SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

$$\begin{tikzcd}
 & \sigma F \arrow[rd, "\omega"] & & \\
X \arrow[dd, "c"] & X \arrow[l, "m"'] \arrow[u, "!_c"] \arrow[r, dotted, "!_{bocom}"'] & \sigma F \arrow[rd, "\mathbf{id}"] \arrow[dd, "1_F"] & \\
 & & & \sigma(F) \\
F(X) \arrow[r, "b"'] & F(X) \arrow[r, dotted, "F(!_{bocom})"] \arrow[d, "F(!_c)"'] & F(\sigma F) \arrow[ru, "1_F^{-1}"'] & \\
 & F(\sigma F) \arrow[ru, "F(\omega)"'] & &
\end{tikzcd}$$

Latent Behaviour Analysis

Submitted by

Eric G. ROTHSTEIN MORRIS

Thesis Advisor

Dr. Sudipta CHATTOPADHYAY

Information Systems Technology and Design (ISTD)

A thesis submitted to the Singapore University of Technology and Design in fulfillment of the requirement for the degree of Doctor of Philosophy

2021

PhD Thesis Examination Committee

TEC Chair:	Prof. Jianying Zhou
Main Advisor:	Prof. Sudipta Chattopadhyay
Internal TEC member 1:	Prof. CHEN Binbin
Internal TEC member 2:	Prof. Cyrille Jegourel

Declaration

I, Eric G. ROTHSTEIN MORRIS, declare that this thesis titled, "Latent Behaviour Analysis" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

I hereby confirm the following:

- I hereby confirm that the thesis work is original and has not been submitted to any other University or Institution for higher degree purposes.
- I hereby grant SUTD the permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created in accordance with Policy on Intellectual Property, clause 4.2.2.
- I have fulfilled all requirements as prescribed by the University and provided 1 copy of my thesis in PDF.
- I have attached all publications and award list related to the thesis (e.g. journal, conference report and patent).
- The thesis does / does not (delete accordingly) contain patentable or confidential information.
- I certify that the thesis has been checked for plagiarism via turnitin/ithenticate. The score is 100%.

Name and signature:

Date:

Abstract

Information Systems Technology and Design (ISTD)

Doctor of Philosophy

Latent Behaviour Analysis

by Eric G. ROTHSTEIN MORRIS

Latent behaviour analysis is the study of how the behaviour of a dynamical system changes when its state space is transformed. A transformation of the state space can model the effects of an attack, the effects of a fault, or arbitrary interactions of the system with an external force.

Complete

Publications

Journal Papers, Conference Presentations, etc...

Acknowledgements

I have so many people to thank...I should start writing this.

Contents

List of Figures

List of Tables

List of Abbreviations

AIG	**A**nd-**I**nverter **G**raphs
COI	**C**one **O**f **I**nfluence
OIC	Dual **C**one **O**f **I**nfluence
LBA	**L**atent **B**ehaviour **A**nalysis

To my family and friends. Your constant love and support made this dream possible.

Chapter 1

Introduction

The first thing we need to do is describe why latent behaviours are interesting. This requires a historical context.

1.1 A Brief History of Coalgebras

This section explains the importance of coalgebras as a modelling system, why they are so compatible with computability (monads), and in general why they are useful.

I could probably be inspired on how to structure this section by publications on coalgebras. There is a large body of literature I can use.

At least cover that systems are of a form $\delta \colon X \to D$ where D is some domain, and that there is a way to go from D back to X to continue computation.

Final coalgebras are important because $D = X$, so it is obvious how to continue.

1.2 Latent Behaviours

Explain that in this document we explore the notion of creating new behaviours through transformations of the state space of systems.

1.2.1 Intuition

We present a light version of latent behaviours: you normally start at X, then you go to D and then back to X, but what if we transformed X?

In latent behaviours we use a transformation $X \to X$ to affect the behaviour of the system. This transformation is an abstraction; the original system naturally transforms to a system $X \to D \to X$ using composition, also of type $X \to X$. Each function $X \to X$ can be seen as a black-box of some behaviour. Their interweaving means that they are acting simultaneously.

1.3 Research Questions

We consider the following broad yet interesting research questions that are pervasive in systems security to motivate our work.

Research Question 1.1. How do we precisely describe and generate attacker models, attacks and attackers?

Research Question 1.2. How do we efficiently quantify the effect that a given attacker has on a given system?

Research Question 1.3. How do we compare attacker models and attackers with respect to the effect they potentially have on a given system?

Research Question 1.4. How do we repair a system that is vulnerable with respect to a given attacker model?

We do not pretend to solve these questions at this broad level, but we address concrete instances of these questions, which appear as we instantiate systems, attacker models, security requirements, etc. in the following sections.

1.4 Applications

1.4.1 Attacker Classification

How can we model attackers? How can we model their actions? How can we prepare against them?

1.4.2 Cyber-Physical System Redesign

Many attacks are discovered through testing: an experienced hacker creates a proof of concept to illustrate how some system may be attacked. Then, an abstraction is created to explain why the attack works, and to suggest countermeasures against it.

1.4.3 Program Repair

Chapter 2

Preliminaries and Notation

This section has content which I did not develop, but that I use. The content I developed follows in the next chapters.

2.1 Notation and Constructions on Sets and Functions

2.1.1 Sets and elements

We denote *sets* by upper-case letters X, Y, Z, etc, and we denote their *elements* by lower-case letters x, y, z, etc.

We denote the set of natural numbers by $\mathbb{N}$ and the set of natural numbers without zero by $\mathbb{N}^+$. Similarly, we denote the set of real numbers by $\mathbb{R}$ and the set of positive real numbers (excluding zero) by $\mathbb{R}^+$.

We denote the size of a finite set X by $|X|$, and the empty set by $\emptyset$.

2.1.2 Functions

We denote *functions* by lower-case letters f, g, h, etc. A function $f\colon X \to Y$ maps the elements from the set X to elements of the set Y. An *endofunction* is a function $g\colon X \to X$ which maps a set X to itself. We denote function composition by $\circ$. We say that an endofunction $h\colon X \to X$ has finite support iff $h(x) \neq x$ for a finite number of x in X, even if X is infinite.

2.1.3 Range set

A *range set* n where $n \in \mathbb{N}^+$ is a set of n elements, i.e., $n = \{0, 1, \ldots, n-1\}$. It is usually clear from the context when n represents a number or a range set. We highlight three important range sets: 0, 1 and 2.

The range set 0 is equal to the empty set $\emptyset$. The range set 1, which is equal to the set $\{0\}$, serves as a constant for several set operations (modulo isomorphism). The range set 2, which is equal to $\{0, 1\}$, to model booleans: 0 models `false` and 1 models `true`.

2.1.4 Subsets $X \to 2$

Given a set X, we characterise the *subsets of* X using functions of type $X \to 2$; we say that $x \in X$ is an element of the subset characterised by $f \colon X \to 2$ iff $f(x) = 1$.

2.1.5 Exponential Set Y^X

The *exponential set* Y^X is the set of all functions of type $X \to Y$. The exponential set $1 \to X$ is isomorphic to X, and we use it to select objects from X; more precisely, we choose an element $x \in X$ via a function $f \colon 1 \to X$ such that $f(0) = x$. (This seems overcomplicated at first, but we use this equivalence when discussing the following concepts of F-algebra and F-coalgebra.)

2.1.6 Sum Set $X + Y$

The *sum set* $X + Y$ is the disjoint union of X and Y, i.e.

$$X + Y \triangleq \{\iota_1(x) | x \in X\} \cup \{\iota_2(y) | y \in Y\},$$

where ι_1 and ι_2 act as different tags.

2.1.7 Product Set $X \times Y$

The *product set* $X \times Y$ is the cartesian product of X and Y, i.e.

$$X \times Y \triangleq \{(x, y) | x \in X, y \in Y\}.$$

The *finite product set* of sets $X_1, \ldots, X_n$ is their cartesian product; i.e., $X_1 \times \ldots \times X_n$. We denote product sets by upper-case letters with an arrow above $\vec{X}, \vec{Y}, \vec{Z}$, etc.

2.1.8 Vector/Tuple $\vec{x}$

Given a finite product set $\vec{X} = X_1 \times \ldots \times X_n$, we denote the elements of $\vec{X}$ by $\vec{x}$, $\vec{y}$, $\vec{z}$, etc. We call these elements *vectors* or, alternatively, *tuples*.

2.1.9 Coordinates $\vec{x}[\pi]$

Given a finite product set $\vec{X} = X_1 \times \ldots \times X_n$, its *coordinates* are $\pi_1, \ldots, \pi_n$, where $\pi_i \colon \vec{X} \to X_i$ extract the i-th value of a vector; i.e., $\pi_i(x_1, \ldots, x_n) \triangleq x_i$, for $1 \leq i \leq n$. We often write the expression $\vec{x}[\pi]$ instead of $\pi(\vec{x})$ when π is a coordinate.

We use natural numbers for coordinates when no explicit set of coordinates is provided. More precisely, given $\vec{x} \in X_1 \times \ldots \times X_n$ where $\vec{x} = (v_1, \ldots, v_n)$, if no set of coordinates is provided, then we use the coordinates $1, 2, \ldots, n$, so $\vec{x}[i] = v_i$ for $i \in \{1, \ldots, n\}$.

2.1.10 Constant Function Δ_x

Given sets X and Y, each element $x \in X$ lifts to a *constant function* $\Delta_x \colon Y \to X$, defined by $\Delta_x(y) = x$, for $y \in Y$.

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[rr, bend left, "g \circ f"] & Y \arrow[r, "g"] \arrow[rr, bend right, "h \circ g"'] & Z \arrow[r, "h"] & A
\end{tikzcd}$$

FIGURE 2.1: Commutative diagram describing the equality $(h \circ g) \circ f = h \circ g \circ f = h \circ (g \circ f)$

2.1.11 Pair Function (f, g)

Given two functions $f \colon X \to Y$ and $g \colon X \to Z$, the *pair function* $(f, g) \colon X \to Y \times Z$ is defined by $(f, g)(x) = (f(x), g(x))$.

2.1.12 Product Function $f \times g$

Given two functions $f \colon X \to Y$ and $g \colon A \to B$, the *product function* $f \times g \colon X \times A \to Y \times B$ is defined by $f \times g(x, a) = (f(x), g(a))$.

2.1.13 Function Mapping $f(X)$

Given a set X and a function $f \colon X \to Y$, the *mapping of f over X*, denoted $f(X)$, is the set defined by $f(X) \triangleq \{f(x) | x \in X\}$.

2.2 Category Theory Preliminaries and Notation

2.2.1 Categories, Objects and Arrows

A *category* $\mathbf{C}$ is a mathematical concept similar to a graph, comprised of two aspects: a collection of *objects*, denoted $Obj(\mathbf{C})$, and a collection of *arrows* among those objects, denoted $Arr(\mathbf{C})$. Each category satisfies the following rules:

- every object X has an identity arrow that maps X to itself, denoted $\mathtt{id}_X$,
- given the objects X, Y and Z, and the arrows $X \xrightarrow{f} Y$ and $Y \xrightarrow{g} Z$, the arrow $X \xrightarrow{g \circ f} Z$ exists; i.e. the composition of arrows is well defined,
- the identity arrows act as units for arrow composition; i.e., for $X \xrightarrow{f} Y$, the equality $\mathtt{id}_Y \circ f = f = f \circ \mathtt{id}_X$ holds.
- the composition of arrows is associative; i.e., $(h \circ g) \circ f = h \circ (g \circ f)$.

2.2.2 Commutative Diagram

A *commutative diagram* is a directed graph which visually represents equations. Like any directed graph, source nodes do not have incoming arrows, and sink nodes do not have outgoing arrows, and they represent the beginning and end of equations. For example, given the functions $f \colon X \to Y$, $g \colon Y \to Z$ and $h \colon Z \to A$, the equalities $(h \circ g) \circ f = h \circ g \circ f = h \circ (g \circ f)$ is represented by the diagram shown in Figure 2.1.

