure 8.6, we show the rules used to simplify the equal circuits based on the distinguishing input X_{DI}. We also show an example, where we simplify the circuit from Figure 2.6 and the distinguishing input 000. The equal circuit before the simplification had 19 variables representing the input/output/internal wires. After simplifying, and removing the unneeded `po2` gate, the circuit has 8 variables representing the input/output/internal wires. We can clearly see that this simplification can be very beneficial to the ILP solver, it requires less memory storing the constraints, and if the ILP solver is unable to perform this type of optimization, then more computation to find a new distinguishing input that satisfies all constraints.

8.7 Experiments

In Table 8.2, we detail the benchmark circuits used in our experiments. All benchmarks are considered small circuits when compared to the density of modern CPUs, which have hundreds of millions of transistors, and gates are made up 1 to 6 transistors. The Intel i7-940 CPU has approximately 731 million transistors [53]. The size of these circuits is still relevant to circuit manufactures, smaller and more specialized electronics do not require the

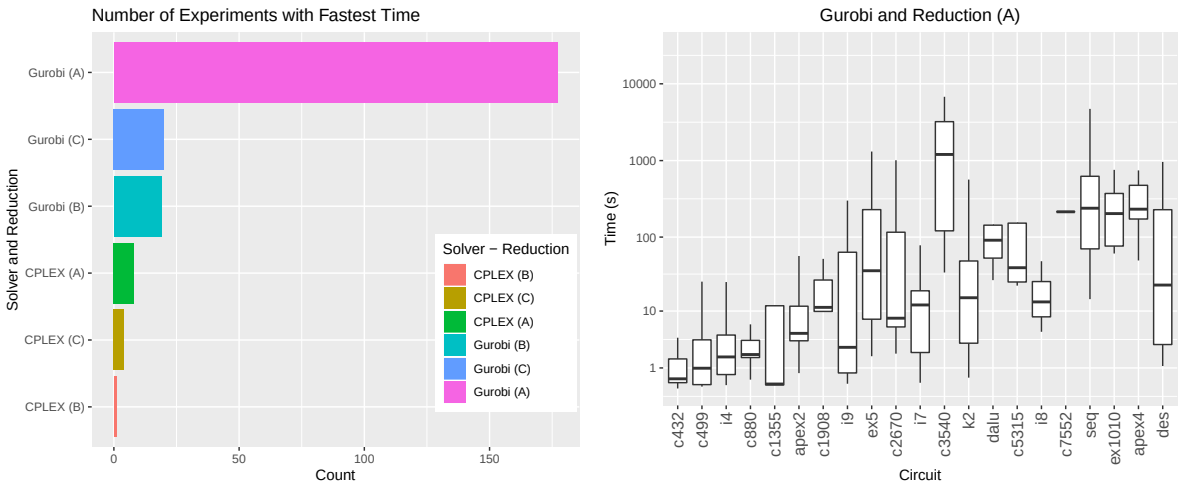


Figure 8.7: Ranking the best ILP Solver and Reduction, and the duration the best took to solve all benchmarks. On the left: the number of solved circuits where a particular solver/reduction pair had the fastest time. Reduction (D) is not on the graph as it never solved a circuit fastest. On the right: the timing of circuit benchmarks from Table 8.2 across all locking algorithms and encryption overheads. The circuits are sorted by the number of gates in their unlocked circuit.

generalized architecture of a CPU. All benchmark circuits have been locked with the logic locking techniques: `rnd` (EPIC) [98], `dac12` [88], `toc13xor` and `toc13mux` [89], `iolts14` [32], and `sarlock` applied with `dac12` and with `iolts14` [149]. The benchmark circuits were logic locked with 5%, 10%, 25%, and 50% encryption overhead. The exception is `sarlock`, which was only locked for 5%, 10%, and 25%, and for only the MCNC circuits. These locked circuit netlists are accessible from Subramanyan et al. [127].

All 486 locked circuits were tested with a combination of ILP solver (CPLEX [51] and Gurobi [45]) and the 4 reduction variants (Section 8.6.1 and Appendix J). All experiments were run on an AMD Ryzen 7 2700 8-core 3.2 GHz base clock CPU, with 32 GB of DDR4 RAM, 1 TB of SSD space, and using Ubuntu 18.04. Experiments were allowed to run for a maximum of 10 hrs.

Table 8.3: Number of Circuits Solved with the ILP-Attack. Tests stopped after 10hrs.

Algorithm	Encrypt	ILP/Total	Algorithm	Encrypt	ILP/Total
dac12 [88]	5 %	9 / 21	toc13mux [89]	5 %	19 / 21
	10 %	7 / 21		10 %	15 / 21
	25 %	2 / 21		25 %	12 / 21
	50 %	1 / 21		50 %	9 / 21
iolts14 [32]	5 %	21 / 21	toc13xor [89]	5 %	10 / 21
	10 %	20 / 21		10 %	7 / 21
	25 %	20 / 21		25 %	5 / 21
	50 %	18 / 21		50 %	1 / 21
rnd (EPIC [98])	5 %	19 / 21	sarlock (dac12) [149]	5 %	1 / 11
	10 %	15 / 21		10 %	1 / 11
	25 %	9 / 21		25 %	0 / 11
	50 %	4 / 21			
sarlock (iolts14) [149]	5 %	1 / 11			
	10 %	1 / 11			
	25 %	1 / 11			

8.8 Results

In Figure 8.7, we show the results of running the ILP Attack on all the locked circuits. From the figure on the left we see that Gurobi is able to produce the fastest solve time for all reductions compared to CPLEX. Within the reductions, reduction (A) (from Table 8.1), was able to produce the fastest key by a significant number of circuits. It is this combination, Gurobi and reduction (A), that we show the box plots for each circuit across all locking techniques that were solvable. This plot shows that the `i9` circuit has a lot of variability due to the logic locking technique, whereas `c7552` is much more consistent.

In Table 8.3 we outline the number of solved circuits based on the logic locking technique and the encryption overhead. `sarlock` is able to withstand the ILP-Attack for all but 1 circuit (`ex5`). The ILP attack is not very proficient at solving `dac12` or `toc13xor` locked circuits. The attack works very well with `toc13mux` and `iolts14` locked circuits.

8.9 Future Work

The reductions in Section 8.6.1 are unable to utilize ILP's full strength and utilize more range of the integer variable type. Future work into non-binary variable types could increase the performance of the ILP Attack. Future work into condensing constraints for known sub-circuits into single constraints could increase performance. An example sub-circuit would be a 1 bit adder, using non-binary integer variables could produce the required complexity to model this sub-circuits in a way that is more favourable for the ILP solver. This would also allow for more complex equal circuit simplifications in Section 8.6.5.

In Section 8.6.2, the difference circuit constraint requires at least one bit difference, but future work into using the number of differing bits as part of the optimization function. The idea is that distinguishing inputs which maximize the number of changed bits could reduce the number of distinguishing inputs required to solve for a key.

8.10 Conclusions

Recreating the SAT-Attack using ILP is possible, but it is not as performant as the original SAT-Attack. The ILP version of this attack is not able to solve more benchmarks compared to the SAT approach, nor is it able to significantly outperform on any specific benchmark. From this research we can see that for a logic locking technique to claim to be "SAT-Attack resistant", they must also make the claim for all NP-complete solvers that can be used in place for the SAT solver. While ILP was not able to perform better than SAT, this is not sure for all **NP**-complete solvers. This is a much harder claim for defence technique to make.

Part III

Conclusions

Chapter 9

Conclusions

We have proved our thesis that there exists effective solutions based on polynomially time reductions to Model Checking or Integer Linear Programming (ILP), for the problems: ATRBAC-Safety (Chapter 3 and Chapter 4), Reduced-Solidity Safety (Chapter 5), Minimizing Information Leakage via VM Scheduling (Chapter 7), and ILP Attack on Logic Locked Circuits (Chapter 8). For ATRBAC-Safety and Reduced-Solidity Safety, we have established two new reductions. The ATRBAC-Safety reduction, named Cree, implements 4 optimizations techniques which allowed for very comparable completion times compared to prior work tools and allowed for better scaling to large test cases to support large ATRBAC policies without timing out or failing. The Reduced-Solidity Safety reduction is not fully complete, but in a testable state. This reduction is novel and shows that for test smart contracts it is able to find all vulnerabilities. For VM Scheduling we have established a more efficient reduction, and for the SAT attack we have shown that a reduction to ILP is also effective. The VM Scheduling reduction is able to solve a problem which was previously reported to take over a day in around 4 minutes, and there are instances where our ILP reduction is able to out perform the Nomad (polynomially time greedy algorithm). The ILP attack is implemented with optimization techniques to preload distinguishing inputs and the reduce the number of constraints for each Equal circuits. We have also provided a new attacker vector for Total Information Leakage which completely invalidates the protection mechanisms of VM Scheduling.

Our theory has shown the considerable power available by mature model checker and ILP solvers to solving problems in computer security. By solving this thesis, we have also identified the difficulties that arise from performing a reduction to model checking or ILP. Difficulties in creating an effective reduction comes from performing high level optimizations before passing the problem to the solver. Model checking and ILP solvers

implement their own domain specific optimizations. These optimizations often do not work well with a direct reduction from the computer security problems. This thesis has found that creative optimization, before performing the reduction, create much better results. In Chapter 3, 4 creative optimization techniques were created, and in Chapter 4 we show how effective each technique is to prior test cases. Optimization techniques were created for Chapter 5 and Chapter 8. The reduction for VM Scheduling in Chapter 7 contains an optimization in the reduction that reduced the runtime from 1 day to 4 minutes for cluster size of 40 in Figure 7.5.

This thesis can naturally be applied to many problems in computer security. Future access control schemes should have their first solver be a reduction to model checking. New features are regularly added to EVM and Solidity, if adding those features to Reduced-Solidity and this new language is **PSPACE**-complete, then our solver in Chapter 5 can be modified to support the new features. In Chapter 6, we showed that ATRBAC-Safety can be embedded into smart contracts, if future computer security problems are found to be embedded in smart contracts, this thesis supports extracting the problem and analyzing it outside of the EVM environment. Total Information Leakage was shown to contain an attack vector which invalidates the security protection provided by VM Scheduling, this thesis provides the framework for creating a reduction to ILP for future information leakage models. In Chapter 8, we showed that ILP was an effective tool to perform the SAT attack, this leads to the natural question, how do other mature **NP**-complete solvers perform in attacking logic locked circuits. This thesis naturally extends to any problem that is shown to be **PSPACE**-complete and **NP**-complete.

References

[1] Cygwin. https://www.cygwin.com/.

[2] Lingeling. http://fmv.jku.at/lingeling/.

[3] Leonardo Alt and Christian Reitwiessner. Smt-based verification of solidity smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 376–388, Cham, 2018. Springer International Publishing.

[4] Paul Ammann, Richard J. Lipton, and Ravi S. Sandhu. The expressive power of multi-parent creation in monotonic access control models. *Journal of Computer Security*, 4(2/3):149–166, 1996.

[5] Pedro Antonino and A. W. Roscoe. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity, 2020.

[6] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, USA, 1st edition, 2009.

[7] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptol. ePrint Arch.*, 2016:1007, 2016.

[8] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–122, 2019.

[9] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova, and Bruce Shepard. Co-location-resistant clouds. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, CCSW '14, pages 9–20, New York, NY, USA, 2014. ACM.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3), August 2015.

[11] Alex Baumgarten, Akhilesh Tyagi, and Joseph Zambreno. Preventing ic piracy using reconfigurable logic barriers. *IEEE Design & Test of Computers*, 27(1):66–75, 2010.

[12] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, August 2001.

[13] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96, 2016.

[14] Armin Biere. Lingeling: Sat competition 2016 version. http://fmv.jku.at/lingeling/lingeling-bbc-9230380-160707.tar.gz, Jun 2016.

[15] Armin Biere. Plingeling: Sat competition 2016 version. http://fmv.jku.at/lingeling/plingeling-ayv-86bf266-140429.zip, Jun 2016.

[16] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[17] Armin Biere and Mathias Preiner. Hardware Model Checking Contest 2019. http://fmv.jku.at/hwmcc19/, October 2019.

[18] Khalid Bijon, Ram Krishnan, and Ravi Sandhu. Mitigating multi-tenancy risks in iaas cloud through constraints-driven virtual resource scheduling. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, pages 63–74, New York, NY, USA, 2015. ACM.

[19] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[20] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Circuits and Systems, 1989., IEEE International Symposium on*, pages 1929–1934 vol.3, May 1989.

[21] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[22] Amy Burnett and Markus Gaasedelen. Building up from the Ethereum Bytecode - Practical Decompilation of Ethereum Smart Contracts. https://blog.ret2.io/2018/05/16/practical-eth-decompilation/, 2018. Online; accessed May 2020.

[23] Vitalik Buterin. EIP 170: Contract code size limit. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md, 2020. Online; accessed May 2020.

[24] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. Quantitative analysis of smart contracts. *Lecture Notes in Computer Science*, page 739–767, 2018.

[25] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 7–18, New York, NY, USA, 2017. ACM.

[26] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[27] ConsenSys. Mythrill - Security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril, 2018. Online; accessed May 2020.

[28] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[29] Crytic. ethersplay: EVM dissassembler. https://github.com/crytic/ethersplay, 2020. Online; accessed May 2020.

[30] Crytic. pyevmas: Ethereum Virtual Machine (EVM) disassembler and assembler. https://github.com/crytic/pyevmasm, 2020. Online; accessed May 2020.

[31] Docker. Docker swarm strategies, Accessed: Oct. 2016. Available from https://docs.docker.com/swarm/scheduler/strategy/.

[32] Sophie Dupuis, Papa-Sidi Ba, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans. In *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, pages 49–54. IEEE, 2014.

[33] Mohamed El Massad, Siddharth Garg, and Mahesh V Tripunitara. Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes. In *NDSS*, pages 1–14, 2015.

[34] Ethereum. Solidity, the Contract-Oriented Programming Language. https://github.com/ethereum/solidity, 2020. Online; accessed May 2020.

[35] D. Evans, Anh Nguyen-Tuong, and J Knight. Effectiveness of moving target defenses. In *Moving Target Defenses*, pages 29–48. Springer, 2011.

[36] Eveem. About Eveem - Panoramix. https://www.eveem.org/about/, 2020. Online; accessed May 2020.

[37] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, August 2001.

[38] Anna Lisa Ferrara, P. Madhusudan, and Gennaro Parlato. Policy analysis for self-administrated role-based access control. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 432–447, Berlin, Heidelberg, 2013. Springer-Verlag.

[39] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[40] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[41] Mikhail I. Gofman, Ruiqi Luo, Ayla C. Solomon, Yingbin Zhang, Ping Yang, and Scott D. Stoller. *RBAC-PAT: A Policy Analysis Tool for Role Based Access Control*, pages 46–49. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[42] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[43] Truffle Blockchain Group. Ganache - One Click Blockchain. https://www.trufflesuite.com/ganache, 2020. Online; accessed May 2020.

[44] Mudit Gupta. EIP 1662: Removing Contract Size Limit. https://github.com/ethereum/EIPs/issues/1662, 2020. Online; accessed May 2020.

[45] Gurobi. Gurobi 9.0 Optimizer, 2019. http://www.gurobi.com/products/gurobi-optimizer, Apr 2019.

[46] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 161–179. Springer, 2019.