2.2.3 Initial and Terminal Objects

A category $\mathbf{C}$ has an *initial object*, often denoted by 0, if there exists a unique arrow from 0 to every object o in the category. Dually, a category $\mathbf{C}$ has a *final object*, often denoted by 1, if there exists a unique arrow from every object o in the category to 1.

A category may have several different initial and terminal objects, but they are always unique modulo isomorphism.

2.2.4 Functor

Functors are to categories what arrows are to objects. A *functor* F maps a category $\mathbf{C}$ to a category $\mathbf{D}$, mapping $Obj(\mathbf{C})$ to $Obj(\mathbf{D})$, and $Arr(\mathbf{C})$ to $Arr(\mathbf{D})$ such that the following conditions are satisfied for all objects X, Y, Z and arrows f, g.

- for $f\colon X \to Y$, f is mapped to a function of type $g\colon F(X) \to F(Y)$,
- $\mathtt{id}_X$ is mapped to $\mathtt{id}_{F(X)}$; i.e. $F(\mathtt{id}_X) = \mathtt{id}_{F(X)}$.
- if $f\colon X \to Y$ and $g\colon Y \to Z$, then $F(g \circ f) = F(g) \circ F(f)$.

These conditions ensure functors preserve the structure of the category they are mapping.

2.3 The Category of Sets and Functions

The *category of sets and functions* $\mathbf{Set}$ is the category whose objects are sets and whose arrows are functions. The empty set 0 is the initial object, and any set isomorphic to the range set 1 is a terminal object.

2.3.1 Currying

Currying is the use of the isomorphism between the sets $Z^{X\times Y}$ and Z^{Y^X} which maps a function $f\colon (X \times Y) \to Z$ to a function $g\colon X \to Z^Y$ such that $g(x)(y) = f(x, y)$ for $x \in X$ and $y \in Y$. We use currying to adjust the types between equivalent constructions (e.g. between Automata and F-coalgebras).

2.3.2 Partial Application

Given a function $f\colon X \to Y \to Z$ and $x \in X$, the *partial application* of f given x is the function $f_x\colon Y \to Z$, defined for $y \in Y$ by $f_x(y) = f(x)(y)$.

2.3.3 Monoids and Sequences

A *monoid* in the category $\mathbf{Set}$ is a set X paired with a sum function $+\colon X \to X \to X$ and a unit element $0 \in X$ such that 0 is neutral for $+$ and $+$ is associative.

We highlight two monoids for a given set X: the *free monoid of* X and the *endofunctions monoid of* X.

The *free monoid of* X is the set X^* of finite sequences of elements of X, where the sum function is sequence concatenation $\cdot$ and the unit is the empty sequence ε.

The *endofunctions monoid of* X is the set X^X of endofunctions in X, where the sum is function composition $\circ$ and the unit is the identity function $\mathtt{id}_X$.

2.3.4 Languages

Given a set X, a *language with alphabet* X is a subset of X^*.

2.3.5 Deterministic Finite Automata (DFA)

Given a finite set A, a *deterministic finite automaton* (DFA) is a tuple $\mathbb{X} = (X, x_0, \delta, F)$, where X is a finite set, x_0 is an element of X, $\delta\colon X \times A \to X$ is a transition function, and $F\colon X \to 2$ is a subset of X which marks accepting states. Given a sequence $w \in A^*$, we say that the automaton $\mathbb{X}$ *accepts* w if and only if the following conditions are satisfied,

- if $w = \varepsilon$, then $\mathbb{X}$ accepts w iff $F(x_0) = 1$,
- if $w = a \cdot w'$, then $\mathbb{X}$ accepts w iff the automaton $(X, \delta(x_0, a), \delta, F)$ accepts w', with $a \in A$ and $w' \in A^*$.

The denotational semantics of a DFA is the language with alphabet A whose sequences it accepts, rejecting the rest.

2.3.6 Set Isomorphisms

Homomorphisms and isomorphisms

List of Relevant Isomorphisms

- $1 \times X \simeq X$,
- $2 \times X \simeq X + X$,
- $X^1 \simeq X$,
- $X^2 \simeq X \times X$,
- $Z^{X \times Y} \simeq {Z^Y}^X$,
- $X \times Y \simeq Y \times X$
- $X + Y \simeq Y + X$
- $|X| = n$ iff $X \simeq n$.
- $\underbrace{(X \times \ldots \times X)}_{n \text{ times}} \simeq X^n$ (for this case, we say that n is the set of coordinates of X^n)

2.4 Universal (Co)Algebra Preliminaries and Notation

2.4.1 F-(co)algebras

Given an endofunctor F in a category $\mathbf{C}$, an F*-coalgebra* is a pair $(X, X \xrightarrow{c} F(X))$ of an object X and a morphism $c\colon X \to F(X)$. We use F-coalgebras to model dynamic systems with state. We call X the *carrier* and c the *dynamics*.

Dually, an F*-algebra* is a pair $(X, F(X) \xrightarrow{a} X)$.

2.4.2 Pointed coalgebras

Pointed coalgebras are coalgebras with a distinguished state, usually referred to as the *initial state*. Formally, for a functor F, a pointed F-coalgebra is a triple (X, c, x_0) where (X, c) is an F-coalgebra and (X, x_0) is a Δ_X-algebra, i.e., $x_0\colon 1 \to X$, and $x_0(\star)$ characterises the initial state.

2.4.3 Automata as Pointed Coalgebras

DFA which characterise languages with an alphabet A can be modelled by coalgebras of the functor $F(X) = 2 \times X^A$. Given a DFA (X, x_0, δ, F), the corresponding F-coalgebra is $(X, (F, \delta))$, where $(F, \delta)\colon X \to 2 \times X^2$ is a function pair. We can make this coalgebra pointed by adding a function $x_0\colon 1 \to X$ to mark the initial state.

In general, given sets I and O, we can model transition systems whose denotational semantics are functions of type $I^* \to O$; i.e., systems which process a finite sequence of elements in I and respond with an element in O. The corresponding functor in this case is $F(X) = O \times X^I$. An $F-$coalgebra would be of the form $(X, X \xrightarrow{(\gamma,\delta)} O \times X^I)$, with $\gamma\colon X \to O$ and $\delta\colon X \to X^I$; the function o lets us explore the component in the state x, and δ lets us perform transitions. We write x/o to denote $\gamma(x) = o$, and we write x^i as a shorthand for $\delta(x)(i)$.

2.4.4 The Category of F-coalgebras and F-homomorphisms

Given a functor F, the *category of F-coalgebras*, denoted $\mathbf{Coalg_F}$ is the category whose objects are F-coalgebras and whose arrows are F*-homomorphisms*. An F*-homomorphism* $\mathbb{X} \xrightarrow{f} \mathbb{Y}$ between an F-coalgebra $\mathbb{X} = (X, c)$ and an F-coalgebra $\mathbb{Y} = (Y, d)$ is a function $f\colon X \to Y$ such that

$$F(f) \circ c = d \circ f. \tag{2.1}$$

2.4.5 Semantic Map and Final F-Coalgebras

Let F be a functor such that $F(X) = O \times (X)^I$, and let I and O be sets; we define the set $\sigma F \triangleq O^{I^*}$, and we define the function pair $(?, (\cdot)')\colon \sigma F \to O \times (\sigma F)^I$, for $\phi \in \sigma F$, $i \in I$ and $w \in I^*$, by

$$\phi? \triangleq \phi(\varepsilon), \phi'(i)(w) \triangleq \phi(i \cdot w) \tag{2.2}$$

Following convention, we write $\phi'(i)$ as ϕ^i, and the pair $(?, (\cdot)')$ as 1_F. The pair $(\sigma F, 1_F)$ is a *final F-coalgebra*, since it is a final object in the category $\mathbf{Coalg_F}$.

Given an F-coalgebra (X, c) where $c = (\gamma, \delta)$, the unique F-homomorphism between (X, c) and $(\sigma F, 1_F)$ is given by a function $!_c \colon X \to \sigma F$, which is defined for $x \in X$, $i \in I$ and $w \in I^*$ by

$$!_c(x)(\varepsilon) \triangleq \gamma(x), \quad \text{and} \quad !_c(x)(i \cdot w) \triangleq !_c(x^i)(w). \tag{2.3}$$

The function $!_c$ is called the *semantic map* because it maps $x \in X$ to its denotational semantics in σF. For example, in the case of DFA as coalgebras, it maps each state x in the carrier of the coalgebra to the language it would be recognised if x were the initial state in the DFA.

2.4.6 Bisimilarity

Given F-coalgebras (X, c) and (Y, d), we say that states $x \in X$ *and* $y \in Y$ *are bisimilar*, denoted $x \sim y$, if and only if $!_c(x) = !_d(y)$. In other words, x and y are bisimilar iff they have the same semantics.

2.4.7 Coalgebraic Specification Language

Given a functor $F(X) = O \times X^I$ where O and I are fixed sets, a *coalgebraic specification language* for F is an F-coalgebra $(\mathcal{E}, c)$ where $\mathcal{E}$ is a set of expressions whose semantics are given by the semantic map $!_c$.

For example, consider the functor $F(X) = 2 \times X^2$; we define the set of expressions $\mathcal{E}$ by the grammar

$$e ::= ID : 0?ID \wedge 1?ID \wedge \downarrow (0|1) \tag{2.4}$$

where ID is a set of identifiers. We define the pair function $c = (\gamma, \delta) \colon \mathcal{E} \to F(\mathcal{E})$ by

$$\begin{aligned} \gamma(x : 0? \wedge 1?z \wedge \downarrow a) &\triangleq a \\ \delta(x : 0?y \wedge 1?z \wedge \downarrow a)(0) &\triangleq y \\ \delta(x : 0?y \wedge 1?z \wedge \downarrow a)(1) &\triangleq z. \end{aligned}$$

The expression $x : 0?x \wedge 1?x \wedge \downarrow 1$ can be considered to describe a single-state automaton whose single state is accepting, and it loops to itself on both inputs 0 and 1. The semantic map $!_c$ maps the expression $x : 0?x \wedge 1?x \wedge \downarrow 1$ to the language 2^*.

Cite Kleene Coalgebra somehow

2.4.8 Completeness

Given a coalgebraic specification language $L = (\mathcal{E}, c)$ for the functor $F(X) = O \times X^I$, we say that L is *complete* if and only if the semantic map $!_c \colon \mathcal{E} \to \sigma F$ is surjective. In other words, for every behaviour $\phi \in \sigma F$, there exists an expression $e \in \mathcal{E}$ such that $!_c(e) = \phi$.

2.4.9 Soundness

Given a coalgebraic specification language $L = (\mathcal{E}, c)$ for the functor $F(X) = O \times X^I$ and a notion of equivalence in $\mathcal{E}$ modelled by a function $\equiv \colon \mathcal{E} \times \mathcal{E} \to 2$, we say that L is *sound* if and only if, for $e_1, e_2 \in \mathcal{E}$, if $e_1 \equiv e_2$, then $e_1 \sim e_2$; i.e., if two expressions are equivalent, then they have the same behaviour.

2.5 Coinduction

Coinduction is a dual principle to induction, and we use it to define what things are in terms of how they behave.

I'll come back to this

2.5.1 Coinduction Proof Principle

Coinductive Definitions?

2.5.2 Bisimulation

2.6 Attacker Classification via Latent Behaviour Analysis

This section may be better in its own chapter?

Cite the sources that give these definitions. Get them from the chapter.

2.6.1 And-inverter Graphs (AIGs): Input, Latches, Gates, Components

An *And-inverter Graph* is a directed graph which models a system of equations [Hel63; Kue+02]. AIGs describe hardware models at the bit-level [BHW11], and have attracted the attention of industry partners including IBM and Intel [Cab+15].