[47] Yi Han, Tansu Alpcan, Jeffrey Chan, and Christopher Leckie. Security games for virtual machine allocation in cloud computing. In *4th International Conference on Decision and Game Theory for Security - Volume 8252*, GameSec 2013, pages 99–118, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[48] Yi Han, Jeffrey Chan, Tansu Alpcan, and Christopher Leckie. Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *IEEE Transactions on Dependable and Secure Computing*, 2015.

[49] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, August 1976.

[50] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018.

[51] IBM. IBM ILOG CPLEX 12.10, 2019. https://www.ibm.com/analytics/cplex-optimizer, Apr 2019.

[52] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *IACR Cryptology ePrint Archive*, 2015:898, 2015.

[53] Intel. Intel core i7-940 processor. https://ark.intel.com/content/www/us/en/ark/products/37148/intel-core-i7-940-processor-8m-cache-2-93-ghz-4-80-gt-s-intel-qpi.html. Accessed: 2020-05-29.

[54] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 591–604. IEEE, 2015.

[55] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *Research in Attacks, Intrusions and Defenses*, pages 299–319. Springer International Publishing, 2014.

[56] Karthick Jayaraman, Mahesh Tripunitara, Vijay Ganesh, Martin Rinard, and Steve Chapin. Mohawk: Abstraction-refinement and bound-estimation for verifying access control policies. *ACM Trans. Inf. Syst. Secur.*, 15(4):18:1–18:28, April 2013.

[57] S. Jha, Ninghui Li, M. Tripunitara, Qihua Wang, and W.H. Winsborough. Towards formal verification of role-based access control policies. *Dependable and Secure Computing, IEEE Transactions on*, 5(4):242–255, Oct 2008.

[58] Nick Johnson. evmdis: EVM disassembler. https://github.com/Arachnid/evmdis, 2020. Online; accessed May 2020.

[59] Anita Jones. Protection mechanism models: their usefulness. *Foundations of secure Computation*, pages 237–252, 1978.

[60] N. Juma, J. Shahen, K. Bijon, and M. Tripunitara. The overhead from combating side-channels in cloud systems using vm-scheduling. *IEEE Transactions on Dependable and Secure Computing*, 17(2):422–435, 2020.

[61] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, pages 1–12, 2018.

[62] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[63] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[64] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[65] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, M. Beccuti, B. Berthomieu, G. Ciardo, S. Dal Zilio, T. Liebke, S. Li, J. Meijer, A. Miner, J. Srba, Y. Thierry-Mieg, J. van de Pol, T. van Dirk, and K. Wolf. Complete Results for the

2019 Edition of the Model Checking Contest. http://mcc.lip6.fr/2019/results.php, April 2019.

[66] Adam J. Lee, Kent E. Seamons, Marianne Winslett, and Ting Yu. *Automated Trust Negotiation in Open Systems*, pages 217–258. Springer US, Boston, MA, 2007.

[67] Meng Li, Kaveh Shamsi, Travis Meade, Zheng Zhao, Bei Yu, Yier Jin, and David Z Pan. Provably secure camouflaging strategy for ic protection. *IEEE transactions on computer-aided design of integrated circuits and systems*, 2017.

[68] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. *ACM Trans. Inf. Syst. Secur.*, 9(4):391–420, November 2006.

[69] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, March 2016.

[70] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 203–215, Washington, DC, USA, 2014. IEEE Computer Society.

[71] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, May 2015 2015.

[72] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.

[73] Soo-Jin Moon, Vyas Sekar, and Michael K. Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1595–1606, New York, NY, USA, 2015. ACM.

[74] Nick Mudge. EIP 2535: Diamond Standard. https://github.com/ethereum/EIPs/issues/2535, 2020. Online; accessed May 2020.

[75] Patrick Nelson. How one startup hopes to solve server underutilization. *Network World*, August 2015. Available from http://www.networkworld.com/article/2959532/data-center/startup-says-it-has-solved-server-underutilization.html.

[76] Niklas Eén and Niklas Sörensson. MiniSat 2.2. http://minisat.se/, Jun 2016.

[77] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 653–663, New York, NY, USA, 2018. Association for Computing Machinery.

[78] NuSMV. NuSMV 2.6.0, 2015. http://nusmv.fbk.eu/, Jun 2016.

[79] The Open Group. *Seconds Since the Epoch*.

[80] Rodney Owens and Weichao Wang. Non-interactive os fingerprinting through memory de-duplication technique in virtual machines. In *30th IEEE International Performance Computing and Communications Conference*, pages 1–8, November 2011.

[81] Natasha Pabrai, Jan Keller, Jessica Lin, Anna Hupa, and Adam Bacchus. Vulnerability Reward Program: 2019 Year in Review. https://security.googleblog.com/2020/01/vulnerability-reward-program-2019-year.html, 2020. Online; accessed May 2020.

[82] D. Page. Partitioned cache architecture as a side-channel defence mechanism, 2005.

[83] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 912–915, New York, NY, USA, 2018. Association for Computing Machinery.

[84] Erman Pattuk, Murat Kantarcioglu, Zhiqiang Lin, and Huseyin Ulusoy. Preventing cryptographic key leakage in cloud virtual machines. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 703–718, San Diego, CA, August 2014. USENIX Association.

[85] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[86] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[87] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 77–84, New York, NY, USA, 2009. ACM.

[88] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*, pages 83–89, 2012.

[89] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Fault analysis-based logic encryption. *IEEE Transactions on computers*, 64(2):410–424, 2013.

[90] Jeyavijayan (JV) Rajendran and Siddharth Garg. *Logic Encryption*, pages 71–88. Springer International Publishing, Cham, 2017.

[91] P. Rajkumar and R. Sandhu. Safety decidability for pre-authorization usage control with finite attribute domains. *IEEE Transactions on Dependable and Secure Computing*, 13(05):582–590, sep 2016.

[92] P. V. Rajkumar and R. Sandhu. Safety decidability for pre-authorization usage control with identifier attribute domains. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.

[93] Silvio Ranise, Anh Truong, and Alessandro Armando. Scalable and Precise Automated Analysis of Administrative Temporal Role-based Access Control. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*, SACMAT '14, pages 103–114, New York, NY, USA, 2014. ACM.

[94] Silvio Ranise, Anh Truong, and Luca Viganò. Automated analysis of rbac policies with temporal constraints and static role hierarchies. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 2177–2184, New York, NY, USA, 2015. ACM.

[95] Silvio Ranise, Anh Truong, and Luca Viganò. Automated analysis of rbac policies with temporal constraints and static role hierarchies. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 2177–2184, New York, NY, USA, 2015. Association for Computing Machinery.

[96] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[97] Shervin Roshanisefat, Hadi Mardani Kamali, and Avesta Sasan. Srclock: Sat-resistant cyclic logic locking for protecting the hardware. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 153–158, 2018.

[98] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. Epic: Ending piracy of integrated circuits. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1069–1074, 2008.

[99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The arbac97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, February 1999.

[100] Ravi S Sandhu. The typed access matrix model. In *IEEE Symposium on Security and Privacy*, pages 122–136, 1992.

[101] Amit Sasturkar, Ping Yang, Scott D. Stoller, and C. R. Ramakrishnan. Policy analysis for administrative role based access control. *Theoretical Computer Science*, 412(44):6208–6234, October 2011.

[102] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970.

[103] Jonathan Shahen. Cree: Source Code and Supplementary Material. https://ece.uwaterloo.ca/~jmshahen/cree/, Jan 2019.

[104] Jonathan Shahen. Cree and Mohawk+T datasets. https://git.uwaterloo.ca/atrbac-safety-mohawk-t-cree/mohawk-t-data, 2015. Online; accessed Jun 2020.

[105] Jonathan Shahen. Cree: Source Code. https://ece.uwaterloo.ca/~jmshahen/mohawk+t/, 2015. Online; accessed Jun 2018.

[106] Jonathan Shahen. Mohawk+T: Source Code. https://ece.uwaterloo.ca/~jmshahen/mohawk+t/, 2015. Online; accessed Jun 2015.

[107] Jonathan Shahen. Mohawk+T: Efficient Analysis of Administrative Temporal Role-Based Access Control (ATRBAC) Policies. Master's thesis, University of Waterloo, 2016. Available from https://uwspace.uwaterloo.ca/.

[108] Jonathan Shahen. Vagabond: Source Code. https://git.uwaterloo.ca/jmshahen/vagabond, 2018. Online; accessed May 2018.

[109] Jonathan Shahen. Analyze ATRBAC Policies Source Code. https://git.uwaterloo.ca/atrbac-safety-mohawk-t-cree/cree-policy-analyzer, 2019. Online; accessed May 2020.

[110] Jonathan Shahen. Generic ATRBAC Smart Contract Source Code. https://git.uwaterloo.ca/jmshahen/ethereum-atrbac-smart-contract, 2019. Online; accessed May 2020.

[111] Jonathan Shahen. ILP Logic Locking: Source Code. https://git.uwaterloo.ca/jmshahen/LogicLocking-Empirical, 2019. Online; accessed June 2019.

[112] Jonathan Shahen. Convert ATRBAC Policy to Smart Contract Source Code. https://git.uwaterloo.ca/jmshahen/atrbac-policy-to-solidity-smart-contract, 2020. Online; accessed May 2020.

[113] Jonathan Shahen. Hard Instances of ATRBAC-Safety - Generator Source Code. https://git.uwaterloo.ca/jmshahen/hard-atrbac-instances, 2020. Online; accessed Jun 2020.

[114] Jonathan Shahen. Solidity Safety: Source Code. https://git.uwaterloo.ca/jmshahen/solidity-safety-to-model-checking, 2020. Online; accessed May 2020.

[115] Jonathan Shahen, Jianwei Niu, and Mahesh Tripunitara. Mohawk+t: Efficient analysis of administrative temporal role-based access control (atrbac) policies. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, pages 15–26, New York, NY, USA, 2015. ACM.

[116] Jonathan Shahen, Jianwei Niu, and Mahesh Tripunitara. Cree: a performant tool for safety analysis of administrative temporal role-based access control (atrbac) policies. *IEEE Transactions on Dependable and Secure Computing*, PP:1–1, 10 2019.

[117] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. Appsat: Approximately deobfuscating integrated circuits. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 95–100. IEEE, 2017.

[118] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. Cyclic obfuscation for creating sat-unresolvable circuits. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 173–178, 2017.

[119] Kaveh Shamsi, Meng Li, David Z Pan, and Yier Jin. Cross-lock: Dense layout-level interconnect locking using cross-bar architectures. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 147–152, 2018.

[120] Yuanqi Shen and Hai Zhou. Double dip: Re-evaluating security of logic encryption algorithms. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 179–184, 2017.

[121] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, pages 194–199, Washington, DC, USA, 2011. IEEE Computer Society.

[122] S Shinde, ZL Chua, V Narayanan, and P Saxena. Preventing your faults from telling your secrets. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS), Xian, China*, 2016.

[123] Deepak Sirone and Pramod Subramanayan. Functional analysis attacks on logic locking. *IEEE Transactions on Information Forensics and Security*, 2020.

[124] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts, 2019.

[125] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[126] Scott D. Stoller, Ping Yang, C R. Ramakrishnan, and Mikhail I. Gofman. Efficient policy analysis for administrative role based access control. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 445–455, New York, NY, USA, 2007. ACM.

[127] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 137–143. IEEE, 2015.

[128] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.

[129] Ethereum Team. Design Rationale. https://github.com/ethereum/wiki/wiki/Design-Rationale#gas-and-fees, 2019. Online; accessed May 2020.

[130] Mahesh Tripunitara and Ninghui Li. A theory for comparing the expressive power of access control models. *Journal of Computer Security*, 15(2):231–272, February 2007.

[131] A. Truong and D. H. T. That. Solving the user-role reachability problem in arbac with role hierarchy. In *2016 International Conference on Advanced Computing and Applications (ACOMP)*, pages 3–10, Nov 2016.

[132] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery.

[133] Emre Uzun, Vijayalakshmi Atluri, Shamik Sural, Jaideep Vaidya, Gennaro Parlato, Anna Lisa Ferrara, and Madhusudan Parthasarathy. Analyzing Temporal Role Based Access Control Models. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 177–186, New York, NY, USA, 2012. ACM.

[134] Emre Uzun, Vijayalakshmi Atluri, Jaideep Vaidya, Shamik Sural, Anna Ferrara, Gennaro Parlato, and P Madhusudan. Security analysis for temporal role based access control. *Journal of Computer Security*, 22:961–996, 06 2014.

[135] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 687–702, 2014.

[136] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 41–46, New York, NY, USA, 2011. ACM.

[137] S. A. Vavasis. Quadratic programming is in np. *Inf. Process. Lett.*, 36(2):73–77, October 1990.

[138] Vyperlang. Vyper - Pythonic Smart Contract Language for the EVM. https://github.com/vyperlang/vyper, 2020. Online; accessed May 2020.

[139] Yuepeng Wang, Shuvendu Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. Formal specification and verification of smart contracts for azure blockchain. April 2019.

[140] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93. IEEE, 2008.

[141] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[142] Yang Xie and Ankur Srivastava. Mitigating sat attack on logic locking. In *International conference on cryptographic hardware and embedded systems*, pages 127–146. Springer, 2016.

[143] Yang Xie and Ankur Srivastava. Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[144] Xiaolin Xu, Bicky Shakya, Mark M Tehranipoor, and Domenic Forte. Novel bypass attack and bdd-based tradeoff analysis against all known logic locking attacks. In *International conference on cryptographic hardware and embedded systems*, pages 189–210. Springer, 2017.

[145] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[146] Fangfei Yang, Ming Tang, and Ozgur Sinanoglu. Stripped functionality logic locking with hamming distance-based restore unit (sfll-hd)–unlocked. *IEEE Transactions on Information Forensics and Security*, 14(10):2778–2786, 2019.

[147] S. Yang. Logic synthesis and optimization benchmarks user guide: Version 3.0. Technical report, MCNC Technical Report, January 1991.

[148] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[149] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. Sarlock: Sat attack resistant logic locking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 236–241. IEEE, 2016.

[150] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Removal attacks on logic locking and camouflaging techniques. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[151] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Security analysis of anti-sat. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 342–347. IEEE, 2017.

[152] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2015.

[153] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan Rajendran, and Ozgur Sinanoglu. Provably-secure logic locking: From theory to practice. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1601–1618, 2017.

[154] Muhammad Yasin, Abhrajit Sengupta, Benjamin Carrion Schafer, Yiorgos Makris, Ozgur Sinanoglu, and Jeyavijayan Rajendran. What to lock? functional and parametric locking. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 351–356, 2017.

[155] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[156] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 990–1003, New York, NY, USA, 2014. ACM.

[157] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 827–838, New York, NY, USA, 2013. ACM.

[158] Yulong Zhang, Min Li, Kun Bai, Meng Yu, and Wanyu Zang. Incentive compatible moving target defense against vm-colocation attacks in clouds. In *Gritzalis, Dimitris and Furnell, Steven and Theoharidou, Marianthi (eds.) Information Security and Privacy Research*, pages 388–399, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[159] Hai Zhou. A humble theory and application for logic encryption. *IACR Cryptology ePrint Archive*, 2017:696, 2017.

[160] Hai Zhou, Ruifeng Jiang, and Shuyu Kong. Cycsat: Sat-based attack on cyclic logic encryptions. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 49–56. IEEE, 2017.

[161] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1371–1385, 2018.

[162] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. arXiv preprint, arXiv:1603.05615v1, 2016. http://arxiv.org/.

[163] Jian Zhu, Kai Hu, Mamoun Filali, Jean-Paul Bodeveix, and Jean-Pierre Talpin. Formal verification of solidity contracts in event-b, 2020.