Formally, an AIG has m boolean inputs, n boolean state variables and o boolean gates. The elements in the set $W = \{w_1, \ldots, w_m\}$ represent the *inputs*, the elements in $V = \{v_1, \ldots, v_n\}$ represent the *latches*, and the elements in $G = \{g_1, \ldots, g_o\}$ represent the *and-gates*. We assume that W, V and G are pairwise disjoint, and we define the set of *components* C by $C \triangleq W + V + G$. Without loss of generality, we assume that W, V and G are range sets.

2.6.2 State and Input Vectors

The *states* of an AIG are all the different valuations of the variables in V. Formally, a state $\vec{v}$ is a vector with coordinates in V and values in 2, so the set of all states is

$\vec{V} = 2^V$. This definition only considers latches to be part of the state; gates, inputs and requirements are external to the state.

Similarly, the *input vector* is a valuation of the variables in W. An input vector $\vec{w}$ is a vector with coordinates in W and values in 2; the set of input vectors is $\vec{W} = 2^W$. For $t \in \mathbb{N}$, we denote the state of the system at time t by $\vec{v}(t)$. The initial state is $\vec{v}(0)$. Similarly, we denote the input of the system at time t by $\vec{w}(t)$. There are no restrictions or assumptions over the value of $\vec{w}(t)$.

2.6.3 Expressions

An *expression* e is described by the grammar $e ::= 0 \mid 1 \mid c \mid \neg c$, where $c \in C$. The set of all expressions is E. We use AIG expressions to describe a system of equations over discrete time steps $t = 0, 1, ...$ To each latch $v \in V$ we associate a transition equation of the form $v(t+1) = e(t)$ and an initial equation of the form $v(0) = b$, where $e \in E$ and $b \in 2$. To each gate $g \in G$ we associate an equation of the form $g(t) = e_1(t) \wedge e_2(t)$, where $e_1, e_2 \in E$.

2.6.4 Invariant Requirements

Given an expression $e \in E$, the *invariant* $\Box e$ is the property that requires $e(t)$ to be true for all $t \geq 0$. The system S *fails the invariant* $\Box e$ iff there exists a finite sequence of inputs $\vec{w}_0, \ldots, \vec{w}_t$ such that, if $\vec{w}(i) = \vec{w}_i$ for $0 \leq i \leq t$, then $e(t)$ is false. The system *satisfies* the invariant $\Box e$ if no such sequence of inputs exists. Every expression e represents a boolean predicate over the state of the latches of the system, and can be used to characterise states that are unsafe. These expressions are particularly useful in safety-critical hardware, as they can signal the proximity of a critical state.

2.6.5 Cone of Influence (COI)

This connects with model checking, information flow analysis, and other research areas. Add appropriate citations.

The *Cone-of-Influence* (COI) is a mapping from an expression to the components that can potentially influence its value. We obtain the COI of an expression $e \in E$, denoted $\blacktriangledown(e)$, by transitively tracing its dependencies to inputs, latches and gates. More precisely,

- $\blacktriangledown(0) = \emptyset$ and $\blacktriangledown(1) = \emptyset$;
- if $e = \neg c$ for $c \in C$, then $\blacktriangledown(e) = \blacktriangledown(c)$;
- if $e = w$ and w is an input, then $\blacktriangledown(e) = \{w\}$;
- if $e = v$ and v is a latch whose transition equation is $l(t+1) = e'(t)$, then $\blacktriangledown(e) = \{v\} \cup \blacktriangledown(e')$;
- if $e = g$ and g is a gate whose equation is $g(t) = e_1(t) \wedge e_2(t)$ then $\blacktriangledown(e) = \{g\} \cup \blacktriangledown(e_1) \cup \blacktriangledown(e_2)$.

The COI of a requirement $r = \Box e$ is $\blacktriangledown(r) \triangleq \blacktriangledown(e)$.

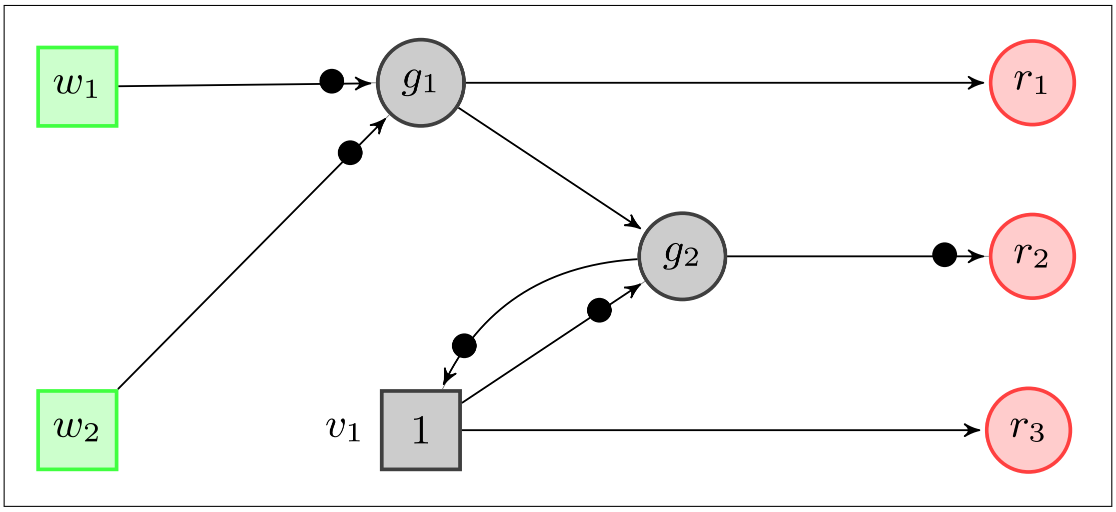

FIGURE 2.2: **Left:** And-inverter graph describing a system with two inputs w_1 and w_2 (green boxes), one latch v_1 with initial value 1 (grey box), two gates g_1 and g_2 (gray circles), and three invariant requirements $r_1 = \Box g_1$, $r_2 = \Box\neg g_2$ and $r_3 = \Box v_1$ (red circles). The arrows represent logical dependencies, and bullets in the arrows imply negation.

Example 2.1. Figure 2.2 shows an example AIG with $W = \{w_1, w_2\}$, $V = \{v_1\}$ and $G = \{g_1, g_2\}$. The corresponding system of equations is

$$\begin{aligned} v_1(0) &= 1, & v_1(t+1) &= \neg g_2(t), \\ g_1(t) &= \neg w_1(t) \wedge \neg w_2(t), & g_2(t) &= g_1(t) \wedge \neg v_1(t). \end{aligned}$$

The set states for this system is $\vec{V} = 2^{\{v_1\}}$, the inputs are $\vec{W} = 2^{\{w_1, w_2\}}$, and the initial state is $\vec{v}(0)$, where $\vec{v}(0)[v_1] = 1$.

For this example, we define three requirements: $r_1 \triangleq \Box g_1$, $r_2 \triangleq \Box\neg g_2$, and $r_3 \triangleq \Box v_1$. This system satisfies r_2 and r_3, but it fails r_1 because $w_1 = 1$ and $w_2 = 0$ results in $g_1(0)$ being 0.

In Example 2.1, the COI for the requirements are $\blacktriangledown(\Box g_1) = \{g_1, w_1, w_2\}$, and $\blacktriangledown(\Box g_2) = \blacktriangledown(\Box g_3) = \{g_1, g_2, v_1, w_1, w_2\}$.

The set of *sources* of an expression $e \in E$, denoted $\texttt{src}(e)$ is the set of latches and inputs in the COI of e; formally, $\texttt{src}(e) \triangleq \blacktriangledown(e) \cap (V \cup W)$. The *Jaccard index* of two expressions e_1 and e_2 is equal to $\frac{|\texttt{src}(e_1) \cap \texttt{src}(e_2)|}{|\texttt{src}(e_1) \cup \texttt{src}(e_2)|}$. This index provides a measure of how similar the sources of e_1 and e_2 are.

The *dual cone-of-influence* (IOC) of a component $c \in C$, denoted $\blacktriangle(c)$, is the set of components influenced by c; more precisely $\blacktriangle(c) \triangleq \{c' \in C | c \in \blacktriangledown(c')\}$.

2.7 Cyber-Physical System

This section probably makes more sense in its own chapter, when we explain what the latent behaviour of a CPS is.

2.8 Side Channels

Chapter 3

Foundations of Latent Behaviour Analysis

3.1 Introduction

A *latent behaviour* is a functionality that is not coded into a system, so the system normally does not display it, but given extraordinary circumstances, the system displays such functionality. To reveal a latent behaviour, we transform the state space of the system. Normally, the state space is not transformed, so we can model "extraordinary circumstances" using these transformations. This concept is probably well illustrated by *return-oriented programming* (ROP) as described in [Sha07], where the state of the call stack is transformed to alter the behaviour of the program being executed, but the program itself is not modified.

In ROP, the adversary manipulates the call stack to string code gadgets together and arbitrarily execute code. However, if the program is very small and does not have enough gadgets, then the computational power of the attacker is restricted, since it becomes impossible to create certain behaviours. This restriction is similar to the restrictions we face when working with non-Turing complete languages (e.g., we cannot use regular expressions to specify arbitrary Turing machines). However, given enough gadgets, we can reveal any behaviour we want if we appropriately compose the gadgets.

In this section, we explain the commutative diagram presented in Figure 3.1. This diagram, which we refer to as the “Arrow" of latent behaviour analysis, illustrates how spatial and dynamics transformations affect the behaviour of an arbitrary F-coalgebra (X, c) by changing the semantic map from $!_c$ to $!_{b \circ c \circ m}$. The intuition behind the diagram of the Arrow is the following. Normally, an F-coalgebra (X, c) gives semantics to the elements in X via the semantic map $!_c$. When affected by a spatial transformation $m\colon X \to X$ and a dynamics transformation $b\colon F(X) \to F(X)$, the F-coalgebra (X, c) reveals an F-coalgebra $(X, b \circ c \circ m)$. The behaviour of elements in $(X, b \circ c \circ m)$ may bear no resemblance to the original behaviour. At most, we can say that if m and b preserve bisimilarity, then there exists a transformation $\omega\colon \sigma F \to \sigma F$ in the carrier of the final F-coalgebra $(\sigma F, 1)$ which maps the original behaviours of the system into the new behaviours revealed by m and b.

$$\begin{tikzcd}
 & \sigma F \arrow[dr, "\omega"] & & \\
X \arrow[dd, "c"] & X \arrow[l, "m"'] \arrow[u, "!_c"] \arrow[r, dotted, "!_{bocom}"'] & \sigma F \arrow[dr, "\mathbf{id}"] \arrow[dd, "1_F"] & \\
 & & & \sigma(F) \\
F(X) \arrow[r, "b"'] & F(X) \arrow[r, dotted, "F(!_{bocom})"] \arrow[d, "F(!_c)"'] & F(\sigma F) \arrow[ur, "1_F^{-1}"'] & \\
 & F(\sigma F) \arrow[ur, "F(\omega)"'] & &
\end{tikzcd}$$

FIGURE 3.1: "The Arrow of Latent Behaviour Analysis." This commutative diagram summarises the effect of spatial and dynamics transformations over the F-coalgebra (X, c), respectively modelled by m and b: the semantic map changes from $!_c$ to $!_{bocom}$. Through this work, we assume $b = \mathtt{id}$.

3.2 Motivational Example

Consider the automaton shown in Figure 3.2, which recognises the language of sequences in 2^* that end in two consecutive 1; i.e., the language $(0|1)^*11$. This automaton is defined by the tuple $(\vec{X}, \vec{x}_0, \delta, F)$; the carrier is $\vec{X} \triangleq 2 \times 2$, the initial state selector is $\vec{x}_0 \colon 1 \to \vec{X}$ with $\vec{x}_0(0) \triangleq (0, 0)$, the transition function is $\delta \colon \vec{X} \to 2 \to \vec{X}$, defined for $\vec{x} \in \vec{X}$ and $i \in 2$ by $\delta(\vec{x})(i) \triangleq (i, \vec{x}[0])$, and the characteristic predicate of the set of accepting states is $F \colon \vec{X} \to 2$, where $F(x, y) \triangleq x \wedge y$ (i.e. $(1, 1)$ is the only accepting state). This automaton is not minimal, since the states $(0, 0)$ and $(0, 1)$ are different, yet they are bisimilar.

Now, consider a transformation $m \colon \vec{X} \to \vec{X}$, defined by $m(\vec{x}) \triangleq (\neg\vec{x}[0], \neg\vec{x}[1])$ for $\vec{x} \in \vec{X}$; i.e., m maps (a, b) to $(\neg a, \neg b)$. This transformation models some fault which could be introduced by a malicious programmer or by an attacker which corrupts the state of the system. Due to this transformation, if the current state is $\vec{x} = (a, b)$, when we intend to check whether $\vec{x}$ is accepting, we check whether $(\neg a, \neg b)$ is accepting instead, and when we intend to compute $\delta(\vec{x})(i)$, we instead compute $\delta(\neg a, \neg b)(i)$.

Figure 3.3 shows a model of the transformed automaton. This automaton recognises the language of sequences that are either empty or end in 10; i.e., $\varepsilon|(0|1)^*10$. Before we explain why the automaton in Figure 3.3 is in fact the resulting system when we apply the transformation m, we can first test if the transformed system recognises sequences $\varepsilon|(0|1)^*10$ due to the transformation m. To do so, let us consider pairs of states $[\vec{x}, \vec{y}]$ where $\vec{x} = (x_1, x_2)$ and $\vec{y} = (y_1, y_2)$ such that the pair $\vec{x}$ follows the original behaviour while $\vec{y}$ follows the new behaviour. Let $[(0, 0), (0, 0)]$ be the initial state, and let $\delta_i \colon \vec{X} \to$

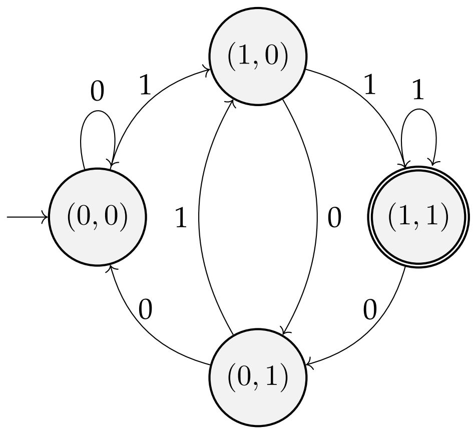

FIGURE 3.2: An automaton which recognises the language $(0|1)^*11$.

$\vec{X}$ be the functions defined by $\delta_i(\vec{x}) = \delta(\vec{x})(i)$ for $i \in 2$ (i.e., δ_0 is a partial application of δ given 0 and δ_1 is a partial application of δ given 1); the trace of the sequence 00 is

$$\begin{aligned}[(0,0),(0,0)] &\xrightarrow{\mathtt{id}\times m} [(0,0),(1,1)] \xrightarrow{\delta_0\times\delta_0} [(0,0),(0,1)] \\ &\xrightarrow{\mathtt{id}\times m} [(0,0),(1,0)] \xrightarrow{\delta_0\times\delta_0} [(0,0),(0,1)] \\ &\xrightarrow{\mathtt{id}\times m} [(0,0),(1,0)] \xrightarrow{F\times F} [0,0],\end{aligned}$$

so 00 is rejected by both automata. Now, if we receive the sequence 10, the resulting state trace is

$$\begin{aligned}[(0,0),(0,0)] &\xrightarrow{\mathtt{id}\times m} [(0,0),(1,1)] \xrightarrow{\delta_1\times\delta_1} [(1,0),(1,1)] \\ &\xrightarrow{\mathtt{id}\times m} [(1,0),(0,0)] \xrightarrow{\delta_0\times\delta_0} [(0,1),(0,0)] \\ &\xrightarrow{\mathtt{id}\times m} [(1,0),(1,1)] \xrightarrow{F\times F} [0,1];\end{aligned}$$

the transformed automaton accepts 10, but the original automaton does not. The trace of the sequence 11 is

$$\begin{aligned}[(0,0),(0,0)] &\xrightarrow{\mathtt{id}\times m} [(0,0),(1,1)] \xrightarrow{\delta_1\times\delta_1} [(1,0),(1,1)] \\ &\xrightarrow{\mathtt{id}\times m} [(1,0),(0,0)] \xrightarrow{\delta_1\times\delta_1} [(1,1),(1,0)] \\ &\xrightarrow{\mathtt{id}\times m} [(1,1),(0,1)] \xrightarrow{F\times F} [1,0],\end{aligned}$$

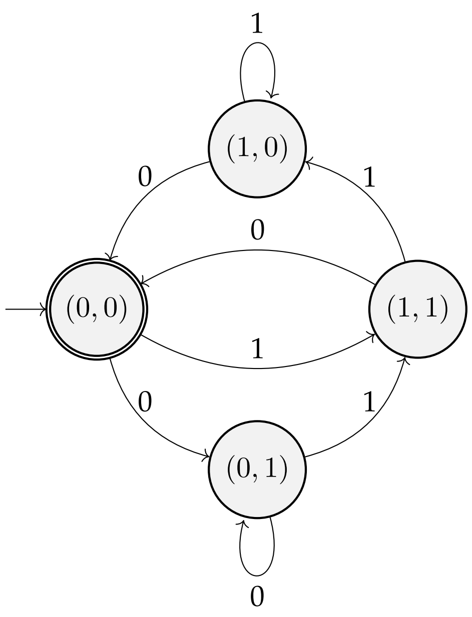

FIGURE 3.3: Automaton that models the transformed automaton, which now recognises $\varepsilon|(0|1)^*10$.

so 11 is accepted by the original automaton, but rejected by the transformed automaton. For the sequence 110, the resulting state trace is

$$\begin{aligned}[(0,0),(0,0)] &\xrightarrow{\mathtt{id}\times m} [(0,0),(1,1)] \xrightarrow{\delta_1\times\delta_1} [(1,0),(1,1)]\\ &\xrightarrow{\mathtt{id}\times m} [(1,0),(0,0)] \xrightarrow{\delta_1\times\delta_1} [(1,1),(1,0)]\\ &\xrightarrow{\mathtt{id}\times m} [(1,0),(0,1)] \xrightarrow{\delta_0\times\delta_0} [(0,1),(0,0)]\\ &\xrightarrow{\mathtt{id}\times m} [(0,1),(1,1)] \xrightarrow{F\times F} [0,1];\end{aligned}$$

the transformed automaton accepts 110, but the original automaton does not. Finally, the original automaton rejects the empty sequence ε, but the transformed automaton accepts it, since

$$[(0,0),(0,0)] \xrightarrow{\mathtt{id}\times m} [(0,0),(1,1)] \xrightarrow{F\times F} [0,1].$$

We obtain the automaton shown in Figure 3.3 by "copying" the original behaviour from images of m to their preimages. Figure 3.4 shows this procedure for $(0,0)$ and $(1,1)$. This includes copying the behaviour under δ and under F.

We preserve the initial state because $\vec{x}_0$ is a selection/construction/algebraic operation, not a dynamics/behavioural/coalgebraic operation. We do not apply transformations to the results of algebraic operations to avoid aggregating the transformation erroneously due to composition of algebraic and coalgebraic operations. Consider the followign: the initial state $\vec{x}_0$ is of type $1 \to \vec{X}$ which is algebraic, so the only way to apply m to $\vec{x}_0$ is by composing it on the left, i.e., $m \circ \vec{x}_0(0)$. Checking if the initial state is

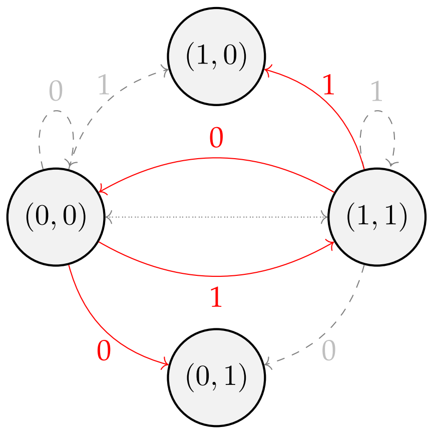

FIGURE 3.4: Composition of the fault m and the original behaviour for the states $(0, 0)$ and $(1, 1)$. The dotted, gray, bidirectional arrow in the centre models the effect of the fault m. The original behaviours appear as dashed, grey lines. The behaviour which results from the composition appears as solid, red lines.

accepting in the original automaton corresponds to the expression $(F \circ \vec{x}_0)(*)$; however, the expression

$$(F \circ m) \circ (m \circ \vec{x}_0)(0) = (F \circ m)(1, 1) = F(0, 0) = 0,$$

applies m twice, and it fails to properly check if the initial state of the transformed automaton is final. Instead, the correct expression is

$$(F \circ m)(\vec{x}_0) = (F \circ m)(0, 0) = F(1, 1) = 1.$$

We say that the transformed system is *latent* with respect to the original system, since the application of the transformation m reveals the new behaviour. In general, we cannot reveal arbitrary behaviours only through transformations of the state space; e.g., we cannot obtain an automaton that accepts every sequence from a single-state automaton that rejects all sequences by just transforming the state space.

In the following, we present a general treatment for spatial transformations and latent behaviours in the context of F-coalgebras. We describe the limitations on revealing new behaviours through spatial transformations in arbitrary F-coalgebras, and we also describe the necessary conditions that lift these limitations.

3.3 Latent F-coalgebras and Latent Behaviours

Given an F-coalgebra $(X, c\colon X \to F(X))$, each state $x \in X$ has an associated dynamics function $c(x) \in F(X)$. In this section, we study the effect of a state transformation $m\colon X \to X$ over the F-coalgebra (X, c), which changes the dynamics of $x \in X$ from $c(x)$ to $c \circ m(x)$.

In terms of types, we can always compose the function c with any carrier transformation m; the composition $c \circ m\colon X \to F(X)$ exists and is well defined. However, the behaviour of elements might be greatly affected. In particular, states which are bisimilar under c might no longer be bisimilar under $c \circ m$. If m preserves bisimilarity, then we say that m is *behaviourally consistent*.

Definition 3.3.1 (Behaviourally Consistent Spatial Transformations)**.** Given an F-coalgebra (X, c), any function of type $m\colon X \to X$ is a *spatial transformation*. A spatial transformation m is *behaviourally consistent* if and only if, whenever $x \sim_c y$, then $m(x) \sim_c m(y)$, for all $x, y \in X$.

Corollary 3.1. If (X, c) is a minimal F-coalgebra, then every spatial transformation is behaviourally consistent.

Applying transformation function $m\colon X \to X$ in the context of an F-coalgebra (X, c) changes the normal behaviour of the states in X, and it reveals the *latent coalgebra of* (X, c) *under* m.

Definition 3.3.2 (Latent Coalgebra and Latent Behaviour)**.** Given an F-coalgebra (X, c) and a transformation m, the *latent coalgebra of* (X, c) *under* m is $(X, c \circ m)$.

The semantic map $!_{c \circ m}$ of the latent coalgebra $(X, c \circ m)$ characterises the *latent behaviours* revealed m.