APPENDICES

Appendix A

Summary of Software and Solvers

A summary of all tools and software created by myself are in Table A.1. Many projects in this thesis reduce one problem space to: ILP, Model Checking, or SAT. A summary of the public ILP and model checker solvers are provided in Table A.2.

Table A.1: Summary of Software Projects created by the author and used in this thesis. All software projects, except Mohawk+T, were created during PhD. Mohawk+T was updated and improved during PhD.

Software	Problem Space	Complexity Class	Project
Mohawk+T [106]	ATRBAC-Safety	PSPACE-Complete	ATRBAC-Safety
Cree [105]	ATRBAC-Safety	PSPACE-Complete	ATRBAC-Safety
ATRBAC Policy Statistics [109]	ATRBAC-Safety	P	Hard ATRBAC Instances
Vagabond [108]	VM Migrations	NP-Complete	Cloud Security
ILP Attack [111]	Logic Locked Circuits	NP-Complete	Logic Locking
Solidity Safety [114]	Ethereum Smart Contracts	PSPACE-Complete	Ethereum Safety
ATRBAC Smart Contract [112]	Ethereum Access Control	P	ATRBAC Smart Contracts

Table A.2: List of publically available solvers used in this thesis.

Solver	Problem Space	Complexity Class	Projects
NuSMV [78]	Model Checking	PSPACE-Complete	ATRBAC-Safety and Ethereum Safety
NuXmv [78]	Model Checking	PSPACE-Complete	ATRBAC-Safety and Ethereum Safety
Gurobi [45]	ILP	NP-Complete	Logic Locking
IBM ILOG CPLEX [51]	ILP	NP-Complete	Logic Locking and Cloud Security
Lingeling [14]	SAT	NP-Complete	Cloud Security
Plingeling [15]	SAT	NP-Complete	Cloud Security
Minisat [76]	SAT	NP-Complete	Cloud Security

Appendix B

NuSMV Supported Specifications

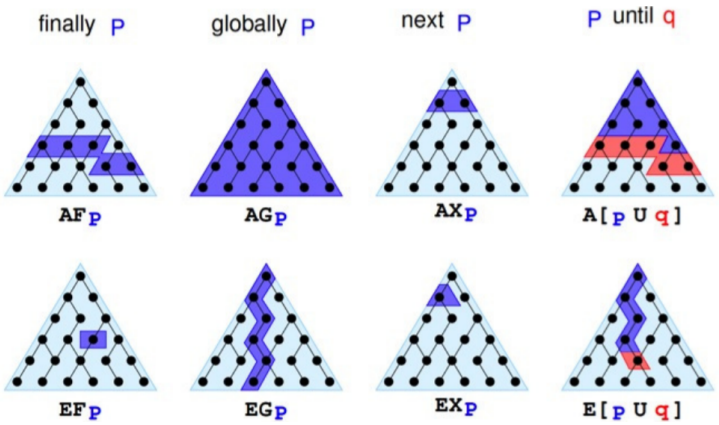


Figure B.1: Figure showing a representation of the CTL Specifications. Image Credit: Dr. Bonakdarpour http://web.cs.iastate.edu/~borzoo/teaching/15/CAS701/lectures/ctl.pdf.

B.1 Computational Tree Logic (CTL) Specifications

Computational Tree Logic (CTL) specifications are used in this thesis. Below we provide the grammar of CTL Specifications accepted by NuSMV v2.6 [78]. We also provide an explanation of the different specifications defined in CTL.

⟨*ctl-expr*⟩ ::= ⟨*boolean-expr*⟩
  | '`!`' ⟨*ctl-expr*⟩ *logical not*
  | '`(`' ⟨*ctl-expr*⟩ '`)`'
  | ⟨*ctl-expr*⟩ ('`&`' | '`|`' | '`xor`' | '`xnor`' | '`->`' | '`<->`') ⟨*ctl-expr*⟩
  | ('`EG`' | '`EX`' | '`EF`' | '`AG`' | '`AX`' | '`AF`') ⟨*ctl-expr*⟩
  | ('`E`' | '`A`') '`[`' ⟨*ctl-expr*⟩ '`U`' ⟨*ctl-expr*⟩ '`]`'

- **EX** p – is true in a state s if **there exists** a state s' such that a transition goes from s to s' and p is true in s'.
- **AX** p – is true in a state s if **for all** states s' where there is a transition from s to s' and p is true in s'.
- **EF** p – is true in a state s_0 if **there exists** a series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, ..., $s_{n-1} \rightarrow s_n$ such that p is true in s_n.
- **AF** p – is true in a state s_0 if for all series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, ... , $s_{n-1} \rightarrow s_n$, p is true in s_n.
- **EG** p – is true in a state s_0 if there exists an infinite series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, ... such that p is true in every s_i.
- **AG** p – is true in a state s_0 if for all infinite series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, ... p is true in every s_i.
- **E**$[p$ **U** $q]$ – is true in a state s_0 if there exists a series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, ..., $s_{n-1} \rightarrow s_n$ such that p is true in every state from s_0 to s_{n-1} and q is true in state s_n.
- **A**$[p$ **U** $q]$ – is true in a state s_0 if for all series of transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, ..., $s_{n-1} \rightarrow s_n$, p is true in every state from s_0 to s_{n-1} and q is true in state s_n.

B.2 Linear Temporal Logic (LTL) Specifications

Linear Temporal Logic (LTL) is another specification format. This format is not used in this thesis, but is included here for completeness. Below is the grammar of LTL Specifications accepted by NuSMV v2.6 [78].

⟨*bound*⟩ ::= '`[`' ⟨*int*⟩ '`,`' ⟨*int*⟩ '`]`'

⟨*ltl-expr*⟩ ::= ⟨*boolean-expr*⟩
| '`!`' ⟨*ltl-expr*⟩ *logical not*
| '`(`' ⟨*ltl-expr*⟩ '`)`'
| ⟨*ltl-expr*⟩ ('`&`' | '`|`' | '`xor`' | '`xnor`' | '`->`' | '`<->`') ⟨*ltl-expr*⟩
| ('`G`' | '`X`' | '`F`' | '`G`' ⟨*bound*⟩ | '`F`' ⟨*bound*⟩) ⟨*ltl-expr*⟩ *Future*
| ('`Y`' | '`Z`' | '`H`' | '`O`' | '`H`' ⟨*bound*⟩ | '`O`' ⟨*bound*⟩) ⟨*ltl-expr*⟩ *Past*
| ⟨*ltl-expr*⟩ ('`S`' | '`T`') ⟨*ltl-expr*⟩ *Since and Triggered*

- **X** p – is true at time t if p is true at time $t+1$.
- **F** p – is true at time t if p is true at some time $t' \geq t$.
- **F** $[l,u]p$ – is true at time t if p is true at some time $t+l \leq t' \leq t+u$.
- **G** p – is true at time t if p is true at all times $t' \geq t$.
- **G** $[l,u]$ p – is true at time t if p is true at all times $t+l \leq t' \leq t+u$.
- p **U** q – is true at time t if q is true at some time $t' \geq t$, and for all time t'' (such that $t \leq t'' < t'$) p is true.
- p **V** q is true at time t if q holds at all time steps $t' \geq t$ up to and including the time step t'' where p also holds. Alternatively, it may be the case that p never holds in which case q must hold in all time steps $t' \geq t$.
- **Y** p is true at time $t > t'$ if p holds at time $t-1$. Y p is false at time t_0.
- **Z** p is equivalent to Y p with the exception that the expression is true at time t_0.
- **H** p is true at time t if p holds in all previous time steps $t' \leq t$.
- **H** $[l,u]$ p is true at time t if p holds in all previous time steps $t-u \leq t' \leq t-l$.
- **O** p is true at time t if p held in at least one of the previous time steps $t' \leq t$.
- **O** $[l,u]$ p is true at time t if p held in at least one of the previous time steps $t-u \leq t' \leq t-l$.
- p **S** q is true at time t if q held at time $t' \leq t$ and p holds in all time steps t'' such that $t' < t'' \leq t$.
- p **T** q is true at time t if p held at time $t' \leq t$ and q holds in all time steps t'' such that $t' \leq t'0 \leq t$. Alternatively, if p has never been true, then q must hold in all time steps t'' such that $t_0 \leq t'' \leq t$

Appendix C

Cree Pruning and Bound Estimation

C.1 Static Slicing

Static pruning, implemented in Cree, has empirically shown to improve performance of the test cases shown in Section 3.6. There is a balance point where searching for more roles, rules, or timeslots to remove, takes longer than just reducing to Model Checking and running NuSMV. From this, we created two pruning techniques (modelled after those shown in [56]): forward and backwards pruning. Peudo-code for our forward and backward pruning algorithms are shown as Algorithm 3 and Algorithm 4. Both algorithms have polynomial run times.

ALGORITHM 3: Forward Pruning

Input : P – Mohawk+T Policy
// rule: $\langle a, L_a, C_t, S_t, t\rangle$
$R_{CA} \leftarrow \{t' | r \in P.\text{CA} \wedge r.t = t'\}$;
$R_{CE} \leftarrow \{t' | r \in P.\text{CE} \wedge r.t = t'\}$;
for $r \in \{P.CA \cup P.CR\}$ **do**
  if $(\exists c \in r.C_t | c \textit{ is } pos \wedge c \notin R_{CA})$ **or** $(\exists c, d \in r.C_t | c \textit{ is } pos \wedge d \textit{ is } neg \wedge c = d)$ **or** $(r.a \notin R_{CA})$ **or** $(r.a \notin R_{CE})$ **then**
    // Do not add rule to P'.
  else
    if $r \in P.CA$ **then** $P'.\text{CA} \leftarrow P'.\text{CA} \cup r$;
    else $P'.\text{CR} \leftarrow P'.\text{CR} \cup r$;
for $r \in \{P.CE \cup P.CD\}$ **do**
  if $(\exists c \in r.C_t | c \textit{ is } pos \wedge c \notin R_{CE})$ **or** $(\exists c, d \in r.C_t | c \textit{ is } pos \wedge d \textit{ is } neg \wedge c = d)$ **or** $(r.a \notin R_{CA})$ **or** $(r.a \notin R_{CE})$ **then**
    // Do not add rule to P'.
  else
    if $r \in CE$ **then** $P'.\text{CE} \leftarrow P'.\text{CE} \cup r$;
    else $P'.\text{CD} \leftarrow P'.\text{CD} \cup r$;
$P'.query \longleftarrow P.\text{query}$;
return P'

ALGORITHM 4: Backward Pruning

Input : P – Mohawk+T Policy
// rule: $\langle a, L_a, C_t, S_t, t\rangle$
1 $P''.\text{query} \leftarrow \langle R_q, s_q\rangle \leftarrow P.\text{query}$;
2 $S_{pos} \leftarrow R_{pos} \leftarrow \{g_1, g_2, \ldots, g_{|R_q|}\} \leftarrow R_q$;
3 $R_{neg}, R_{enb}, R_{dis}, S_{neg}, S_{enb}, S_{dis}, T_{neg}, T_{enb}, T_{dis} \leftarrow \{\}$;
4 $T_{pos} \leftarrow \{g_1 : \{s_q\}, g_2 : \{s_q\}, \ldots, g_{|R_q|} : \{s_q\}\}$;
5 **while** $(S_{pos} \neq \emptyset \bigvee S_{neg} \neq \emptyset \bigvee S_{enb} \neq \emptyset \bigvee S_{dis} \neq \emptyset)$ **do**
6 $s_{pos} \leftarrow S_{pos}.\text{pop}()$;
7 **for** $\{r | r \in P.CA \wedge r.t = s_{pos} \wedge r.S_t \cap T_{pos}[s_{pos}] \neq \emptyset\}$ **do**
8 **for** $\{p | p \in r.C_t \wedge p \textit{ is } pos \wedge p \notin R_{pos}\}$ **do**
9 $S_{pos} \leftarrow S_{pos} \cup p$; $R_{pos} \leftarrow R_{pos} \cup p$;
10 $T_{pos}[p] \leftarrow T_{pos}[p] \cup r.S_t$
11 **for** $\{n | n \in r.C_t \wedge n \textit{ is negative} \wedge n \notin R_{neg}\}$ **do**
12 $S_{neg} \leftarrow S_{neg} \cup n$; $R_{neg} \leftarrow R_{neg} \cup n$;
13 $T_{neg}[n] \leftarrow T_{neg}[n] \cup r.S_t$
14 **if** $r.a \neq TRUE$ **then**
15 $T_{pos}[r.a] \leftarrow T_{pos}[r.a] \cup r.L_a$;
16 $T_{enb}[r.a] \leftarrow T_{enb}[r.a] \cup r.L_a$;
17 **if** $(r.a \notin R_{pos})$ **then**
18 $S_{pos} \leftarrow S_{pos} \cup r.a$; $R_{pos} \leftarrow R_{pos} \cup r.a$
19 **if** $(r.a \notin R_{enb})$ **then**
20 $S_{enb} \leftarrow S_{enb} \cup r.a$; $R_{enb} \leftarrow R_{enb} \cup r.a$
21 **Run Lines 6-20** for: S_{neg} and $P.CR$;
22 **Run Lines 6-20** for: S_{enb} and $P.CE$;
23 **Run Lines 6-20** for: S_{dis} and $P.CD$;
24 $P''.CA \leftarrow \{r | r \in P.CA \bigwedge r.t \in R_{pos} \bigwedge r.S_t \bigcap T_{pos}[r.t] \neq \emptyset\}$;
25 $P''.CR \leftarrow \{r | r \in P.CR \bigwedge r.t \in R_{neg} \bigwedge r.S_t \bigcap T_{neg}[r.t] \neq \emptyset\}$;
26 $P''.CE \leftarrow \{r | r \in P.CE \bigwedge r.t \in R_{enb} \bigwedge r.S_t \bigcap T_{enb}[r.t] \neq \emptyset\}$;
27 $P''.CD \leftarrow \{r | r \in P.CD \bigwedge r.t \in R_{dis} \bigwedge r.S_t \bigcap T_{dis}[r.t] \neq \emptyset\}$;
28 **return** P'';

```
// Path: CE1(admin) → CE3(a) → CA6(a) → CA6(user) → CA4(u) → CR2(u) → CA2(u) → CA6(u)
Query : t2, [r3, r4]
CanAssign:
  /*CA1*/ <TRUE, t1-t3, TRUE, [t2,t3], r1>
  /*CA2*/ <r3, t1-t3, r2 & NOT r3, [t2,t3], r4>
  /*CA3*/ <r1, t1-t3, r1, [t2,t3], r5>
  /*CA4*/ <r3, t1-t3, r3, [t1], r2>
  /*CA5*/ <r1, t1-t3, r5 & NOT r3, [t2,t3], r6>
  /*CA6*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r3>
CanRevoke:
  /*CR1*/ <TRUE, t1-t3, TRUE, [t1,t2], r1>
  /*CR2*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r3>
  /*CR3*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r2>
CanEnable:
  /*CE1*/ <TRUE, t1-t2, TRUE, [t1], r1>
  /*CE2*/ <TRUE, t1-t2, TRUE, [t2], r1>
  /*CE3*/ <TRUE, t1-t2, r1 & NOT r2, [t1], r3>
  /*CE4*/ <TRUE, t1-t2, r1, [t1], r2>
CanDisable:
  /*CD1*/ <TRUE, t1-t2, TRUE, [t1], r1>
  /*CD2*/ <TRUE, t1-t2, TRUE, [t1], r2>
```

Figure C.1: An example that shows rules/roles/timeslots, that can be removed without changing the security of the policy. Removing all unnecessary rules/roles/timeslots with static pruning ensures: (1) the pruned policy's size will be less than or equal to the input policy's size, and (2) the safety response remains unchanged.