Example 3.1. Consider the automaton from Section 3.2 which recognises the language $(0|1)^*11$. Let F be the functor $F(X) = 2 \times X^2$; we model the automaton with an F-coalgebra $(\vec{X}, (F, \delta))$, where $\vec{X} = 2 \times 2$ and $(F, \delta)\colon X \to 2 \times X^2$ is a function pair, defined for $(x, y) \in \vec{X}$ by

$$F(x, y) \triangleq x \wedge y \tag{3.1}$$

$$\delta(x, y)(i) \triangleq (i, x). \tag{3.2}$$

Just like its automaton counterpart, this F-coalgebra is not minimal, since $(0, 0)$ and $(0, 1)$ are bisimilar There are $|X|^{|X|} = 256$ different spatial transformations, but that does not imply the existence of 256 different latent coalgebras; e.g., the transformations $\Delta_{(0,0)}$ and $\Delta_{(0,1)}$ which respectively map all states to $(0, 0)$ and $(0, 1)$ yield isomorphic latent coalgebras, since $(0, 0)$ and $(0, 1)$ are bisimilar; both of these latent coalgebras recognise the empty language starting in all states. The transformation $m(x, y) = (\neg x, \neg y)$ reveals the latent coalgebra where

$$(F \circ m)(x, y) = \neg x \wedge \neg y \tag{3.3}$$

$$(\delta \circ m)(x, y)(i) = (i, \neg x). \tag{3.4}$$

The behaviour of a state x changes when m acts on x. In particular, the image of state $(0, 0)$ under the semantic map is no longer the language $(0|1)^*11$; it is the language $\varepsilon|(0|1)^*10$ instead.

$$\begin{tikzcd}
\sigma F \arrow[r, "m"] \arrow[d, "1_F"'] \arrow[dr, "c"] & \sigma F \arrow[d, "1_F"] \\
F(\sigma F) \arrow[r, "b"'] & F(\sigma F)
\end{tikzcd}$$

FIGURE 3.5: Every F-coalgebra $(\sigma F, c)$ can be revealed from the final coalgebra $(\sigma F, 1_F)$ by means of a transformation m or a transformation b. In other words, it suffices to use spatial transformations m and the final F-coalgebra to reveal all F-coalgebras of σF, so dynamics transformations b are unnecessary.

3.4 Latent Behaviour Analysis

LBA is the study of the effects that a spatial transformation $m\colon X \to X$ has over the behaviour of states in X in the context of an F-coalgebra $(X, X \xrightarrow{c} F(X))$. The choice to focus on spatial transformations seems arbitrary; clearly, it is also possible to affect the behaviour of states by composing c with a dynamics transformation $b\colon F(X) \to F(X)$ to obtain the new dynamics function $b \circ c$, as shown in the "Arrow" from Figure 3.1. However, there are systems for which spatial and behavioural transformations can reveal the same behaviours; moreover, we can reveal any arbitrary behaviours via spatial transformations: *final F-coalgebras*, and *sound and complete coalgebraic specification languages*.

A final F-coalgebra $(\sigma F, 1_F)$ has a dynamics function $1_F\colon \sigma F \to F(\sigma F)$ that is an isomorphism. Since $\sigma F \simeq F(\sigma F)$, it naturally follows that

$$\sigma F \to \sigma F \simeq \sigma F \to F(\sigma F) \simeq F(\sigma F) \to F(\sigma F). \tag{3.5}$$

In other words, in the carrier of the final F-coalgebra, spatial transformations (of type $\sigma F \to \sigma F$), dynamics transformations (of type $F(\sigma F) \to F(\sigma F)$) and F-coalgebras (of type $\sigma F \to F(\sigma F)$) are in a one-to-one correspondence.

Intuition tells us that $1_F\colon \sigma F \to F(\sigma F)$ should correspond to $\mathtt{id}_{\sigma F}\colon \sigma F \to \sigma F$ and should correspond to $\mathtt{id}_{F(\sigma F)}\colon F(\sigma F) \to F(\sigma F)$. The general correspondence is given by the following proposition.

Proposition 3.4.1. For every F-coalgebra $(\sigma F, c)$, there are transformations $m\colon \sigma F \to \sigma F$ and $b\colon F(\sigma F) \to F(\sigma F)$ such that the diagram in Figure 3.5 commutes.

Proof. Since $1_F\colon \sigma F \to F(\sigma F)$ is an isomorphism, by taking $m = 1_F^{-1} \circ c$ and $b = c \circ 1_F^{-1}$, the diagram in Figure 3.5 commutes. □

Corollary 3.2. Every F-coalgebra $(\sigma F, c)$ is a latent coalgebra of $(\sigma F, 1_F)$ under some spatial transformation $m\colon \sigma F \to \sigma F$.

This property applies to the carrier of the final coalgebra because of the reversibility of the final dynamics map 1_F, which formalises a correspondence between state and behaviour. This property is also shared by sound and complete coalgebraic specification languages, since their semantic map is an epimorphism. For an arbitrary F-coalgebra (X, c), only some latent F-coalgebras can be revealed by spatial transformations $m\colon X \to X$. It would not be an exaggeration to state that F-coalgebras which satisfy this property have all the "necessary gadgets" (with respect to the functor F) to reveal any arbitrary functionality using spatial transformations.

Given a final F-coalgebra $(\sigma F, 1_F)$ and an arbitrary coalgebra $(\sigma F, c)$, the spatial transformation $\omega\colon \sigma F \to \sigma F$, defined by $\omega \triangleq 1_F^{-1} \circ c$, implements c. The identity spatial transformation implements 1_F. This duality between F-coalgebras and spatial transformations in σF enables a natural composition of F-coalgebras in σF.

Given two F-coalgebras $(\sigma F, c_1)$ and $(\sigma F, c_2)$ whose respective spatial transformation implementations are ω_1 and ω_2, the composition $\omega_2 \circ \omega_1$ gives rise to the following equivalent concepts:

- the F-coalgebra $(\sigma F, c_2 \circ 1_F^{-1} \circ \omega_1)$,
- the latent coalgebra of $(\sigma F, c_2)$ under ω_1,
- the latent coalgebra of $(\sigma F, 1_F)$ under $\omega_2 \circ \omega_1$.

In summary, we can represent F-coalgebras whose carrier is σF by endofunctions in σF, and we can reason about their composition as we would with functions in the monoid of endofunctions $(\sigma F \to \sigma F, \circ, \mathtt{id})$.

In the following, we return to arbitrary F-coalgebras (X, c), but we keep in mind the following: when we reveal a latent coalgebra of (X, c) under a spatial transformation m, we are implicitly sequentially composing the behaviour function c with the behaviour function of the F-coalgebra implemented by m; first we apply m and then we apply c.

3.5 Some Applications of Latent Behaviour Analysis

We study the effects that spatial transformations have on the behaviour of the system. Just like in geometry, some spatial transformations may preserve some properties of the system, while others may break them. In this section, we present three ideas for LBA to study three aspects of systems: *quantifying and improving robustness*, *classification of attacker models*, and *side-channel repair*.

3.5.1 Quantifying and Improving Robustness

Informally, the robustness of a system quantifies how much stress the system can withstand without compromising its functionality.

Find some nice definitions of robustness.

In this section, we explore the idea of modelling stress via spatial transformations, and how we can quantify and improve the robustness of a system by studying properties of latent coalgebras.

To formally capture the notion of whether the functionality of a system is compromised, we use *behavioural properties*.

Definition 3.5.1 (Behavioural Property)**.** Let F be a functor for which a final F-coalgebra $(\sigma F, 1)$ exists. A *behavioural property* P is a function $P\colon \sigma F \to 2$. Given an arbitrary F-coalgebra (X, c), we say that $x \in X$ satisfies the behavioural property P if and only if $(P \circ !_c)(x) = 1$.

Behavioural properties come in all shapes and forms, and are often described using logics like LTL [**LTL**]. CTL [**CTL**], and μ calculus [**MuCalculus**]. Let us for now assume that we have a set of behavioural properties $\mathcal{R} = \{R_1, \dots, R_n\}$ which model both functional and security requirements of the system. We discuss concrete ways to define these behavioural properties in Chapters 4 and **??**.

Consider the following problem: we are given an arbitrary F-coalgebra (X, c), an initial state $x_0 \in X$, and a set of behavioural properties $\mathcal{R} = \{R_1, \dots, R_n\}$. Normally, we would check if x_0 satisfies all behavioural properties in $\mathcal{R}$, but now we consider a spatial transformation $m\colon X \to X$, and we want to check if x_0 still satisfies all the given behavioural properties in the resulting latent coalgebra revealed by m.

We might think that it suffices to check $(R_i \circ !_c)(m(x_0)) = 1$ for each requirement R_i, but that would be incorrect as illustrated by the following example.

Example 3.2 (Wrong Verification)**.** In the context of Example 3.1, the functor is $F(X) = 2 \times X^2$, and it has a final F-coalgebra $(2^{2^*}, (\varepsilon?, (\cdot)'))$ where $\varepsilon?\colon 2^{2^*} \to 2$ checks whether the empty sequence ε belongs to the language and $(\cdot)'\colon 2^{2^*} \to (2 \to 2^{2^*})$ computes the Brzozowski derivative [**BrzozowskiDerivative**] of the language.

Let $R\colon 2^{2^*} \to 2$ be the behavioural property defined by $R(\phi) \triangleq \phi \sim \varepsilon|(0|1)^*10$, for $\phi \in 2^{2^*}$. The property R is not satisfied by any state of the original F-coalgebra, so $(R \circ !_c)(m(x_0))$ would be 0 for all spatial transformations m; nevertheless, R is satisfied by x_0 in the latent coalgebra revealed by the spatial transformation $m(x, y) = (\neg x, \neg y)$ used in Example 3.1, since it is the language the state $(0, 0)$ recognises in the latent coalgebra.

Part of the novelty of LBA is that we can study a latent coalgebra $(X, c \circ m)$ just as we would study the original coalgebra (X, c). By this, we mean that we can apply testing and verification techniques without complication. In the following, we illustrate how we can use spatial transformations to model attacks which target the state of systems rather than the program, and we study the details of this approach in Chapter **??** when we apply it to cyber-physical systems.

Definition 3.5.2 (State/Behaviour-based Attacks)**.** Given an F-coalgebra (X, c), we define the set of its *state-based attacks* corresponds to be the set of its spatial transformations, i.e., X^X. We also define the set of its *behaviour-based attacks* by $F(X)^{F(X)}$.

Under this definition, LBA studies the effect of state-based attacks. Henceforth, when we refer to an attack, we implicitly mean a state-based attack (unless stated otherwise). In the particular case of final F-coalgebras, their state-based attacks coincide with their behaviour-based attacks.

We know that an attack m reveals a latent coalgebra $(X, c \circ m)$ which may or may not satisfy the same behavioural properties that (X, c) satisfies. We could check whether $(X, c \circ m)$ satisfies all requirements in $\mathcal{R}$, but this is not very informative if we arbitrarily choose m. We propose to systematically generate a set of spatial transformations $\mathcal{M} = \{m_1, \ldots, m_p\}$ based on some attacker model, and test whether each revealed latent coalgebra $(X, c \circ m_i)$ satisfies each requirement R_j for $m_i \in \mathcal{M}$ and $R_j \in \mathcal{R}$. If a requirement R is not satisfied in some latent coalgebra $(X, c \circ m)$, then the attacker can use m to break R in the original system (X, c); in this case, we say that *the attack m breaks the requirement R*. If no attack in $\mathcal{M}$ breaks any requirement in $\mathcal{R}$, then we say that the system is *robust* against the attacker model that generated $\mathcal{M}$. This approach gives us a qualitative notion of robustness.

We can extend qualitative robustness into quantitative robustness by discounting broken requirements. More precisely, given (X, c), $\mathcal{M}$, and $\mathcal{R}$ such that $|\mathcal{R}| = n$, we can define the set $\mathcal{M}[\mathcal{R}]$ of requirements in $\mathcal{R}$ that are broken by one or more attacks in $\mathcal{M}$; similarly $m[\mathcal{R}]$ is the set of requirements that m breaks. We estimate the robustness of the system by the formula

$$\sum_{i=i}^{n} w_i(1 - (R_i \in \mathcal{M}[\mathcal{R}])),$$

where $w_i \in \mathbb{R}^+$ is a weight which models the importance of requirement R_i.

Let us consider the case where (X, c) fails a non-empty set of requirements; this set, given the definitions, is equal to $\mathtt{id}[\mathcal{R}]$. Now, let $\mathcal{K} = \{k_1, \ldots, k_q\}$ be an arbitrary set of spatial transformations independent of $\mathcal{M}$; we say that a spatial transformation $k \in \mathcal{K}$ *repairs* (X, c) if $k[\mathcal{R}] \subset \mathtt{id}[\mathcal{R}]$; we say that k *fully repairs* (X, c) if $k[\mathcal{R}]$ is empty. The set spatial transformations $\mathcal{K}$ corresponds to a repair toolkit, which we generate systematically just like we did with the set of attacks $\mathcal{M}$. We study the process of system repair in more detail in Chapter **??**.

We now consider an analysis that combines both $\mathcal{M}$ and $\mathcal{K}$ as follows: if an attack $m \in \mathcal{M}$ breaks some requirements, is there a *counter attack k* that fully repairs $(X, c \circ m)$? If so, we can improve the robustness of the system (X, c) by transforming it into $(X, c \circ m \circ k)$ whenever we detect that the attack m is affecting (X, c); we refer to this dependent notion of robustness as *latent robustness*, and we explore this concept in cyber-physical systems during Chapter **??**.

Example 3.3 (Counter Attack)**.** In the context of Example 3.1, consider the behavioural property $Q\colon 2^{2^*} \to 2$ which accepts a language $\phi \in 2^{2^*}$ if and only if ϕ accepts a sequence $w \in 2^*$, then w ends in 1 or w is empty.

In the original F-coalgebra, all states in 2×2 satisfy the property Q since both $(0, 0)$ and $(1, 0)$ recognise the language $(0|1)^*11$, $(0, 1)$ recognises the language $(0|1)^*1$, and $(1, 1)$ recognises $\varepsilon + 0(0|1)^*11 + 1(0|1)^*1$. However, in the latent coalgebra revealed by the attack m, which maps (a, b) to $(\neg a, \neg b)$, the state $(0, 0)$ no longer satisfies Q, since it now recognises the language $\varepsilon|(0|1)^*10$.

Now, if we use m again in the latent coalgebra $(2\times 2, c\circ m)$ to reveal the latent coalgebra $(2\times 2, c\circ m\circ m)$, we repair the system with respect to the property Q, since $m\circ m = \mathtt{id}$. Thus, m can be used as its own counter attack, but only when the latent coalgebra $(2\times 2, c\circ m)$ has been revealed (otherwise, the application of m reveals $(2\times 2, c\circ m)$, and it breaks Q).

3.5.2 Classification of Attackers

We have not discussed how we systematically generate attacks and counter attacks for an F-coalgebra (X, c). Both attacks and counter attacks are spatial transformations, so they are elements of the set X^X. Our goal is the following: given an *attacker model* (whatever that is), we want to automatically obtain a set of attacks $\mathcal{M}$, which we can use to perform LBA on the F-coalgebra (X, c). We can use the results of this analysis to quantify robustness as shown in Chapter **??**. In this section, we are interested in also varying the attacker model, and in comparing them.

Consider an F-coalgebra $(\vec{X}, c)$ such that $\vec{X}$ is a product type of finite types; e.g., $\vec{X} = 2\times 2$. We use the coordinates of $\vec{X}$, e.g., `fst` and `snd` to define attacker models.

Definition 3.5.3 (Attacker Model)**.** Let $(\vec{X}, c)$ be an F-coalgebra such that $\vec{X} = X_1 \times \ldots \times X_n$ is a product type whose coordinates are $\Pi = \{\pi_1, \ldots, \pi_n\}$ where $\pi_i \colon \vec{X} \to X_i$, and let α be a set of coordinates, with $\alpha \subseteq \Pi$; the *attacker model characterised by* α is the set of transformations in $\vec{X}$, denoted $\langle\alpha\rangle$, such that $m\colon \vec{X} \to \vec{X}$ is an element of $\langle\alpha\rangle$ if and only, for all states $\vec{x} \in \vec{X}$ and coordinates $\pi \in \Pi$,

$$m(\vec{x})[\pi] \neq \vec{x}[\pi] \text{ implies that } \pi \in \alpha. \tag{3.6}$$

In other words, if m modifies any state $\vec{x}$ at coordinate π, then π must be in α. Note that the identity transformation is in every attacker model, since the modification requirement is not strict; what an attacker fitting the model characterised by α cannot do is to modify a state in a coordinate that is not in α. In Chapters 4 and **??**, we show how to systematically generate attacks for an attacker fitting a particular attacker model; more precisely, we show how to automatically generate elements of $\langle\alpha\rangle$.

We give attackers fitting an attacker model characterised by α full control over the coordinates in α. More precisely, for $0 \leq i \leq n$, if $\pi_i\colon \vec{X} \to X_i$ is a coordinate in α, then, for every value in $x_i \in X_i$, the attackers fitting the model characterised by α can always find a transformation m in $\langle\alpha\rangle$ such that $m(\vec{x})[\pi_i] = x_i$, for all $\vec{x} \in \vec{X}$. We did not find any practical scenarios where restricting the control of attackers over a particular coordinate offered an advantage over attackers with full control over such a coordinate.

Example 3.4 (Some Invariant Attacker Models for Example 3.1)**.** Since $\vec{X} = 2\times 2$, we have two coordinates: $\mathtt{fst}\colon \vec{X} \to 2$ and $\mathtt{snd}\colon \vec{X} \to 2$ mapping $(x, y) \xmapsto{\mathtt{fst}} x$ and $(x, y) \xmapsto{\mathtt{snd}} y$. The trivial attacker model is an attacker that does nothing because it cannot transform states in any component, and it corresponds to the attacker model for the empty subset of coordinates $\emptyset$. An attacker fitting the model characterised by $\emptyset$ can only use the `id` spatial transformation, since $\langle\emptyset\rangle = \{\mathtt{id}\}$.

We consider three other attacker models now: one which has full control over `fst` but no control over `snd` (i.e., the attacker model characterised by $\{\texttt{fst}\}$), one which has full control over `snd` but not over `fst` (i.e., the attacker model characterised by $\{\texttt{snd}\}$), and one attacker that has full control over both `fst` and `snd` (i.e., the attacker model characterised by $\{\texttt{fst}, \texttt{snd}\}$). The set $\langle\{\texttt{fst}\}\rangle$ is the set of endofunctions in $\vec{X}$ which only affect the `fst` coordinate (if any). Dually, the set $\langle\{\texttt{snd}\}\rangle$ is the set of endofunctions in $\vec{X}$ which only affect the `snd` coordinate (if any). Finally, the set $\langle\{\texttt{fst}, \texttt{snd}\}\rangle$ is the set of all endofunctions $\vec{X}^{\vec{X}}$, so attackers fitting this model can use any arbitrary spatial transformations.

In Chapters 4 and **??**, we compare and classify attackers based on their model. The intuition is the following: given an F-coalgebra $(\vec{X}, c)$ where $\vec{X}$ is a product type of finite types whose coordinates are $\Pi = \{\pi_1, \ldots, \pi_p\}$, and a list of requirements $\mathcal{R} = \{R_1, \ldots, R_n\}$, we can systematically generate attacker models by choosing a subset of coordinates in Π. Under this formulation, the set of attacker models is characterised by $\mathscr{P}(\Pi)$. Since attacker models are characterised by sets, we can compare them via set inclusion. This comparison sorts attackers by *capabilities*; i.e., it classifies attackers with respect to the ways they can interact with the system. Then, using LBA, we obtain the set of requirements $\langle\alpha\rangle[\mathcal{R}]$ that the attacker model $\langle\alpha\rangle$ can break, and we compare these sets of requirements by set inclusion to compare attackers by their *power*; i.e., for two attacker models $\langle\alpha\rangle$ and $\langle\beta\rangle$, if $\langle\alpha\rangle[\mathcal{R}] \subset \langle\beta\rangle[\mathcal{R}]$, then $\langle\beta\rangle$ breaks strictly more requirements than $\langle\alpha\rangle$, making attackers which fit the model $\langle\beta\rangle$ more dangerous adversaries than those which fit the model $\langle\alpha\rangle$.

3.5.3 Side-Channel Repair

The last application of LBA that we consider is of program repair. We briefly mentioned the idea in Section 3.5.1: given an F-coalgebra (X, c) and a set of requirements $\mathcal{R} = \{R_1, \ldots, R_n\}$, if $\langle\emptyset\rangle[\mathcal{R}]$ is non-empty (i.e. the system fails some requirements), then we can look for spatial transformations to reveal a latent behaviour $(X, c \circ m)$ such that $\{m\}\,[\mathcal{R}]$ is empty. We consider an interesting case where the state x is a program, and R consists of a single requirement: constant memory access patterns. We propose an idempotent spatial transformation $\mathcal{O}$ which maps x to a state which satisfies constant memory access patterns. In Chapter **??**, we show how we can use $\mathcal{O}$ to repair a family of timing side-channel leaks for LLVM programs.

Chapter 4

Systematic Classification of Attacker Models

4.1 Introduction

Problem Statement Some systems are designed to provide security guarantees in the presence of attackers. For example, the Diffie-Hellman key agreement protocol guarantees *perfect forward secrecy* (PFS) [Gün90; MVO96]. PFS is the security property which guarantees that the session key remains secret even if the long-term keys are compromised. These security guarantees are only valid in the context of the attacker models for which they were proven; it is unknown whether those guarantees apply for stronger or incomparable attacker models. For instance, PFS describes an attacker model, say $\mathcal{M}_{DH}$, that can compromise the long-term keys *and only those*, and it also describes a property (i.e., the confidentiality of the session keys) that is guaranteed in the presence of an attacker that fits the model $\mathcal{M}_{DH}$. However, if we consider a stronger attacker model (e.g., an attacker that can directly compromise the session key), then Diffie-Hellman can no longer guarantee the confidentiality of the session keys. It is difficult to provide any guarantees against an attacker model that is too proficient/powerful, so it is in the interest of the system designer to choose an adequate attacker model that puts the security guarantees of the system in the context of realistic and relevant attackers.

While $\mathcal{M}_{DH}$ is an attacker model that describes attackers who compromise confidentiality, we are interested in attacker models that characterise attackers who compromise integrity. We associate the security guarantees with respect to attacker models that target integrity with *robustness*. We recall Research Questions 1.1, 1.2, and 1.3, and consider the following hypothetical scenario: we are given a system and a list of security requirements R, we are also given a mechanism which systematically generates attackers, and a method to check for each generated attacker A which requirements in R fail in the presence of A; we denote such set of failed requirements by $A[R]$. Now, for two attackers A and B, we can compare $A[R]$ and $B[R]$ since they are both subsets of R, and we would say that A is more powerful than B if $B[R] \subseteq A[R]$, since A breaks at least the same requirements that B can, but maybe more. This methodology offers a possible solution to Research Questions 1.2 and 1.3. We now need to find a way to systematically generate attackers and decide which security requirements they break.

And-inverter Graphs (AIGs) and their Attacker Models. We first approached the problem of systematically generating attackers in Chapter 3. In summary, given an F-coalgebra $(\vec{X}, c)$ whose carrier is a product type $\vec{X}$ with coordinates Π that models the system, we can systematically generate attacker models by choosing a set of coordinates $\alpha \subseteq \Pi$, following Definition 3.5.3; the resulting attacker model is the set of endofunctions in $\vec{X}$ which only affect the coordinates in α. Now, in this chapter, we adapt this notion for AIGs by modelling them as F-coalgebras of some functor F; that way, we can use Definition 3.5.3 and systematically generate attacker models for AIGs.

Due to being systems described at bit-level, AIGs present a convenient system model to study the problem of attacker model classification, because the range of actions that attackers have over coordinates is greatly restricted: either the attacker leaves the value of the coordinate as it is, or the attacker negates its current value. To be more precise, an attacker has one of three options when it comes to transforming a coordinate $\pi \in \Pi$ at state $\vec{x}$: 1) the attacker leaves $\vec{x}[\pi]$ as is, applying `id` in this coordinate, 2) the attacker forces $\vec{x}[\pi]$ to be 0, applying Δ_0 in coordinate c, or 3) the attacker applies Δ_1 in coordinate π, forcing $\vec{x}[x]$ to be 1. We can omit the transformation `id`, since it is always possible for the attacker to emulate `id` by applying $\Delta_{\vec{x}[c]}$ to coordinate c at state $\vec{x}$.