C.2 Bound Estimation Example

Using Figure 3.2, we will show how to calculate each of the bound estimation steps. We have recreated that example here in Figure C.1.

C.2.1 Initial Upper Bound

User-Role-Timeslot Assignment:

$$\begin{aligned} u &= |\text{Users}| \cdot |\text{Roles}| \cdot |\text{Timeslots}| \\ &= (|\text{Roles}| \cdot |\text{Timeslots}|) \cdot |\text{Roles}| \cdot |\text{Timeslots}| \end{aligned}$$

Role-Timeslot Enablement:

$$r = |\text{Roles}| \cdot |\text{Timeslots}|$$

Upper Bound:

$$d_0 = 2^{u+r} = 2^{(|\text{Roles}|^2 \cdot |\text{Timeslots}|^2)+(|\text{Roles}| \cdot |\text{Timeslots}|)}$$

Applying to Figure C.1:

$$\begin{aligned} \text{Roles} &= \{r1, r2, r3, r4, r5, r6\} \\ \text{Timeslots} &= \{t1, t2, t3\} \\ d_0 &= 2^{(6^2 \cdot 3^2)+(6 \cdot 3)} = 2^{342} \end{aligned}$$

C.2.2 Tightening 1

$$|\text{Users}| = |\text{Admin Roles}| + 1$$
$$d_1 = 2^{(|\text{Admin Roles}|+2)\cdot|\text{Roles}|\cdot|\text{Timeslots}|}$$

Applying to Figure C.1:

$$\begin{aligned}\text{Roles} &= \{r1, r2, r3, r4, r5, r6\}\\ \text{Timeslots} &= \{t1, t2, t3\}\\ \text{Admin Roles} &= \{r1, r3\}\\ d_1 &= 2^{(2+2)\cdot 6\cdot 3} = 2^{72}\end{aligned}$$

C.2.3 Tightening 2

User-Role-Timeslot Assignment:

$$\begin{aligned}|u_2| =&(|\text{Admin-Roles}| + 1) \cdot |\text{CA-Target-Roles} \cap (\text{CA-CR-Positive-Precondition} \cup \text{Admin-Roles}\\ &\cup \text{Goal-Roles})| \cdot |\text{CA-Target-Timeslots}|\end{aligned}$$

Role-Timeslot Enablement:

$$|r_2| = |\text{CE-Target-Roles} \cap (\text{CE-CD-Positive-Precondition} \cup \text{Admin-Roles})| \cdot |\text{CE-Target-Timeslots}|$$

Upper Bound:

$$d_2 = 2^{|u_2|+|r_2|}$$

Applying to Figure C.1:

$$\begin{aligned}\text{CA-Target-Roles} &= \{r1, r2, r3, r4, r5, r6\}\\ \text{CA-CR-Positive-Precondition} &= \{r1, r2, r3, r5\}\\ \text{CE-Target-Roles} &= \{r1, r2, r3\}\\ \text{CE-CD-Positive-Precondition} &= \{r1\}\\ \text{Goal-Roles} &= \{r3, r4\}\\ \text{Admin Roles} &= \{r1, r3\}\\ u_2 &= (2+1) \cdot |\{r1, r2, r3, r4, r5\}| \cdot |\{t1, t2, t3\}| = 45\\ r_2 &= |\{r1, r3\}| \cdot |\{t1, t2\}| = 4\\ d_2 &= 2^{|u_2|+|r_2|} = 2^{49}\end{aligned}$$

C.2.4 Tightening 3

$$U_{goal} = \left| \text{CA-Target-Roles} \bigcap \left[\bigcup_{g \in \text{Goal-Roles}} \text{Get-Roles}[LSP_{CA,CR}(g)] \right] \right| \cdot \left| \bigcup_{g \in \text{Goal-Roles}} \text{Get-Timeslots}[LSP_{CA,CR}(g)] \right|$$

$$U_{admins} = \left[\sum_{a \in \text{Admin-Roles}} |\text{CA-Target-Roles} \cap \text{Get-Roles}[LSP_{CA,CR}(a)]| \cdot |\text{Get-Timeslots}[LSP_{CA,CR}(a)]| \right]$$

$$RE = \left| \text{CE-Target-Roles} \bigcap \left[\bigcup_{a \in \text{Admin-Roles}} \text{Get-Roles}[LSP_{CE,CD}(a)] \right] \right| \cdot \left| \bigcup_{a \in \text{Admin-Roles}} \text{Get-Timeslots}[LSP_{CE,CD}(a)] \right|$$

$$d_3 = 2^{U_{goal}+U_{admins}+RE}$$

Applying to Figure C.1:

$$\text{CA-Target-Roles} = \{r1, r2, r3, r4, r5, r6\}$$
$$\text{CE-Target-Roles} = \{r1, r2, r3\}$$
$$\text{Goal-Roles} = \{r3, r4\}$$
$$\text{Admin Roles} = \{r1, r3\}$$
$$LSP_{CA,CR}(r1) = \{CA1\}$$
$$LSP_{CA,CR}(r3) = \{CA6\}$$
$$LSP_{CA,CR}(r4) = \{CA2, CA4, CA6, CR2\}$$
$$LSP_{CE,CD}(r1) = \{CE2\}$$
$$LSP_{CE,CD}(r3) = \{CE3, CE1\}$$
$$U_{goal} = |\{r1, r2, r3, r4, r5, r6\} \cap \{r3, r2, r4\}| \cdot |\{t1, t2, t3\}| = 9$$
$$U_{admins} = (|\{r1, r2, r3, r4, r5, r6\} \cap \{r1\}| \cdot |\{t2, t3\}| + |\{r1, r2, r3, r4, r5, r6\} \cap \{r3\}|) \cdot |\{t1, t2, t3\}| = 5$$
$$RE = |\{r1, r2, r3\} \cap [\{r1\} \cup \{r1, r3\}]| \cdot |\{t2\} \cup \{t1\}| = 4$$
$$d_3 = 2^{9+5+4} = 2^{18}$$

The $LSP_{A,B}(r)$ function uses A as a positive rule set (either $t\_\ can\_\ assign$ or $t\_\ can\_\ enable$) and B as the negative rule set ($t\_\ can\_\ revoke$ or $t\_\ can\_\ disable$). The function returns the longest length path, from the set of shortest rule paths starting from r till all conditions are satisfied. We start the LSP function with a rule in A that has r as its target role. We then branch out from this using an algorithm found in abstraction refinement to build a dependency graph that includes all rules. If there exists more than one rule in A that has r as the target role, then LSP returns the longest list of rules.

The `Get-Roles` function returns a set of target roles from a set of rules. The `Get-Timeslots` function returns a set of the target timeslots from a set of rules.

Appendix D

Cree Empirical Results without Mohawk+T

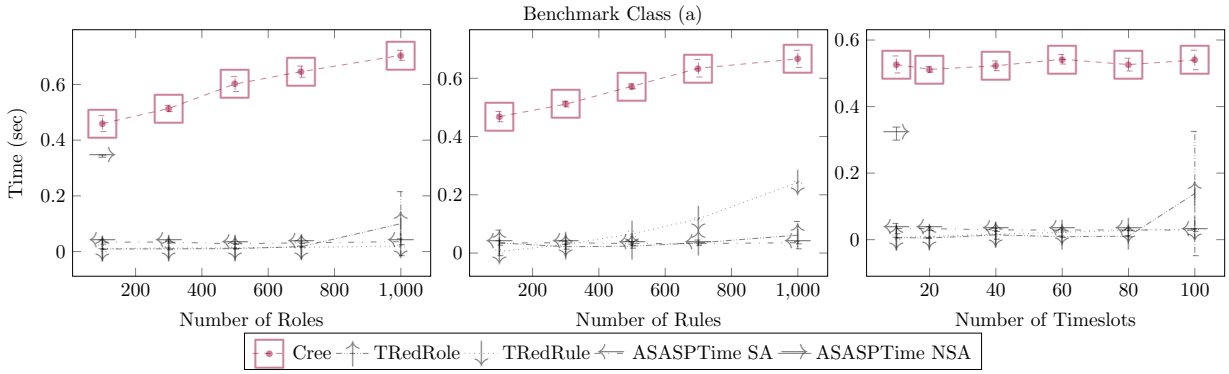


Figure D.1: Results from all tools for Benchmark Class (a). It comprises random input instances from a generator from Uzun et al. [133]. The curves interpolate averages, and the error-bars show the standard deviation. Mohawk+T has been removed.

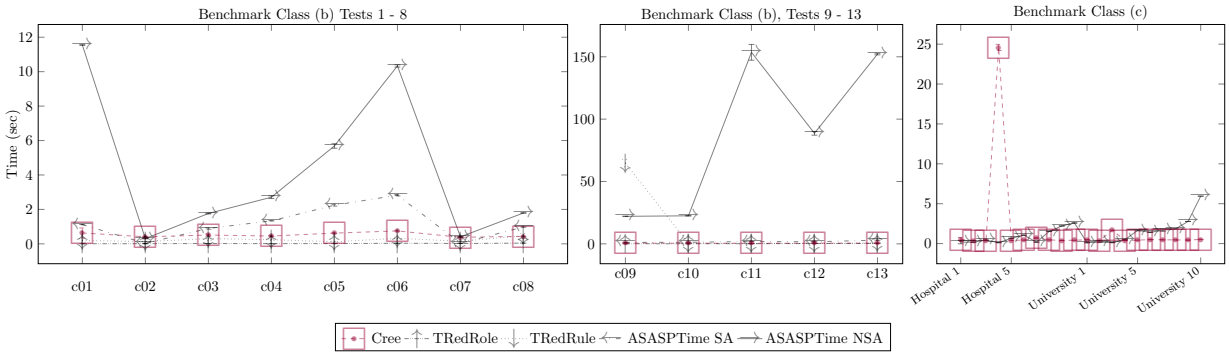


Figure D.2: Results from Benchmark Class (b) (two graphs to the left), and Benchmark Class (c) (right). These comprise input instances from the work of Ranise et al. [93]. Mohawk+T has been removed.

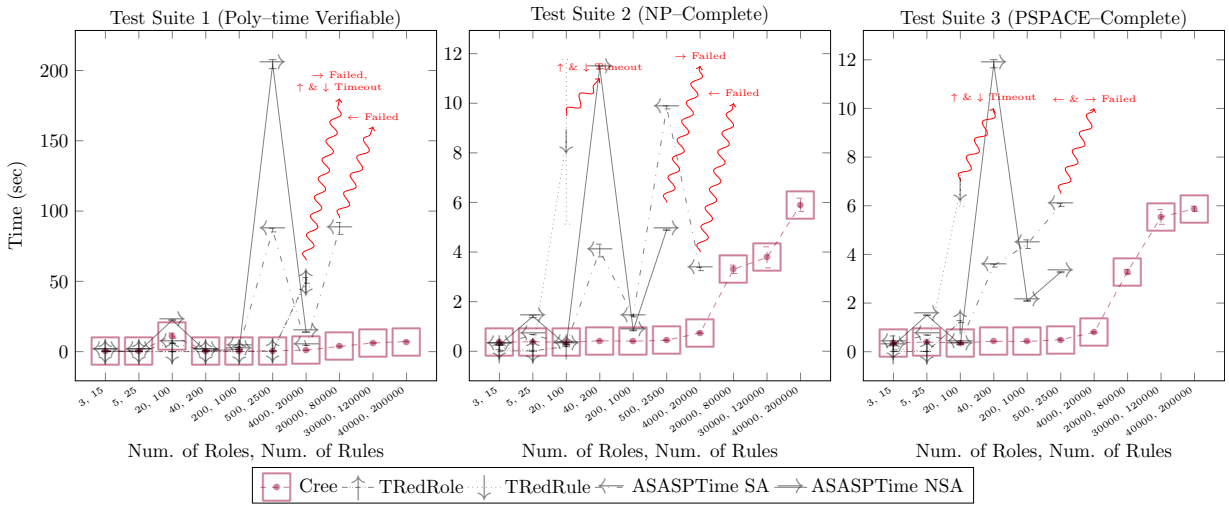


Figure D.3: Mohawk inputs [56] converted to ATRBAC-Safety instances using one time-slot. Test Suite 1 are inputs with non-negated preconditions only. Test Suite 2 are inputs with no revoke rules. Test Suite 3 are both positive and negated preconditions, and assign/revoke rules. Red wavy lines are tools that crashed/timed-out during testing for certain input sizes. Mohawk+T has been removed.

Appendix E

Analysis of the New Datasets Created in Chapter 4

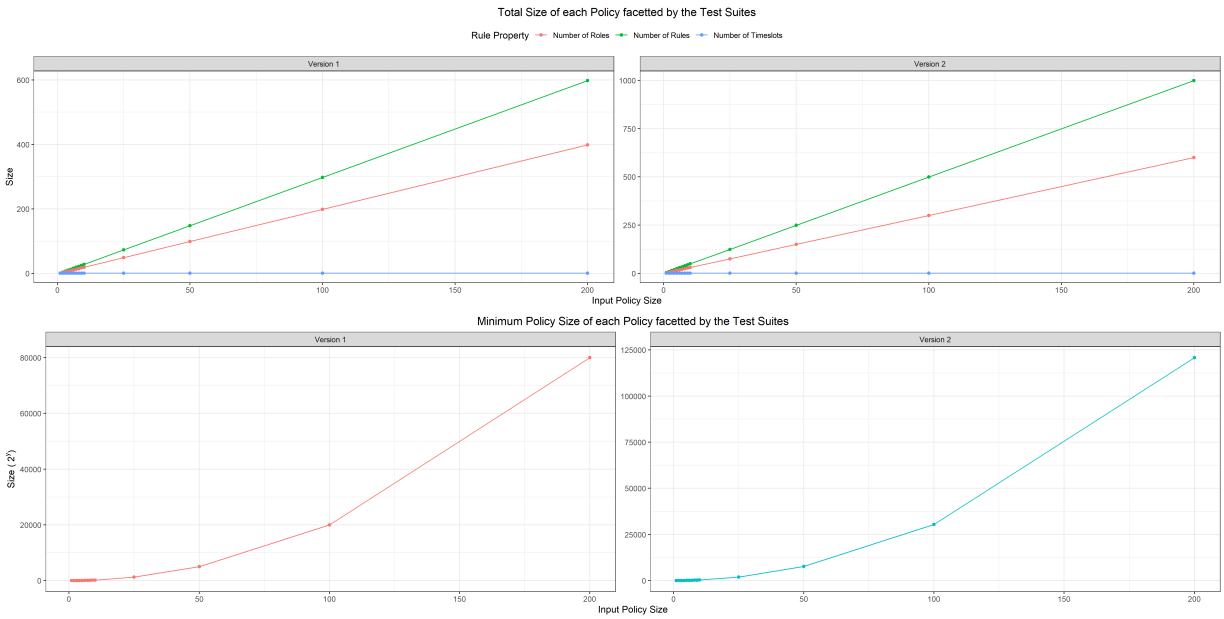


Figure E.1: Policy size, using rule/roles/timeslots and minimum simple path, for the new datasets.