An Exponential Number of Attacks. For an AIG whose set of coordinates is Π, the set of attacker models is $\mathscr{P}(\Pi)$, which has size $2^{|\Pi|}$. Now, each attacker model $\langle\alpha\rangle$, with $\alpha \subseteq \Pi$, has $2^{|\alpha|}$ spatial transformations, since each coordinate in α is assigned either 0 or 1 by the effect of attacks in $\langle\alpha\rangle$. To compute for each attacker model $\langle\alpha\rangle$ the list of security requirements in R that $\langle\alpha\rangle$ breaks, a brute force algorithm would enumerate all attacker models by iterating α in $\mathscr{P}(\Pi)$, and for each attacker model $\langle\alpha\rangle$, it would try to check for every requirement $r \in R$ if there exists an attack $m \in \langle\alpha\rangle$ such that m causes R to break. This strategy is impractical; even if we only consider a single attacker model $\langle\alpha\rangle$, systematically verifying each attack $m \in \langle\alpha\rangle$ is an exponential problem. We need to adopt a different strategy if we are to produce meaningful results in a sensible amount of time. Instead of systematically generating attacks for a given attacker model $\langle\alpha\rangle$, we ask a SAT solver to give us an attack which fits the model $\langle\alpha\rangle$. More precisely, we use an SAT solver to obtain a set of assignments for the coordinates in α, which we can convert into an attack that fits the model $\langle\alpha\rangle$. We obtain the sequence of assignments by performing *bounded model checking* (BMC).

We observe that an attacker $\langle\alpha\rangle$ may only affect an isolated part of the system, so requirements that refer to other parts of the system should not be affected by $\langle\alpha\rangle$. We also observe that if some attacker $\langle\beta\rangle$ affects the system in a similar way to $\langle\alpha\rangle$ (e.g., if they control a similar set of coordinates), then the knowledge we obtain while verifying the system in against $\langle\alpha\rangle$ may be useful when verifying the system against $\langle\beta\rangle$. In this chapter, we present a bounded model checking strategy that uses these two observations, and provides a possible answer to Research Question 1.3 by illustrating how this model checking strategy efficiently maps each attacker to the set of requirements that they break.

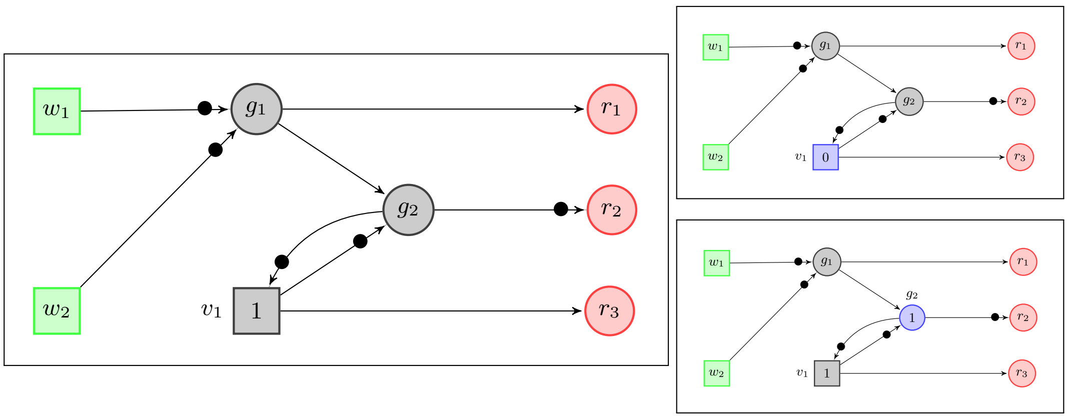

FIGURE 4.1: **Left:** Recall the AIG from Figure 2.2 describing a system with two inputs w_1 and w_2 (green boxes), one latch v_1 with initial value 1 (grey box), two gates g_1 and g_2 (gray circles), and three invariant requirements $r_1 = \Box g_1$, $r_2 = \Box\neg g_2$ and $r_3 = \Box v_1$ (red circles). The arrows represent logical dependencies, and bullets in the arrows imply negation. **Right above**: an attacker that controls latch v_1 can set its initial value to 0 to break r_2 and r_3 in 0 steps. This attacker uses a spatial transformation to implement this attack. **Right below**: an attacker that controls gate g_2 can set its value to 1 at time 0 to break r_2 in 0 steps and r_3 in 1 step, because the value of v_1 at time 1 is 0. This attacker uses a dynamics transformation, since gates are not part of the state of AIGs.

Summary of this Chapter. In this chapter we model AIGs as F-coalgebras, and we model their attackers as sets of spatial transformations. We use bounded model checking to classify and compare attackers, which provides an answer for Research Question 1.3 in the case of AIGs. Due to the exponential size of the problem, we propose a set of heuristics to perform bounded model efficiently, for which we provide experimental evidence.

The contents of this chapter are published in [RMSC20].

4.2 Motivational Example

Consider a scenario where an attacker A controls the gate g_2 of the AIG shown on the left of Figure 4.1. By controlling g_2, we mean that A can choose the value of $g_2(t)$ at will for $t \geq 0$.This system fails r_1 at time t if $w_1(t) = 1$, so the attacker A does not need to do anything to cause the system to fail r_1. Now, it is possible for A to break r_2 at time $t = 0$ and to break r_3 at time $t = 1$ by setting $g_2(0)$ to 1 via an attack, since $r_2 = \Box\neg g_2$, and by causing $v_1(1)$ to be equal to 0. This attack breaks both requirements r_2 and r_3, but in general an attacker may need one attack to break one requirement, and a different attack to break another. We say that A has the *power to break the requirements* r_1, r_2 and r_3, since there are attacks in the capabilities of A that break those requirements. Now, consider a different attacker B which only controls the gate g_1. No matter what value B chooses for $g_1(t)$ for all t, it is impossible for B to break r_2 or r_3, so we say that B only has the power to break r_1.

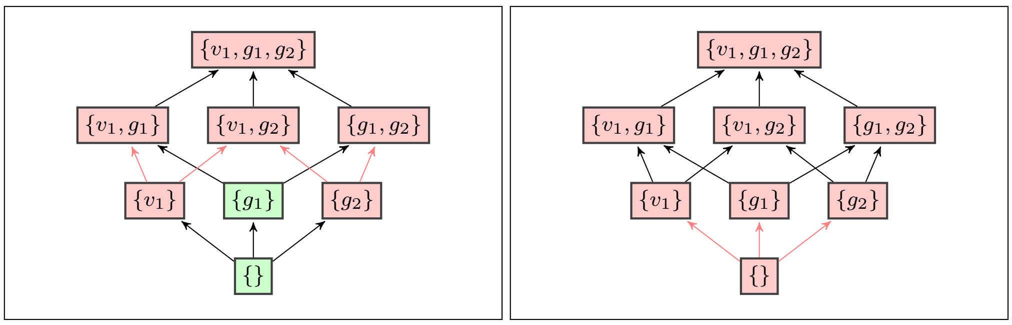

FIGURE 4.2: Left: classification of attackers for requirements r_2 and r_3. Right: classification of attackers for requirement r_1. A green attacker model cannot break the requirement, while a red attacker model can. The red arrows originate from minimal attacker models.

If we allow attackers to control any number of coordinates, then there are 8 different attackers, described by the subsets of $\{v_1, g_1, g_2\}$. We do not consider attackers that control inputs, because the model checking of invariant properties requires the property to hold for all inputs, so giving control of inputs to an attacker does not make it more powerful (i.e. the attacker cannot break more requirements than it already could without the inputs). Figure 4.2 illustrates the classification of attackers depending on whether they break a given requirement or not. Based on it, we can provide the following security guarantees: 1) the system cannot enforce r_1, and 2) that the system can only enforce r_2 and r_3 in the presence of attackers that are as capable to interact with the system as $\{g_1\}$ (i.e. they only control g_1 or nothing).

According to the classification, attacker $\{g_2\}$ is as powerful as the attacker $\{v_1, g_1, g_2\}$, since both attackers break the same requirements r_1, r_2 and r_3. This information may be useful to the designer of the system, because they may prioritise attackers that control less coordinates but are as powerful as attackers that control more when deploying defensive mechanisms.

4.3 AIGs as F-coalgebras

If we go by the strict definition of AIGs (see Section 2.6.2), then only latches are part of the states of AIGs; inputs and gates are external. We, however, include gates into the state. Consider an AIG whose set of latches is V, its set of gates is G, its set of inputs is W and its set of requirements is R. For each latch $v \in V$ there is an associated expression $v(t+1)$, for each gate $g \in G$, there is an associated expression $g(t)$, and each requirement $r \in R$ is of the form $r = \Box e$, where e is an expression. Let $\vec{G} = 2^G$, $\vec{V} = 2^V$, $\vec{W} = 2^W$ and let $\vec{R} = 2^R$; we plan to use the set $\vec{V} \times \vec{G}$ as the carrier so that a state $(\vec{v}, \vec{g})$ models the values for $\vec{v}(t+1)$ and $\vec{g}(t)$. We now define the functor $F(X) = \vec{R}^{\vec{W}} \times X^{\vec{W}}$, and we define the F-coalgebra $(\vec{V} \times \vec{G}, (\theta, \delta))$ where (θ, δ) is a pair function with $\theta \colon \vec{V} \times \vec{G} \to \vec{R}^{\vec{W}}$, and $\delta \colon \vec{V} \times \vec{G} \to (\vec{V} \times \vec{G})^{\vec{W}}$. Given an input $\vec{w} \in \vec{W}$, the function δ maps the state $(\vec{v}, \vec{g})$ to the state $(\vec{v}^{\vec{w}}, \vec{g}^{\vec{w}})$, which is defined for all $v' \in V$,

$g' \in G$ and $w' \in W$, by

$$\vec{v}^{\vec{w}}[v'] \triangleq [\![v'(t+1)]\!], \tag{4.1}$$

$$\vec{g}^{\vec{w}}[g'] \triangleq [\![g'(t)]\!], \tag{4.2}$$

where $[\![g(t)]\!]$ is the result of evaluating the expression $g'(t)$ by assuming $w'(t) = \vec{w}[w']$ and $v'(t) = \vec{v}[v']$. Note that if the expression $g'(t)$ depend on the value of $g''(t)$ where g'' is another gate, then we assume $g''(t) = \vec{g}^{\vec{w}}[g'']$ (AIGs are acyclic in their dependencies, only looping back to latches). Similarly, $[\![v'(t+1)]\!]$ is the result of evaluating the expression $v(t+1)$ by assuming $w'(t) = \vec{w}[w']$, $v'(t) = \vec{v}[v']$, and $g'(t) = \vec{g}^{\vec{w}}[g']$.

The function θ evaluates the requirements given an input $\vec{w}$; for $r \in R$, we define

$$\theta(\vec{v}, \vec{g})(\vec{w})[r] \triangleq [\![e(t)]\!], \quad \text{if } r = \Box e, \tag{4.3}$$

where $[\![e(t)]\!]$ is the result of evaluating the expression $e(t)$ by assuming $w'(t) = \vec{w}[w']$, $v'(t) = \vec{v}[v']$ and $g'(t) = \vec{g}^{\vec{w}}[g']$.

In the following we show how to use bounded model checking to find spatial transformations which reveal a latent coalgebra that fails one or more security requirements.