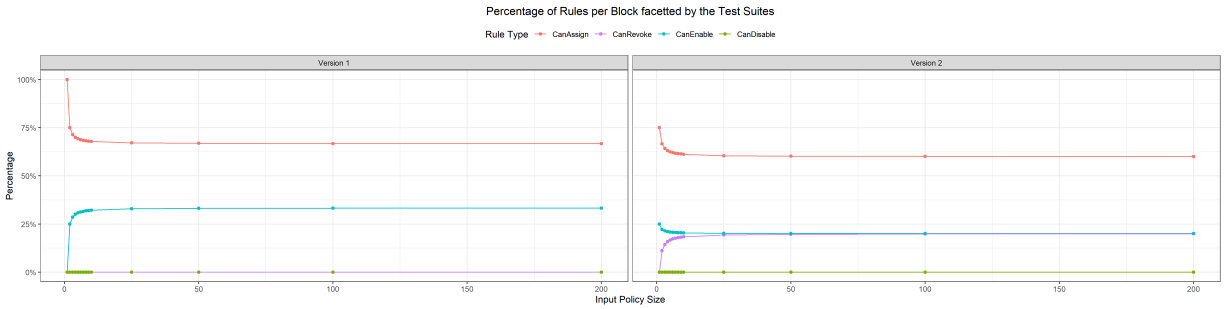


Figure E.2: The distribution of rules by type: Can Assign, Can Revoke, Can Enable, Can Disable.

Figure E.3: Number of rules which are: Truly Startable, Startable, Invokable, and Unassignable Precondition.

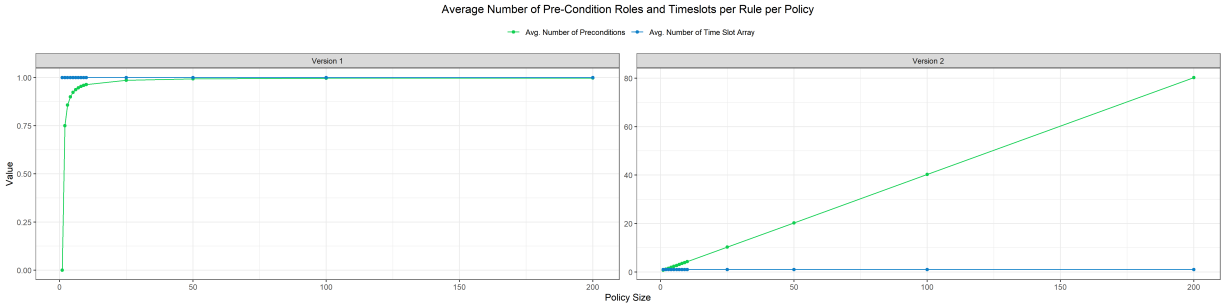


Figure E.4: Average number of roles in the precondition and average number of time slots in the target time slot array.

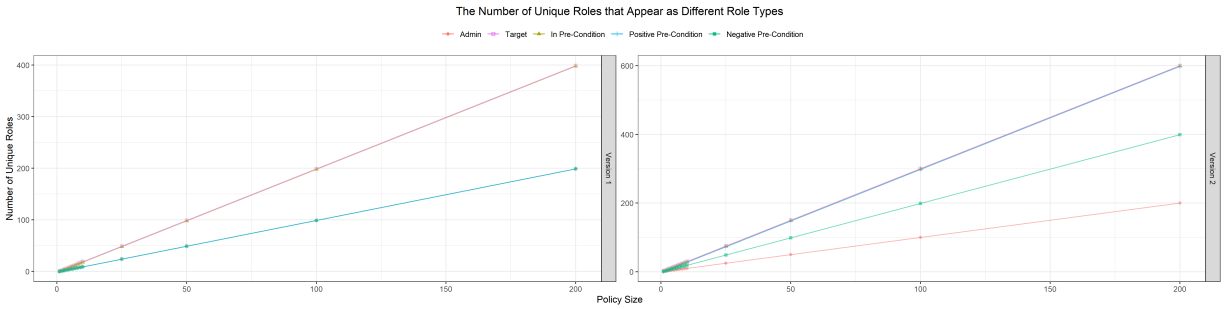


Figure E.5: The number of unique roles that appear for each role type.

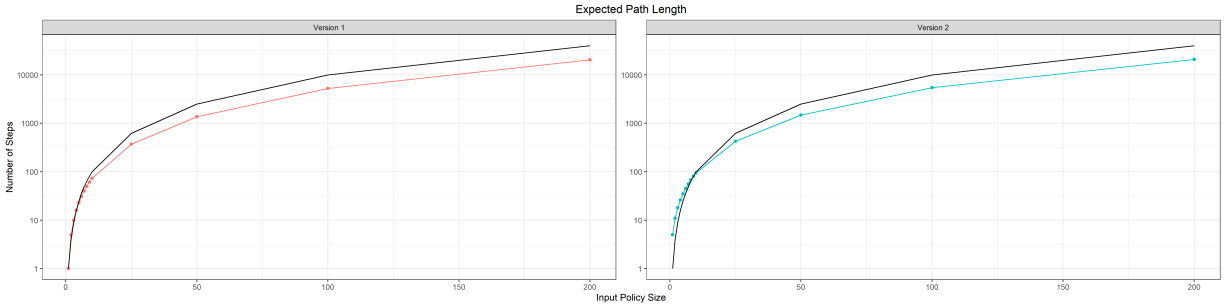


Figure E.6: The expected path length required to reach a satisfiable state. This is shown with a log vertical scale. Black line is x^2, where the x is the input policy size.

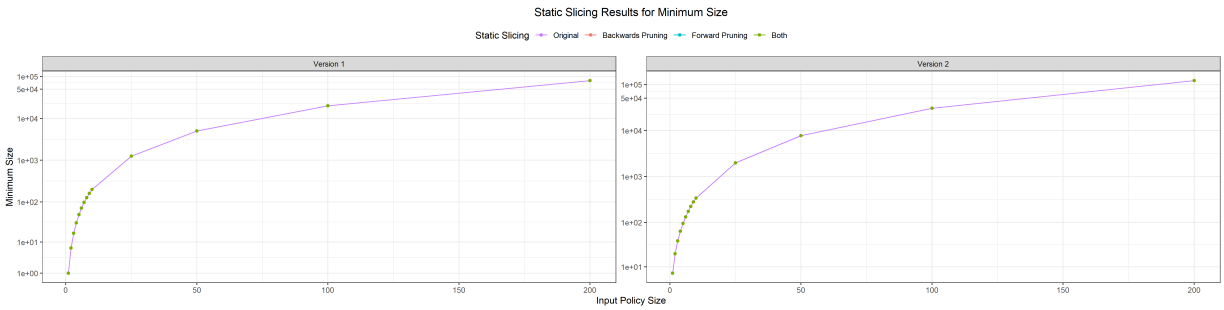


Figure E.7: The measured minimum size of each policy file after each pruning technique is performed.

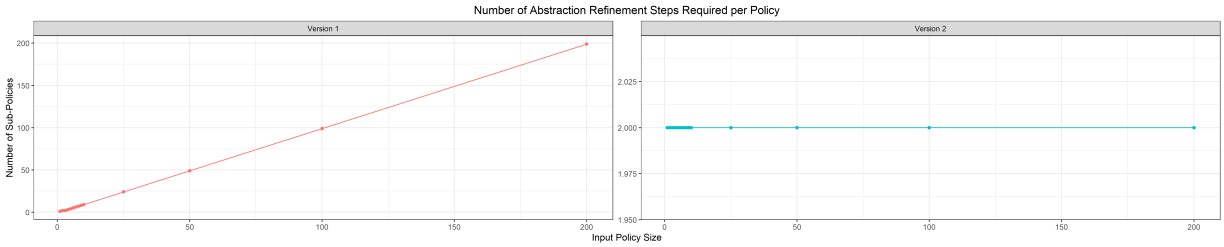


Figure E.8: The number of required abstraction refinement steps used to solve each policy from the new datasets.

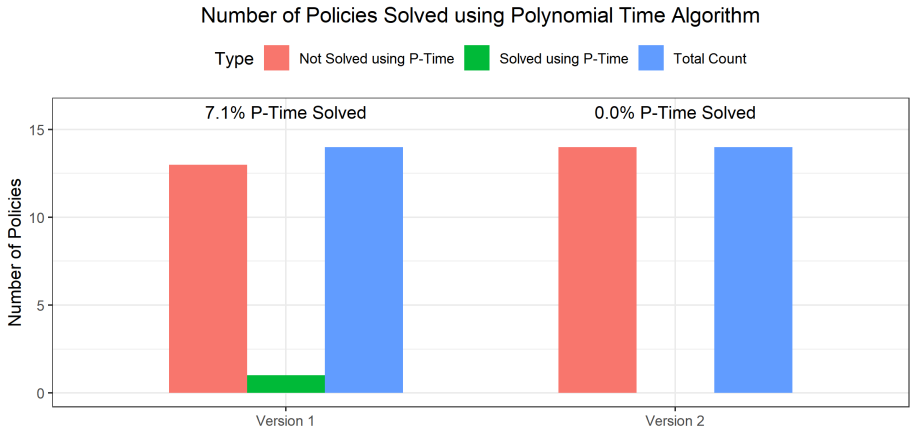


Figure E.9: Number of policies solved using the Cree's polynomial time algorithm.

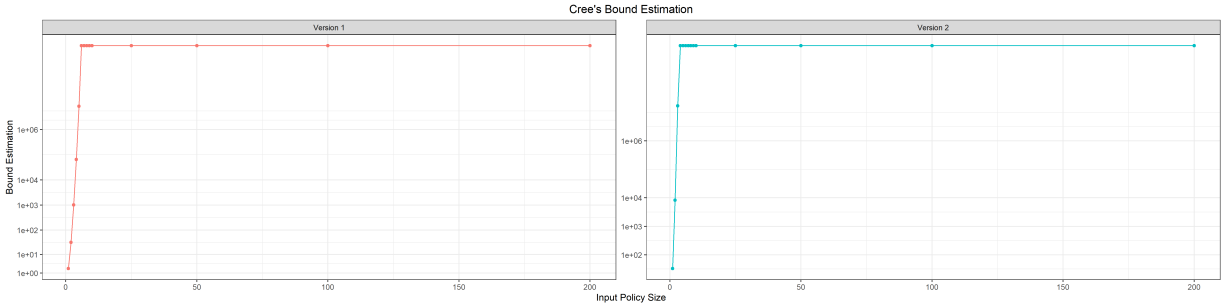


Figure E.10: Cree's Bound Estimation for new datasets.

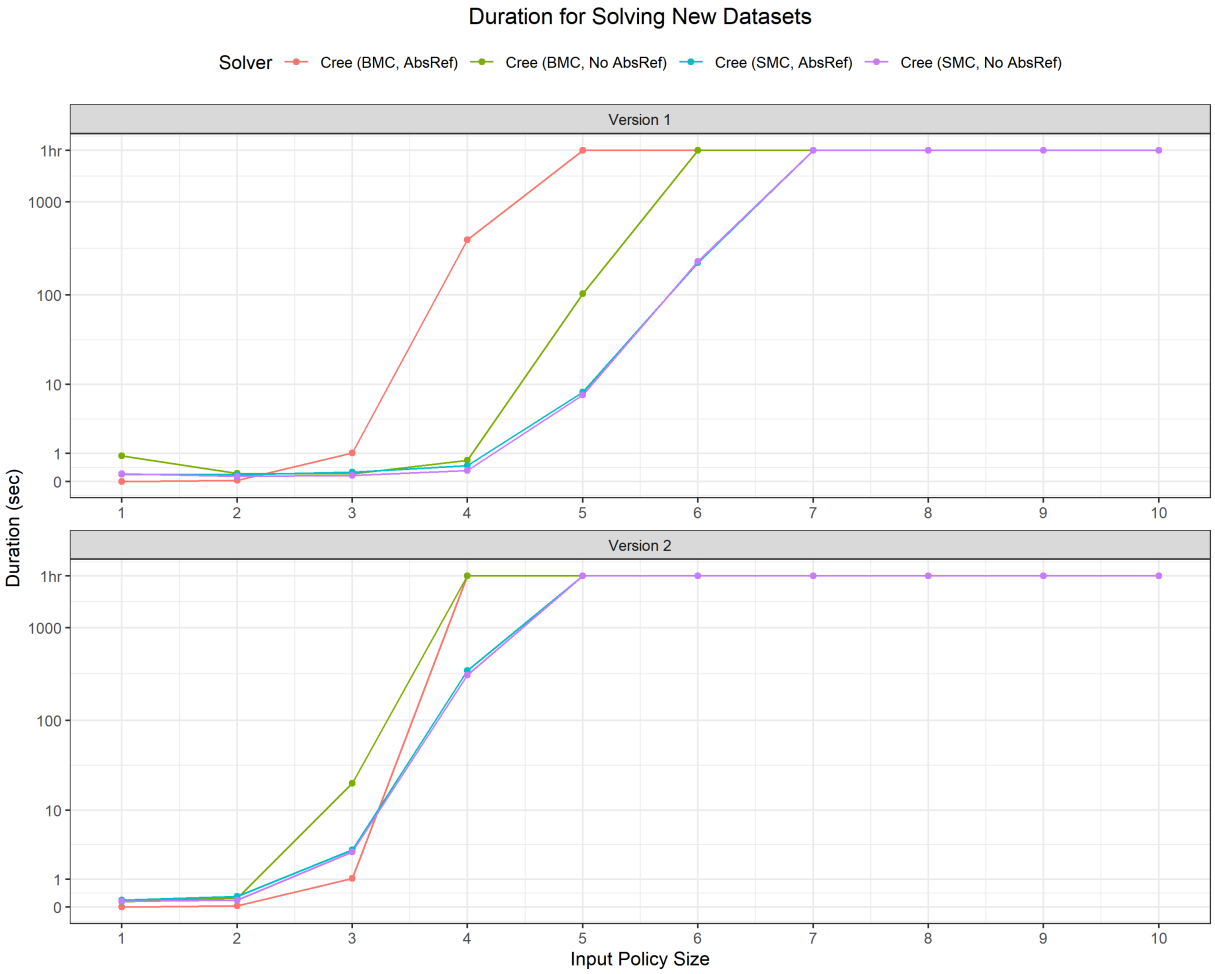


Figure E.11: Duration for Cree to solve the new datasets given 1hr time limit.

Appendix F

Generic ATRBAC Smart Contract

```
pragma solidity ^0.5.0;

import "./RoleTimeslots.sol";
import "./Rule.sol";
import "./Timeslot.sol";

/**
 * @author Jonathan Shahen
 * @notice Implements the Administrative Temporal Role Based Access Control empirical
     version described
 *      in the Mohawk+T paper in SACMAT'15 (http://dx.doi.org/10.1145/2752952.2752966)*/
contract ATRBAC {
  using RoleTimeslots for RoleTimeslots.Role;
  using Timeslot for Timeslot.timeslot;
  using Rule for Rule.rule;

  bool constant IGNORE_TIME_DEBUG = true;
  /** the current state of the access control policy.
    We use this to determine if a rule can fire and if a user satisfies a condition.*/
  RoleTimeslots.Role[] public trbac_state;

  /** stores wether a role is enabled for a particular timeslot.
    If a role is enabled, then that allows rules to be fired for the admin condition.*/
  mapping(uint => mapping(uint => bool)) public role_enablement;
  /** Can Assign rules allow for changes to the TRBAC State by adding role/timeslots to a
      target user.*/
  Rule.rule[] ca_rules;
  /** Can Revoke rules allow for changes to the TRBAC State by removing role/timeslots
      from a target user.*/
  Rule.rule[] cr_rules;
  /** Can Enable rules allow for changes to the Role Enablement table by enabling role/
      timeslots.*/
  Rule.rule[] ce_rules;
  /** Can Disable rules allow for changes to the Role Enablement table by disabling role/
      timeslots.*/
  Rule.rule[] cd_rules;
  enum RuleType {CanAssign, CanRevoke, CanEnable, CanDisable}
```

```
/**
 * @title A list of all the timeslots in the policy
 * @notice A timeslot must be added before it is referenced in a rule's time interval or
      timeslots array
 */
Timeslot.timeslot[] public timeslots;

/** The list of users who can DO ANYTHING to the ATRBAC policy
 * SECURITY: Limit this list to the MINIMUM number of users.*/
mapping(address=>bool) public super_users;

// CONSTRUCTOR AND EVENTS
constructor() public {
  super_users[msg.sender] = true;
  suAddNewRole("TRUE"); // Add the TRUE role (assigned to all users and enabled for all
      timeslots)
}
event SuperUser_AddSuperUser(address _super_user, address _new_super_user);
event SuperUser_RemoveSuperUser(address _super_user, address _new_super_user);

// Modifiers
modifier onlySuperUsers {
  require(super_users[msg.sender] == true, "Sender must be a Super User.");
  _;
}
modifier noZeroAddress(address account) {
  require(account != address(0), "Address cannot be the zero address");
  _;
}
modifier roleExists(uint _role) {
  require(_role < trbac_state.length, "Role does not exists.");
  _;
}
modifier timeslotExists(uint _timeslot) {
  require(_timeslot < timeslots.length, "Timeslot does not exists.");
  _;
}

// Super User Actions
function suAddSuperUser(address _super_user) public onlySuperUsers noZeroAddress(
    _super_user) returns(bool){
  require(super_users[_super_user] == false, "User is already a Super User.");
  super_users[_super_user] = true;

  emit SuperUser_AddSuperUser(msg.sender, _super_user);
  return true;
}
function suRemoveSuperUser(address _super_user) public onlySuperUsers noZeroAddress(
    _super_user) returns(bool){
  require(super_users[_super_user] == true, "User is not a Super User.");
  super_users[_super_user] = false;

  emit SuperUser_RemoveSuperUser(msg.sender, _super_user);
  return true;
}
function suAddRoleTimeslotToUser(address _user, uint _role, uint _timeslot) public
```

```
    onlySuperUsers noZeroAddress(_user) roleExists(_role) returns(bool){
  trbac_state[_role].add(_timeslot, _user);
  return true;
}
function suRemoveRoleTimeslotToUser(address _user, uint _role, uint _timeslot) public
    onlySuperUsers noZeroAddress(_user) roleExists(_role) returns(bool){
  trbac_state[_role].remove(_timeslot, _user);
  return true;
}
function suAddTimeSlot(uint _start_hour, uint _end_hour) public onlySuperUsers returns(
    uint) {
  require(_start_hour <= _end_hour, "The Start Hour must be less than or equal to the
      End Hour.");
  require(0 <= _start_hour && _start_hour <= 23, "The Start Hour must be in the range
      [0,23]");
  require(0 <= _end_hour && _end_hour <= 23, "The End Hour must be in the range [0,23]")
      ;

  timeslots.push(Timeslot.timeslot(_start_hour, _end_hour));
  return timeslots.length - 1;
}
function suAddNewRole(string memory _name) public onlySuperUsers returns(uint) {
  require(bytes(_name).length > 0, "The Role's name must be non-zero length");

  trbac_state.push(RoleTimeslots.Role(_name));
  return trbac_state.length - 1;
}
function suEnableRole(uint _role, uint _timeslot) public
  onlySuperUsers roleExists(_role) timeslotExists(_timeslot) returns(bool) {
  role_enablement[_role][_timeslot] = true;
  return true;
}
function suDisableRole(uint _role, uint _timeslot) public
  onlySuperUsers roleExists(_role) timeslotExists(_timeslot) returns(bool) {
  role_enablement[_role][_timeslot] = false;
  return true;
}

/**
 * @notice Adds a new Can Assign rule to the policy
 * @param _adminRole The role that will act as administrator (0 for TRUE)
 * @param _adminTimeIntervalStart The index of the timeslot to start the interval
 * @param _adminTimeIntervalEnd The index of the timeslot to end the interval
 * @param _precondition List of role indexes, negative role indexes indicate a negative
     precondition
 * @param _targerTimeSlotArray List of timeslot indexes that the target user/role must
     satisfy
 * @param _targetRole Index of the target role
 */
function suAddCARule(uint _adminRole, uint _adminTimeIntervalStart, uint
    _adminTimeIntervalEnd,
      int[] memory _precondition, uint[] memory _targerTimeSlotArray, uint _targetRole)
          public
      roleExists(_adminRole) roleExists(_targetRole)
      timeslotExists(_adminTimeIntervalStart) timeslotExists(_adminTimeIntervalEnd) {
  require(_adminTimeIntervalStart <= _adminTimeIntervalEnd,
    "The Admin Time Interval Start index must be <= to the End index");
```

```
    // Verify Precondition
    PreconditionRole memory r;
    for(uint i = 0; i < _precondition.length; i++) {
      r = splitRolePrecondition(_precondition[i]);
      require(r.role < trbac_state.length, "Role does not exists.");
    }
    // Verify Timeslot Array
    for(uint i = 0; i < _targerTimeSlotArray.length; i++) {
      require(_targerTimeSlotArray[i] < timeslots.length, "Timeslot does not exists.");
    }

    ca_rules.push(Rule.rule(_adminRole, _adminTimeIntervalStart, _adminTimeIntervalEnd,
        _precondition,
      _targerTimeSlotArray, _targetRole));
  }
  function suAddCRRule(uint _adminRole, uint _adminTimeIntervalStart, uint
      _adminTimeIntervalEnd,
        int[] memory _precondition, uint[] memory _targerTimeSlotArray, uint _targetRole)
            public
        roleExists(_adminRole) roleExists(_targetRole)
        timeslotExists(_adminTimeIntervalStart) timeslotExists(_adminTimeIntervalEnd) {
    require(_adminTimeIntervalStart <= _adminTimeIntervalEnd,
      "The Admin Time Interval Start index must be <= to the End index");

    // Verify Precondition
    PreconditionRole memory r;
    for(uint i = 0; i < _precondition.length; i++) {
      r = splitRolePrecondition(_precondition[i]);
      require(r.role < trbac_state.length, "Role does not exists.");
    }
    // Verify Timeslot Array
    for(uint i = 0; i < _targerTimeSlotArray.length; i++) {
      require(_targerTimeSlotArray[i] < timeslots.length, "Timeslot does not exists.");
    }

    cr_rules.push(Rule.rule(_adminRole, _adminTimeIntervalStart, _adminTimeIntervalEnd,
        _precondition,
      _targerTimeSlotArray, _targetRole));
  }
  function suAddCERule(uint _adminRole, uint _adminTimeIntervalStart, uint
      _adminTimeIntervalEnd,
        int[] memory _precondition, uint[] memory _targerTimeSlotArray, uint _targetRole)
            public
        roleExists(_adminRole) roleExists(_targetRole)
        timeslotExists(_adminTimeIntervalStart) timeslotExists(_adminTimeIntervalEnd) {
    require(_adminTimeIntervalStart <= _adminTimeIntervalEnd,
      "The Admin Time Interval Start index must be <= to the End index");

    // Verify Precondition
    PreconditionRole memory r;
    for(uint i = 0; i < _precondition.length; i++) {
      r = splitRolePrecondition(_precondition[i]);
      require(r.role < trbac_state.length, "Role does not exists.");
    }
    // Verify Timeslot Array
    for(uint i = 0; i < _targerTimeSlotArray.length; i++) {
```

```
    require(_targerTimeSlotArray[i] < timeslots.length, "Timeslot does not exists.");
  }

  ce_rules.push(Rule.rule(_adminRole, _adminTimeIntervalStart, _adminTimeIntervalEnd,
      _precondition,
    _targerTimeSlotArray, _targetRole));
}
function suAddCDRule(uint _adminRole, uint _adminTimeIntervalStart, uint
    _adminTimeIntervalEnd,
      int[] memory _precondition, uint[] memory _targerTimeSlotArray, uint _targetRole)
          public
      roleExists(_adminRole) roleExists(_targetRole)
      timeslotExists(_adminTimeIntervalStart) timeslotExists(_adminTimeIntervalEnd) {
  require(_adminTimeIntervalStart <= _adminTimeIntervalEnd,
    "The Admin Time Interval Start index must be <= to the End index");

  // Verify Precondition
  PreconditionRole memory r;
  for(uint i = 0; i < _precondition.length; i++) {
    r = splitRolePrecondition(_precondition[i]);
    require(r.role < trbac_state.length, "Role does not exists.");
  }
  // Verify Timeslot Array
  for(uint i = 0; i < _targerTimeSlotArray.length; i++) {
    require(_targerTimeSlotArray[i] < timeslots.length, "Timeslot does not exists.");
  }

  cd_rules.push(Rule.rule(_adminRole, _adminTimeIntervalStart, _adminTimeIntervalEnd,
      _precondition,
    _targerTimeSlotArray, _targetRole));
}

function hasAccess(address _user, uint _role, uint _timeslot) public view
  roleExists(_role) timeslotExists(_timeslot) returns(bool) {
  return trbac_state[_role].has(_timeslot, _user);
}
function getRoleName(uint _role) public view roleExists(_role) returns(string memory) {
  return trbac_state[_role].name;
}
function roleEnabled(uint _role, uint _timeslot) public view
  roleExists(_role) timeslotExists(_timeslot) returns(bool) {
  return role_enablement[_role][_timeslot];
}
function getLastRoleId() public view returns(uint) {
  return trbac_state.length - 1;
}
function getNextRoleId() public view returns(uint) {
  return trbac_state.length;
}

// TIMESLOTS
function getTimeslotHours(uint _timeslot) public view timeslotExists(_timeslot) returns(
    uint[2] memory) {
  return([timeslots[_timeslot].start_hour, timeslots[_timeslot].end_hour]);
}
function getLastTimeslotId() public view returns(uint) {
  return timeslots.length - 1;
```

```
}
function getNextTimeslotId() public view returns(uint) {
  return timeslots.length;
}

// HELPER FUNCTIONS
function getCanAssign() public pure returns(uint) { return uint(RuleType.CanAssign); }
function getCanRevoke() public pure returns(uint) { return uint(RuleType.CanRevoke); }
function getCanEnable() public pure returns(uint) { return uint(RuleType.CanEnable); }
function getCanDisable() public pure returns(uint) { return uint(RuleType.CanDisable); }

struct PreconditionRole {
  uint role;
  bool not;
}

function splitRolePrecondition(int _role) internal pure returns(PreconditionRole memory)
    {
  uint role = (_role < 0)? uint(-1 * _role) : uint(_role);
  bool not = _role < 0;
  return(PreconditionRole(role, not));
}

function nowInTimeslot(uint _timeslot) public view timeslotExists(_timeslot) returns(
    bool) {
  return timeslots[_timeslot].active(now);
}

/** Returns TRUE only if the sender is allowed to fire the rule at the timestamp
    provided.
 * Only checks the administrative condition.*/
function canFire(Rule.rule memory _rule, uint _timestamp) internal view returns (bool) {
  if(IGNORE_TIME_DEBUG == true){
    if(_rule.admin_role == 0) {
      return true;
    }

    for(uint i = _rule.admin_start_timeslot; i <= _rule.admin_end_timeslot; i++) {
      if(trbac_state[_rule.admin_role].has(i, msg.sender) == true){
        return true;
      }
    }
  }else{
    for(uint i = _rule.admin_start_timeslot; i <= _rule.admin_end_timeslot; i++) {
      require(i < timeslots.length, "The Rule references a timeslot that does not exists
          .");

      if(timeslots[i].active(_timestamp) || IGNORE_TIME_DEBUG == true) {
        if(_rule.admin_role > 0) {
          return trbac_state[_rule.admin_role].has(i, msg.sender);
        }
        return true;
      }
    }
  }
  return false;
}
```

```
function satifiesPrecondition(Rule.rule memory _rule, address _target_user) internal
    view returns (bool) {
  for(uint r = 0; r < _rule.precondition.length; r++) {
    for(uint ts = 0; ts < _rule.target_timeslot_array.length; ts++) {
      uint role = (_rule.precondition[r] < 0)? uint(-1 * _rule.precondition[r]) : uint(
          _rule.precondition[r]);
      bool not = _rule.precondition[r] < 0;
      if(trbac_state[role].has(_rule.target_timeslot_array[ts], _target_user) == not) {
        return false;
      }
    }
  }
  return true;
}
function satifiesPreconditionRole(Rule.rule memory _rule) internal view returns (bool) {
  for(uint r = 0; r < _rule.precondition.length; r++) {
    for(uint ts = 0; ts < _rule.target_timeslot_array.length; ts++) {
      uint role = (_rule.precondition[r] < 0)? uint(-1 * _rule.precondition[r]) : uint(
          _rule.precondition[r]);
      bool not = _rule.precondition[r] < 0;
      if(role_enablement[role][_rule.target_timeslot_array[ts]] == not) {
        return false;
      }
    }
  }
  return true;
}

// RULES
function fireCanAssignRule(uint _ruleIndex, address _targetUser) public {
  require(_ruleIndex < ca_rules.length, "Unknown Can Assign Rule Index");
  require(canFire(ca_rules[_ruleIndex], now) == true, "Sender is not authorized to fire
      this rule at this time.");
  require(satifiesPrecondition(ca_rules[_ruleIndex], _targetUser) == true,
      "Target User does not satisfy the preconditions for the rule.");

  for(uint i = 0; i < ca_rules[_ruleIndex].target_timeslot_array.length; i++) {
    trbac_state[ca_rules[_ruleIndex].target_role].add(ca_rules[_ruleIndex].
        target_timeslot_array[i],
      _targetUser);
  }
}
function fireCanRevokeRule(uint _ruleIndex, address _targetUser) public {
  require(_ruleIndex < cr_rules.length, "Unknown Can Revoke Rule Index");
  require(canFire(cr_rules[_ruleIndex], now) == true, "Sender is not authorized to fire
      this rule at this time.");
  require(satifiesPrecondition(cr_rules[_ruleIndex], _targetUser) == true,
      "Target User does not satisfy the preconditions for the rule.");

  for(uint i = 0; i < cr_rules[_ruleIndex].target_timeslot_array.length; i++) {
    trbac_state[cr_rules[_ruleIndex].target_role].remove(cr_rules[_ruleIndex].
        target_timeslot_array[i],
      _targetUser);
  }
}

function fireCanEnableRule(uint _ruleIndex) public {
```

```
    require(_ruleIndex < ce_rules.length, "Unknown Can Enable Rule Index");
    require(canFire(ce_rules[_ruleIndex], now) == true, "Sender is not authorized to fire
        this rule at this time.");
    require(satifiesPreconditionRole(ce_rules[_ruleIndex]) == true,
        "Target Role does not satisfy the preconditions for the rule.");

    for(uint i = 0; i < ce_rules[_ruleIndex].target_timeslot_array.length; i++) {
      role_enablement[ce_rules[_ruleIndex].target_role][ce_rules[_ruleIndex].
          target_timeslot_array[i]] = true;
    }
  }
  function fireCanDisableRule(uint _ruleIndex) public {
    require(_ruleIndex < cd_rules.length, "Unknown Can Disable Rule Index");
    require(canFire(cd_rules[_ruleIndex], now) == true, "Sender is not authorized to fire
        this rule at this time.");
    require(satifiesPreconditionRole(cd_rules[_ruleIndex]) == true,
        "Target Role does not satisfy the preconditions for the rule.");

    for(uint i = 0; i < cd_rules[_ruleIndex].target_timeslot_array.length; i++) {
      role_enablement[cd_rules[_ruleIndex].target_role][cd_rules[_ruleIndex].
          target_timeslot_array[i]] = false;
    }
  }
}
```

```
pragma solidity ^0.5.0;

/**
 * @title Roles
 * @dev Library for managing addresses assigned to a Role.
 */
library RoleTimeslots {
  struct Role {
    string name;
    mapping (address => mapping (uint => bool)) bearer;
  }

  /**
   * @dev Give an account access to this role.
   */
  function add(Role storage role, uint timeslot, address account) public {
    require(!has(role, timeslot, account), "Roles: account already has role");
    role.bearer[account][timeslot] = true;
  }

  /**
   * @dev Remove an account's access to this role.
   */
  function remove(Role storage role, uint timeslot, address account) internal {
    require(has(role, timeslot,  account), "Roles: account does not have role");
    role.bearer[account][timeslot] = false;
  }

  /**
   * @dev Check if an account has this role.
   * @return bool
   */
```

```
  function has(Role storage role, uint timeslot, address account) internal view returns (
      bool) {
    require(account != address(0), "Roles: account is the zero address");
    return role.bearer[account][timeslot];
  }
}
```

```
pragma solidity ^0.5.0;

import "./Timeslot.sol";

/**
 * @title Rules
 * @dev Library for managing rules in an ATRBAC policy.
 */
library Rule {
  struct rule {
    uint admin_role;
    uint admin_start_timeslot;
    uint admin_end_timeslot;
    int[] precondition;
    uint[] target_timeslot_array;
    uint target_role;
  }
}
```

```
pragma solidity ^0.5.0;

/**
 * @title Timeslots
 * @author Jonathan Shahen
 * @dev Library for managing timeslots and time intervals in an ATRBAC policy.
 */
library Timeslot {
  /**
   * @title Timeslot
   * @notice Representation of a timeslot, which can be translated to the real world.
        This SPECIFIC implementation of a timeslot if for a single day!
        Any larger, or finer grain than hours, will need a new
        implementation connected to the ATRBAC policy.
   * @param start_hour This holds the inclusive starting hour of the day (hours range
      [0,23])
   * @param end_hour This holds the inclusive ending hour of the day (hours range [0,23])
   */
  struct timeslot {
    uint start_hour;
    uint end_hour;
  }



  uint constant SECONDS_PER_DAY = 24 * 60 * 60;
  uint constant SECONDS_PER_HOUR = 60 * 60;
  uint constant SECONDS_PER_MINUTE = 60;
  function getHour(uint timestamp) internal pure returns (uint hour) {
    uint secs = timestamp % SECONDS_PER_DAY;
```

```
    hour = secs / SECONDS_PER_HOUR;
  }

  /**
   * @dev Check if at timestamp is active for a timeslot.
   * @return bool
   */
  function active(timeslot storage ts, uint timestamp) internal view returns (bool) {
    uint hour = getHour(timestamp);

    if(ts.start_hour <= ts.end_hour) {
      return (ts.start_hour <= hour && hour <= ts.end_hour);
    }else {
      // We allow timeslots to wrap.
      // 11pm - 1am is a valid timeslot which contains the hours: 23(11pm), 0(12am), 1(1am
          )
      return (hour <= ts.start_hour || ts.end_hour <= hour);
    }
  }
}
```

Appendix G

Web3 Commands for Converting ATRBAC Policy to Generic ATRBAC Smart Contract

Web3 commands to create a Generic ATRBAC Smart contract of the ATRBAC policy in Figure 6.3.

```
var atrbac = artifacts.require('ATRBAC');
let accounts = await web3.eth.getAccounts();
var su = accounts[0]; // Super User Account
let instance = await atrbac.deployed(); // Deploy Generic Contract
// Create Roles
await instance.suAddNewRole.sendTransaction('role3', { from: su });
await instance.suAddNewRole.sendTransaction('role1', { from: su });
await instance.suAddNewRole.sendTransaction('goalRole', { from: su });
// Create Timeslots
await instance.suAddTimeSlot.sendTransaction(0, 1, { from: su });
// CanAssign [TA1Tru] <TRUE, [t1−t1], TRUE, t1, role3>
await instance.suAddCARule.sendTransaction(0, 1, 1, [], [1], 1,  { from: su });
// CanAssign [XX3Pos] <role3, [t1−t1], [role3], t1, role2>
await instance.suAddCARule.sendTransaction(1, 1, 1, [1], [1], 4,  { from: su });
// CanAssign [XX4Mix] <role3, [t1−t1], [role2, NOT role3], t1, goalRole>
await instance.suAddCARule.sendTransaction(1, 1, 1, [4, −1 ∗ 1], [1], 3,  { from: su });
// CanRevoke [TA1Tru] <TRUE, [t1−t1], TRUE, t1, role3>
await instance.suAddCRRule.sendTransaction(0, 1, 1, [], [1], 1,  { from: su });
// CanEnable [TA1Tru] <TRUE, [t1−t1], TRUE, t1, role1>
await instance.suAddCERule.sendTransaction(0, 1, 1, [], [1], 2,  { from: su });
// CanEnable [XA3Pos] <TRUE, [t1−t1], [role1], t1, role2>
await instance.suAddCERule.sendTransaction(0, 1, 1, [2], [1], 4,  { from: su });
// CanEnable [XA4Mix] <TRUE, [t1−t1], [role1, NOT role2], t1, role3>
await instance.suAddCERule.sendTransaction(0, 1, 1, [2, −1 ∗ 4], [1], 1,  { from: su });
// CanDisable [TA1Tru] <TRUE, [t1−t1], TRUE, t1, role2>
await instance.suAddCDRule.sendTransaction(0, 1, 1, [], [1], 4,  { from: su });
```

Appendix H

Example Smart Contracts

H.1 New Coin

This is sample code provided by the Ethereum team on how to create a new Coin. This allows for the creation of a new cryptocurrency without the costs associated with hardware and running the miners. This can be used for internal currencies for games or marketplaces. The owner of the contract is called the *minter*, and they are the only ones able to create new coins. This contract does not support the ability to buy Coins with Ether, the minter can setup an outside store to take money in and then mint coin for that address.

```
//Source: https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html
contract Coin {
    address public minter;
    mapping (address => uint) public balances;
    // Constructor code is only run when the contract is created
    constructor() public {
        minter = msg.sender;
    }
    // Sends an amount of newly created coins to an address
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
        balances[receiver] += amount;
    }
    // Sends an amount of existing coins from any caller to an address
    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient balance.");
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
    }
}
```

H.2 Purchase

The seller creates a new instance of this Smart Contract per item, along with the creation of the contract they give 2 times the value of the product being sold. This tells the contract how much the item costs and it provides "good will" to the Buyer that the seller will ship the item. The Seller will get their money back if they abort the contract or if the Buyer confirms they received the item. Once a buyer has confirmed their purchase, by providing twice the cost of the item as "good will" to the Seller. When the Buyer confirm that they have received the item, the Buyer gets refunded the cost of the item (their good will) and the Seller gets refund 3 * the cost of the item (good will + Buyer's money).

```
// Reference: https://solidity.readthedocs.io/en/v0.5.3/solidity-by-example.html
contract Purchase {
    uint256 public val;
    address payable public seller;
    address payable public buyer;
    enum State {Created, Locked, Inactive}
    State public state;

    // Ensure that `msg.value` is an even number.
    constructor() public payable {
        seller = msg.sender;
        val = msg.value / 2;
        require((2 * val) == msg.value, "Value has to be even.");
    }

    modifier condition(bool _condition) { require(_condition); _; }
    modifier onlyBuyer() { require(msg.sender == buyer, "Only buyer."); _; }
    modifier onlySeller() { require(msg.sender == seller, "Only seller."); _;}
    modifier inState(State _state) { require(state == _state, "Invalid state."); _; }

    // Abort the purchase and reclaim the ether.
    function abort() public onlySeller inState(State.Created) {
        state = State.Inactive;
        seller.transfer(address(this).balance);
    }
    // Confirm the purchase as buyer. Transaction has to include `2 * val` ether.
    function confirmPurchase() public payable inState(State.Created)
            condition(msg.value == (2 * val)) {
        buyer = msg.sender;
        state = State.Locked;
    }
    // Confirm that you (the buyer) received the item. This will release the locked ether.
    function confirmReceived() public onlyBuyer inState(State.Locked) {
        state = State.Inactive;
        buyer.transfer(val);
        seller.transfer(address(this).balance);
    }
}
```

H.3 Auction

A smart contract to host an auction. If your bid is outbid by another account, then you may withdraw your previous bids. If you are the current highest bid, you are unable to remove your last bid, but can remove all previous bids. The auction ends based on a time limit, with no option to end the auction early.

```
// Reference: https://solidity.readthedocs.io/en/v0.5.3/solidity-by-example.html
contract SimpleAuction {
    address payable public beneficiary;
    uint public auctionEndTime;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    bool ended;

    constructor(uint _biddingTime,address payable _beneficiary) public {
        beneficiary = _beneficiary;
        auctionEndTime = now + _biddingTime;
    }
    // Bid on the auction with the value sent together with this transaction.
    // The value will only be refunded if the auction is not won.
    function bid() public payable {
        require(now <= auctionEndTime,"Auction already ended.");
        require(msg.value > highestBid,"There already is a higher bid.");
        if (highestBid != 0) {
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
    // Withdraw a bid that was overbid.
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            pendingReturns[msg.sender] = 0;
            if (!msg.sender.send(amount)) {
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }
    // End the auction and send the highest bid to the beneficiary.
    function auctionEnd() public {
        require(now >= auctionEndTime, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");
        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

Appendix I

Vagabond CPLEX OPL Code

```
int numMachines=...; range Machines=1..numMachines;
int numClients=...; range Clients=1..numClients;
int migrationBudget=...;
int vmsPerClient[Clients]=...;
int machineCapacity[Machines]=...;

// Sum of previous Information Leakage ClientToClient
int L[Clients,Clients]=...;
// Sum of Client VMs per Machine
int p0[Machines,Clients]=...;

int r=...;
int t=ftoi(ceil(lg(r+1)));
int dMaxVal=ftoi(pow(2,2*t-1)-1);

range tRange=0..t;
range zRange=0..t-1;

// NEW Placement of the Sum of Client VMs per Machine
dvar int p1[Machines,Clients] in 0..r;
// Variables to fix the quadratic variable from old reduction
dvar int p[Machines,Clients,tRange] in 0..1;
dvar int e[Clients,Clients,Machines,tRange,tRange] in 0..1;
dvar int d[Clients,Clients,Machines,zRange] in 0..dMaxVal;

minimize
  if(max_client == true) {
    max(c0, c1 in Clients: c0 != c1) (
       L[c0,c1] + sum(k in Machines, z in zRange) (d[c0,c1,k,z])
    );
  }else { //sum total information leakage
    sum(c0, c1 in Clients: c0 != c1)
      (L[c0,c1] + sum(k in Machines, z in zRange) (d[c0,c1,k,z]) );
  }
```

```
subject to {
  // p1 represented as a value from p's bit form
  forall(i in Clients, k in Machines)
    p1[k,i] == sum(l in tRange) (pow(2,l) * p[k,i,l]);
  // Bitwise AND for all pairs in p1
  forall(k in Machines, c0 in Clients, c1 in Clients, u in tRange, v in tRange)
    0 <= p[k,c0,u] + p[k,c1,v] - (2 * e[c0,c1,k,u,v]) <= 1;
  // Setting up values for d
  forall(k in Machines, c0 in Clients, c1 in Clients, z in zRange)
    d[c0,c1,k,z] == sum(l in zRange) (pow(2,z+l) *e[c0,c1,k,l,z]);
  // Capacity of servers respected
  forall(k in Machines)
    machineCapacity[k] >= sum(i in Clients) (p1[k,i]);
  // Sum of clients in p1 should equal the number of vms of that client
  forall(i in Clients)
    vmsPerClient[i] == sum(k in Machines) (p1[k,i]);
  // Migration budget
  2 * migrationBudget >= sum(i in Clients, k in Machines) ( abs(p1[k,i] - p0[k,i]) );
};
```

Appendix J

Boolean Circuit Reductions to ILP

A summary of all ILP reductions, using the intermediate CNF-SAT reductions, are listed in Table J.1. Detailed steps for each reduction can be found in the following sections.

Table J.1: Summary of the "From CNF SAT" reductions to ILP. All ILP variables are integers in the range $[0, 1]$.

GATE	Boolean Expression	From CNF SAT v1 (C)	From CNF SAT v2 (D)
NOT	$y = \neg x_1$	$y + x \geq 1$ $y + x \leq 1$	Same
BUF	$y = x_1$	$y == x$	Same
AND	$y = x_0 \wedge \ldots \wedge x_{n-1}$	$x_0 - y \geq 0$ $x_1 - y \geq 0$ $\ldots$ $x_{n-1} - y \geq 0$ $y - (x_0 + x_1 \ldots + x_{n-1}) \geq 1 - n$	Section J.2.1
NAND	$y = \neg(x_0 \wedge \ldots \wedge x_{n-1})$	$y + x_0 \geq 1$ $y + x_1 \geq 1$ $\ldots$ $y + x_{n-1} \geq 1$ $x_0 + x_1 \ldots + x_{n-1} + y \leq n$	Section J.2.2
OR	$y = x_0 \vee \ldots \vee x_{n-1}$	$x_0 + x_1 + \ldots + x_{n-1} - y \geq 0$ $y - x_0 \geq 0$ $y - x_1 \geq 0$ $\ldots$ $y - x_{n-1} \geq 0$	Section J.2.3
NOR	$y = \neg(x_0 \vee \ldots \vee x_{n-1})$	$y + x_0 + x_1 + \ldots + x_{n-1} \geq 1$ $y + x_0 \leq 1$ $y + x_1 \leq 1$ $\ldots$ $y + x_{n-1} \leq 1$	Section J.2.4
XOR	$y = x_0 \oplus x_1$	$x_0 + x_1 - y \geq 0$ $y + x_0 + x_1 \leq 2$ $y + x_1 - x_0 \geq 0$ $y + x_0 - x_1 \geq 0$	Same
XNOR	$y = \neg(x_0 \oplus x_1)$	$y + x_0 + x_1 \geq 1$ $y - x_0 - x_1 \geq -1$ $x_1 - y - x_0 \geq -1$ $x_0 - y - x_1 \geq -1$	Same

J.1 From CNF-SAT v1

An alternative to the approach of Section 8.6.1. This one adopts a "via CNF SAT" mindset. In each of the following, all the variables are binary, i.e., 0 or 1.

J.1.1 Big AND v1

$$y \iff x_0 \wedge x_1 \wedge \ldots \wedge x_{n-1}$$

CNF:

$$\begin{aligned} & y \implies x_0 \wedge x_1 \wedge \ldots \wedge x_{n-1} \wedge y \impliedby x_0 \wedge x_1 \wedge \ldots \wedge x_{n-1} \\ & \equiv \neg y \vee (x_0 \wedge \ldots \wedge x_{n-1}) \wedge \neg (x_0 \wedge \ldots \wedge x_{n-1}) \vee y \\ & \equiv \neg y \vee x_0 \wedge \neg y \vee x_1 \ldots \wedge \neg y \vee x_{n-1} \wedge \neg x_0 \vee \neg x_1 \vee \ldots \vee \neg x_{n-1} \vee y \end{aligned}$$

ILP:

$$\begin{aligned} x_0 - y &\geq 0 \\ x_1 - y &\geq 0 \\ &\ldots \\ x_{n-1} - y &\geq 0 \\ y - (x_0 + x_1 \ldots + x_{n-1}) &\geq 1 - n \end{aligned}$$

J.1.2 Big NAND v1

$$\neg y \iff x_0 \wedge x_1 \wedge \ldots \wedge x_{n-1}$$

CNF:

$$\begin{aligned} & \neg y \implies x_0 \wedge x_1 \wedge \ldots \wedge x_{n-1} \wedge \neg y \impliedby x_0 \wedge x_1 \wedge \ldots \wedge x_{n-1} \\ & \equiv y \vee (x_0 \wedge \ldots \wedge x_{n-1}) \wedge \neg (x_0 \wedge \ldots \wedge x_{n-1}) \vee \neg y \\ & \equiv y \vee x_0 \wedge y \vee x_1 \ldots \wedge y \vee x_{n-1} \wedge \neg x_0 \vee \neg x_1 \vee \ldots \vee \neg x_{n-1} \vee \neg y \end{aligned}$$

ILP:

$$\begin{aligned} y + x_0 &\geq 1 \\ y + x_1 &\geq 1 \\ &\dots \\ y + x_{n-1} &\geq 1 \\ x_0 + x_1 \dots + x_{n-1} + y &\leq n \end{aligned}$$

J.1.3 Big OR v1

$$x_0 \vee x_1 \vee \dots \vee x_{n-1} \iff y$$

CNF:

$$\begin{aligned} & y \implies x_0 \vee x_1 \vee \dots \vee x_{n-1} \wedge y \impliedby x_0 \vee x_1 \vee \dots \vee x_{n-1} \\ & \equiv \neg y \vee x_0 \vee x_1 \vee \dots \vee x_{n-1} \wedge y \vee (\neg x_0 \wedge \neg x_1 \wedge \dots \wedge \neg x_{n-1}) \\ & \equiv \neg y \vee x_0 \vee x_1 \vee \dots \vee x_{n-1} \wedge y \vee \neg x_0 \wedge y \vee \neg x_1 \dots \wedge y \vee \neg x_{n-1} \end{aligned}$$

ILP:

$$\begin{aligned} x_0 + x_1 + \dots + x_{n-1} - y &\geq 0 \\ y - x_0 &\geq 0 \\ y - x_1 &\geq 0 \\ &\dots \\ y - x_{n-1} &\geq 0 \end{aligned}$$

J.1.4 Big NOR v1

$$x_0 \vee x_1 \vee \dots \vee x_{n-1} \iff \neg y$$

CNF:

$$\begin{aligned}
&\neg y \implies x_0 \vee x_1 \vee \ldots \vee x_{n-1} \wedge \neg y \impliedby x_0 \vee x_1 \vee \ldots \vee x_{n-1} \\
&\equiv y \vee x_0 \vee x_1 \vee \ldots \vee x_{n-1} \wedge \neg y \vee (\neg x_0 \wedge \neg x_1 \wedge \ldots \wedge \neg x_{n-1}) \\
&\equiv y \vee x_0 \vee x_1 \vee \ldots \vee x_{n-1} \wedge \neg y \vee \neg x_0 \wedge \neg y \vee \neg x_1 \ldots \wedge \neg y \vee \neg x_{n-1}
\end{aligned}$$

ILP:

$$\begin{aligned}
y + x_0 + x_1 + \ldots + x_{n-1} &\geq 1 \\
y + x_0 &\leq 1 \\
y + x_1 &\leq 1 \\
&\ldots \\
y + x_{n-1} &\leq 1
\end{aligned}$$

J.1.5 NOT v1

$$x \iff \neg y$$

CNF	**ILP**
$\neg y \implies x \wedge \neg y \impliedby x$	$y + x \geq 1$
$\equiv y \vee x \wedge \neg y \vee \neg x$	$-y - x \geq -1$

J.1.6 XOR v1

$$y \iff (x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1) \equiv y \iff (x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1)$$

CNF:

$$\begin{aligned}
&y \implies (x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1) \wedge y \impliedby (x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1) \\
&\equiv \neg y \vee x_0 \vee x_1 \wedge \neg y \vee \neg x_0 \vee \neg x_1 \wedge y \vee \neg x_0 \vee x_1 \wedge y \vee x_0 \vee \neg x_1
\end{aligned}$$

ILP:

$$\begin{aligned} x_0 + x_1 - y &\geq 0 \\ y + x_0 + x_1 &\leq 2 \\ y + x_1 - x_0 &\geq 0 \\ y + x_0 - x_1 &\geq 0 \end{aligned}$$

J.1.7 XNOR v1

$$\neg y \iff (x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1) \equiv \neg y \iff (x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1)$$

CNF:

$$\begin{aligned} &\neg y \implies (x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1) \wedge \neg y \impliedby (x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1) \\ &\equiv y \vee x_0 \vee x_1 \wedge y \vee \neg x_0 \vee \neg x_1 \wedge \neg y \vee \neg x_0 \vee x_1 \wedge \neg y \vee x_0 \vee \neg x_1 \end{aligned}$$

ILP:

$$\begin{aligned} y + x_0 + x_1 &\geq 1 \\ y - x_0 - x_1 &\geq -1 \\ x_1 - y - x_0 &\geq -1 \\ x_0 - y - x_1 &\geq -1 \end{aligned}$$

J.2 From CNF-SAT v2

Yet another mindset is to reduce from CIRCUIT-SAT to SAT to 3-CNF-SAT, from CLRS.

J.2.1 Big AND v2

$$y \iff x_0 \wedge x_1 \wedge \ldots \wedge x_{n-1}$$

3-CNF:

$$
\begin{aligned}
x_0 \wedge x_1 &\iff y_1 \equiv x_0 \vee x_1 \vee \neg y_1 \wedge x_0 \vee \neg x_1 \vee \neg y_1 \wedge \neg x_0 \vee x_1 \vee \neg y_1 \wedge \neg x_0 \vee \neg x_1 \vee y_1 \\
\wedge y_1 \wedge x_2 &\iff y_2 \equiv y_1 \vee x_2 \vee \neg y_2 \wedge y_1 \vee \neg x_2 \vee \neg y_2 \wedge \neg y_1 \vee x_2 \vee \neg y_2 \wedge \neg y_1 \vee \neg x_2 \vee y_2 \\
&\ldots \equiv \ldots \\
\wedge y_{n-2} \wedge x_{n-1} &\iff y \equiv y_{n-2} \vee x_{n-1} \vee \neg y \wedge y_{n-2} \vee \neg x_{n-1} \vee \neg y \\
&\qquad \wedge \neg y_{n-2} \vee x_{n-1} \vee \neg y \wedge \neg y_{n-2} \vee \neg x_{n-1} \vee y
\end{aligned}
$$

ILP:

$$
\left.
\begin{array}{ll}
x_0 + x_1 - y_1 & \geq 0 \\
x_0 - x_1 - y_1 & \geq -1 \\
-x_0 + x_1 - y_1 & \geq -1 \\
-x_0 - x_1 + y_1 & \geq -1
\end{array}
\right\} x_0 \wedge x_1 \iff y_1
$$

$$
\left.
\begin{array}{ll}
y_1 + x_2 - y_2 & \geq 0 \\
y_1 - x_2 - y_2 & \geq -1 \\
-y_1 + x_2 - y_2 & \geq -1 \\
-y_1 - x_2 + y_2 & \geq -1
\end{array}
\right\} y_1 \wedge x_2 \iff y_2
$$

$$\ldots$$

$$
\left.
\begin{array}{ll}
y_{n-2} + x_{n-1} - y & \geq 0 \\
y_{n-2} - x_{n-1} - y & \geq -1 \\
-y_{n-2} + x_{n-1} - y & \geq -1 \\
-y_{n-2} - x_{n-1} + y & \geq -1
\end{array}
\right\} y_{n-2} \wedge x_{n-1} \iff y
$$

J.2.2 Big NAND v2

$$\neg y \iff x_0 \wedge x_1 \wedge \ldots \wedge x_{n-1}$$

3-CNF	ILP
$x_0 \wedge x_1 \iff y_1$	
$\wedge \quad y_1 \wedge x_2 \iff y_2$	$x_0 + x_1 - y_1 \geq 0$
$\ldots$	$x_0 - x_1 - y_1 \geq -1$
$\wedge \quad y_{n-2} \wedge x_{n-1} \iff \neg y$	$-x_0 + x_1 - y_1 \geq -1$
$\equiv$	$-x_0 - x_1 + y_1 \geq -1$
$x_0 \vee x_1 \vee \neg y_1$	$y_1 + x_2 - y_2 \geq 0$
$\wedge \quad x_0 \vee \neg x_1 \vee \neg y_1$	$y_1 - x_2 - y_2 \geq -1$
$\wedge \quad \neg x_0 \vee x_1 \vee \neg y_1$	$-y_1 + x_2 - y_2 \geq -1$
$\wedge \quad \neg x_0 \vee \neg x_1 \vee y_1$	$-y_1 - x_2 + y_2 \geq -1$
$\wedge \quad y_1 \vee x_2 \vee \neg y_2$	$\ldots$
$\wedge \quad y_1 \vee \neg x_2 \vee \neg y_2$	$y_{n-3} + x_{n-2} - y_{n-2} \geq 0$
$\wedge \quad \neg y_1 \vee x_2 \vee \neg y_2$	$y_{n-3} - x_{n-2} - y_{n-2} \geq -1$
$\wedge \quad \neg y_1 \vee \neg x_2 \vee y_2$	$-y_{n-3} + x_{n-2} - y_{n-2} \geq -1$
$\ldots$	$-y_{n-3} - x_{n-2} + y_{n-2} \geq -1$
$\wedge \quad \neg y_{n-2} \vee \neg x_{n-1} \vee \neg y$	$-y_{n-2} - x_{n-1} - y \geq -2$
$\wedge \quad y_{n-2} \vee \neg x_{n-1} \vee y$	$y_{n-2} - x_{n-1} + y \geq 0$
$\wedge \quad \neg y_{n-2} \vee x_{n-1} \vee y$	$-y_{n-2} + x_{n-1} + y \geq 0$
$\wedge \quad y_{n-2} \vee x_{n-1} \vee y$	$y_{n-2} + x_{n-1} + y \geq 1$

J.2.3 Big OR v2

$$x_0 \vee x_1 \vee \ldots \vee x_{n-1} \iff y$$

3-CNF	ILP
$x_0 \vee x_1 \iff y_1$	$x_0 + x_1 - y_1 \geq 0$
$\wedge \;\; y_1 \vee x_2 \iff y_2$	$x_0 - x_1 + y_1 \geq 0$
$\ldots$	$-x_0 + x_1 + y_1 \geq 0$
$\wedge \;\; y_{n-3} \vee x_{n-2} \iff y_{n-2}$	$-x_0 - x_1 + y_1 \geq -1$
$\wedge \;\; y_{n-2} \vee x_{n-1} \iff y$	$y_1 + x_2 - y_2 \geq 0$
$\equiv$	$y_1 - x_2 + y_2 \geq 0$
$x_0 \vee x_1 \vee \neg y_1$	$-y_1 + x_2 + y_2 \geq 0$
$\wedge \;\; x_0 \vee \neg x_1 \vee y_1$	$-y_1 - x_2 + y_2 \geq -1$
$\wedge \;\; \neg x_0 \vee x_1 \vee y_1$	$\ldots$
$\wedge \;\; \neg x_0 \vee \neg x_1 \vee y_1$	$y_{n-3} + x_{n-2} - y_{n-2} \geq 0$
$\wedge \;\; y_1 \vee x_2 \vee \neg y_2$	$y_{n-3} - x_{n-2} + y_{n-2} \geq 0$
$\wedge \;\; y_1 \vee \neg x_2 \vee y_2$	$-y_{n-3} + x_{n-2} + y_{n-2} \geq 0$
$\wedge \;\; \neg y_1 \vee x_2 \vee y_2$	$-y_{n-3} - x_{n-2} + y_{n-2} \geq -1$
$\wedge \;\; \neg y_1 \vee \neg x_2 \vee y_2$	$y_{n-2} + x_{n-1} - y \geq 0$
$\ldots$	$y_{n-2} - x_{n-1} + y \geq 0$
$\wedge \;\; y_{n-3} \vee x_{n-2} \vee \neg y_{n-2}$	$-y_{n-2} + x_{n-1} + y \geq 0$
$\wedge \;\; y_{n-3} \vee \neg x_{n-2} \vee y_{n-2}$	$-y_{n-2} - x_{n-1} + y \geq -1$
$\wedge \;\; \neg y_{n-3} \vee x_{n-2} \vee y_{n-2}$	
$\wedge \;\; \neg y_{n-3} \vee \neg x_{n-2} \vee y_{n-2}$	
$\wedge \;\; y_{n-2} \vee x_{n-1} \vee \neg y$	
$\wedge \;\; y_{n-2} \vee \neg x_{n-1} \vee y$	
$\wedge \;\; \neg y_{n-2} \vee x_{n-1} \vee y$	
$\wedge \;\; \neg y_{n-2} \vee \neg x_{n-1} \vee y$	

J.2.4 Big NOR v2

$$x_0 \vee x_1 \vee \ldots \vee x_{n-1} \iff \neg y$$

3-CNF	**ILP**
$x_0 \vee x_1 \iff y_1$	$x_0 + x_1 - y_1 \geq 0$
$\wedge \;\; y_1 \vee x_2 \iff y_2$	$x_0 - x_1 + y_1 \geq 0$
$\ldots$	$-x_0 + x_1 + y_1 \geq 0$
$\wedge \;\; y_{n-2} \vee x_{n-1} \iff \neg y$	$-x_0 - x_1 + y_1 \geq -1$
$\equiv$	$y_1 + x_2 - y_2 \geq 0$
$x_0 \vee x_1 \vee \neg y_1$	$y_1 - x_2 + y_2 \geq 0$
$\wedge \;\; x_0 \vee \neg x_1 \vee y_1$	$-y_1 + x_2 + y_2 \geq 0$
$\wedge \;\; \neg x_0 \vee x_1 \vee y_1$	$-y_1 - x_2 + y_2 \geq -1$
$\wedge \;\; \neg x_0 \vee \neg x_1 \vee y_1$	$\ldots$
$\wedge \;\; y_1 \vee x_2 \vee \neg y_2$	$y_{n-2} + x_{n-1} + y \geq 1$
$\wedge \;\; y_1 \vee \neg x_2 \vee y_2$	$y_{n-2} - x_{n-1} - y \geq -1$
$\wedge \;\; \neg y_1 \vee x_2 \vee y_2$	$-y_{n-2} + x_{n-1} - y \geq -1$
$\wedge \;\; \neg y_1 \vee \neg x_2 \vee y_2$	$-y_{n-2} - x_{n-1} - y \geq -2$
$\ldots$	
$\wedge \;\; y_{n-2} \vee x_{n-1} \vee y$	
$\wedge \;\; y_{n-2} \vee \neg x_{n-1} \vee \neg y$	
$\wedge \;\; \neg y_{n-2} \vee x_{n-1} \vee \neg y$	
$\wedge \;\; \neg y_{n-2} \vee \neg x_{n-1} \vee \neg y$	