4.4 Bounded Model Checking of in the Presence of Attackers

In this section, we formalise the problem of attacker model classification for AIGs using bounded model checking in the presence of attackers. More precisely, we formalise attackers and their interactions with systems, and we show how to systematically generate bounded model checking problems that determine whether some given attacker breaks the security requirement being checked. We then propose two methods for the classification of attackers: 1) a brute-force method that creates a model checking problem for each attacker-requirement pair, and 2) a method that incrementally empowers attackers to find *minimal attackers*, i.e. attackers which represent large portions of the universe of attackers thanks to a monotonicity relation between the set of coordinates controlled by the attacker and the set of requirements that the attacker breaks.

4.4.1 Attackers and Compromised Systems

We modify the equations that are associated to the coordinates controlled the attacker to incorporate the possible actions of such attacker into an AIG. Formally, let $S = (W, V, G)$ be a system described by an AIG, let $R = \{r_1, \ldots, r_n\}$ be a set of invariant requirements for S, and let $C = V \cup G$ be the set of coordinates of S. An *attacker model* A is any subset of C. We refer to attackers fitting the model A as *A-attackers*. If a coordinate c belongs to an attacker model A, then an A-attacker has the *capability to interact with S through c*. To capture this interaction, we modify the equations of every latch $v \in V$ to be parametrised by an attacker model A as follows: the original transition equation $v(t+1) = e(t)$ and the initial equation $v(0) = b$ change to

$$v(t+1) = \begin{cases} e(t), & \text{if } v \notin A; \\ A_v(t+1), & \text{otherwise,} \end{cases} \quad v(0) = \begin{cases} b, & \text{if } v \notin A; \\ A_v(0), & \text{otherwise.} \end{cases} \tag{4.4}$$

where $A_v(t)$ is a value chosen by an A-attacker at time t. Similarly, we modify the equation of gate $g \in G$ as follows: the original equation $g(t) = e_1(t) \wedge e_2(t)$ changes to

$$g(t) = \begin{cases} e_1(t) \wedge e_2(t), & \text{if } g \notin A; \\ A_g(t), & \text{otherwise,} \end{cases} \tag{4.5}$$

where $A_g(t)$ is, again, a value chosen by an A-attacker at time t. We use $A[S]$ to denote the system S under the influence of an A-attacker; i.e., $A[S]$ is the modified system of equations.

We now apply the definition of attacks as state transformations. An *attack from an A-attacker*, is a function $m\colon \vec{G} \times \vec{V} \to \vec{G} \times \vec{V}$ that assigns a value to each coordinate in A. Given a state $\vec{x} \in \vec{G} \times \vec{V}$ and an attack $m\colon \vec{G} \times \vec{V} \to \vec{G} \times \vec{V}$, we define the *effect of m on coordinates A at state $\vec{x}$*, denoted $m(\vec{x})|_A\colon A \to 2$, by

$$m(\vec{x})|_A[a] \triangleq m(\vec{x})[a]. \tag{4.6}$$

In the following, we consider a strategy to build an attacks for an A-attacker by computing the effects on coordinates A in the context given by the initial state of the AIG and its successor states. Formally, we propose finding a finite sequence of vectors $\vec{a}_0, \ldots, \vec{a}_t$ such that the vectors fix the values of all $A_c(k)$ by assuming $A_c(k) = \vec{a}_k(c)$, with $c \in A$ and $0 \leq k \leq t$. In this sense, the sequence $(\vec{a}_0, \ldots, \vec{a}_t)$ models an *attack strategy to break requirements*.

Definition 4.4.1 (Broken Requirement)**.** Given a requirement $r \in R$ with $r = \Box e$, we say that an A-attacker *breaks the requirement r (at time t)* if and only if there exists a sequence of inputs $\vec{w}_0, \ldots, \vec{w}_t$ and an attack strategy $\vec{a}_0, \ldots, \vec{a}_t$ such that $e(t)$ is false if we assume $w'(k) = \vec{w}_k[w']$ and $a'(k) = \vec{a}_k[a']$, with $0 \leq k \leq t$, $w' \in W$, and $a' \in A$. We denote the set of requirements that an A-attacker breaks by $A[R]$.

We now define two partial orders for attackers: **1)** an A_i-attacker is strictly less *capable* (to interact with the system) than an A_j-attacker in the context of S iff $A_i \subseteq A_j$ and $A_i \neq A_j$. An A_i-attacker is equally capable to an A_j-attacker iff $A_i = A_j$; and **2)** an A_i-attacker is strictly less *powerful* than an A_j-attacker in the context of S and R iff $A_i[R] \subseteq A_j[R]$ and $A_i[R] \neq A_j[R]$. Similarly, A_i-attacker is equally powerful to an A_j-attacker iff $A_i[R] = A_j[R]$.

We can now properly present the problem of *attacker model classification*.

Definition 4.4.2 (Attacker Classification via Verification)**.** Given a system S, a set of requirements R, and a set of h attacker models $\{A_1, \ldots, A_h\}$, for every attacker model A, we compute the set $A[R]$ of requirements that an A-attacker breaks by performing model checking of each requirement in R on the compromised system $A[S]$.

Definition 4.4.2 assumes that exhaustive verification is possible for S and the compromised versions $A[S]$ for all attackers A. However, if exhaustive verification is not possible (e.g., due to time limitations or memory restrictions), we consider an alternative formulation using *bounded model checking* (BMC):

Definition 4.4.3 (Attacker Model Classification using BMC)**.** Let S be a system, R be a set of requirements, and t be a natural number. Given a set of attacker models $\{A_1, \ldots, A_h\}$, for each attacker model A , we compute the set $A[R]$ of requirements that an A-attacker breaks *using a strategy of length up to* t on the compromised system $A[S]$.

In the following, we show how to construct a SAT formula that describes the attacker model classification problem using BMC.

4.4.2 A SAT Formula for BMC up to t Steps

For a requirement $r = \Box e$ and a time step $t \geq 0$, we are interested in finding an attack strategy that causes the value of $e(k)$ to be false for some k with $0 \leq k \leq t$. To capture this notion, we define the proposition $\texttt{goal}(r, t)$ by

$$\texttt{goal}(\Box e, t) \triangleq \bigvee_{k=0}^{t} \neg e(k). \tag{4.7}$$

We inform the SAT solver of the equalities and dependencies between expressions given by the definition of the AIG (e.g., that $e(k) \Leftrightarrow \neg v_1(k)$). Inspired by the work of Biere *et al.* [Bie+99], we transform the equations into a Conjunctive Normal Form formula (CNF) that the SAT solver can work with using *Tseitin encoding* [Tse83]. Each equation of the form

$$v(0) = \begin{cases} b, & \text{if } v \notin A; \\ A_v(0), & \text{otherwise,} \end{cases}$$

becomes the formula of the form

$$\Big(v^{\downarrow} \vee (v(0) \Leftrightarrow b)\Big) \wedge \Big(\neg v^{\downarrow} \vee (v(0) \Leftrightarrow A_v(0))\Big), \tag{4.8}$$

where $v^{\downarrow}$ is a literal that marks whether the latch v is an element of the attacker model A currently being checked; i.e., we assume that $v^{\downarrow}$ is true if $v \in A$, and we assume that $v^{\downarrow}$ is false if $v \notin A$. Consequently, if $v \notin A$, then $v(0) \Leftrightarrow b$ must be true, and if $v \in A$, then $v(0) \Leftrightarrow A_v(0)$ must be true. We denote this proposition by $\texttt{encode}(v, 0)$, and it informs the SAT solver about the initial state of the AIG.

Similarly, for $0 \leq k < t$, each equation of the form

$$v(k+1) = \begin{cases} e(k), & \text{if } v \notin A; \\ A_v(k+1), & \text{otherwise,} \end{cases}$$

translates to a formula of the form

$$\left(v^{\downarrow} \vee (v(k+1) \Leftrightarrow e(k))\right) \wedge \left(\neg v^{\downarrow} \vee (v(k+1) \Leftrightarrow A_v(k+1))\right). \tag{4.9}$$

We call this formula $\texttt{encode}(v, k)$.

Finally, for $0 \leq k \leq t$, each equation of the form

$$g(k) = \begin{cases} e_1(k) \wedge e_2(k), & \text{if } g \notin A; \\ A_g(k), & \text{otherwise,} \end{cases}$$

becomes a formula of the form

$$\left(g^{\downarrow} \vee (g(k) \Leftrightarrow e_1(k) \wedge e_2(k))\right) \wedge \left(\neg g^{\downarrow} \vee (g(k) \Leftrightarrow A_g(k))\right), \tag{4.10}$$

where $g^{\downarrow}$ is a literal that marks whether the gate g is an element of the attacker model A currently being checked in a similar way that the literal $v^{\downarrow}$ works for the latch v. We refer to this formula by $\texttt{encode}(g, k)$.

Given an attacker model A, to perform SAT solving, we need to find an assignment of inputs in W and attacker actions for each coordinate c in A over t steps, i.e., an assignment of $|W \times A| \times t$ literals. The SAT problem for checking whether requirement r is safe up to t steps, denoted $\texttt{check}(r, t)$, is defined by

$$\texttt{check}(r,t) \triangleq \texttt{goal}(r,t) \wedge \bigwedge_{c \in (V \cup G)} \left(\bigwedge_{k=0}^{t} \texttt{encode}(c,k) \right). \tag{4.11}$$

If the SAT solver can provide an assignment of literals which satisfy $\texttt{check}(r, t)$, then an A-attacker breaks the requirement r.

Proposition 4.4.1. For a given attacker model A and a requirement $r = \Box e$, if we assume the literal $c^{\downarrow}$ for all $c \in A$ and we assume $\neg x^{\downarrow}$ for all $x \notin A$ (i.e., $x \in (V \cup G) - A$), then an A-attacker breaks the requirement r in t steps (or less) if and only if $\texttt{check}(r, t)$ is satisfiable.

Proof. We first show that if an A-attacker breaks the requirement in t steps or less, then $\texttt{check}(r, t)$ is satisfiable. Since an A-attacker breaks $r = \Box e$ in t steps or less, then, by Definition 4.4.1, there exists an assignment of inputs $(\vec{w}_0, \ldots, \vec{w}_k)$ and an attacker strategy $(\vec{a}_0, \ldots, \vec{a}_k)$ which causes $e(k)$ to be false for some $k \leq t$; this means that $\texttt{goal}(r, t)$ is satisfiable, which, in turn, makes $\texttt{check}(r, t)$ satisfiable.

We now show that if $\texttt{check}(r, t)$ is satisfiable, then an A-attacker breaks the requirement r in t or less steps. If $\texttt{check}(r, t)$ is satisfiable then $\texttt{goal}(r, t)$ is satisfiable, and $e(k)$ is false for some $k \leq t$. Consequently, there is an assignment of inputs $\vec{w}(k)$ and attacker actions $A_c(k)$, such that the $\texttt{encode}(c, k)$ formulae are satisfied for all $c \in A$. By assuming $\vec{a}_k(c) = A_c(k)$ and $\vec{w}_k = \vec{w}(k)$, we obtain a witness input sequence and a witness attack strategy which proves that an A-attacker breaks r in k steps (i.e., in t steps or less since $k \leq t$). □

```
Data: system $S = (W, V, G)$, a time step $t \geq 0$, a set of requirements $R$.
Result: A map that maps the attacker $A$ to $A[R]_t$.
1 Map $\mathcal{H}$;
2 foreach $r \in R$ do
3     foreach $A$ such that $A \subseteq (V \cup G)$ do
4         if check$(r, t)$ is satisfiable while assuming $c^{\downarrow}$ for all $c \in A$ then
5             insert $r$ in $\mathcal{H}(A)$;
6         end
7     end
8 end
9 return $\mathcal{H}$;
```

Algorithm 1: Naive attacker model classification algorithm.

Algorithm 1 describes a naive strategy to compute the sets $A[R]_t$ for each attacker model A; i.e. the set of requirements that an A-attacker breaks in t steps (or less). Algorithm 1 works by solving, for each of the $2^{|V \cup G|}$ different attacker models, a set of $|R|$ SAT problems, each of which has a size of at least $\mathcal{O}\left(|W \cup V \cup G| \times t\right)$ on the worst case.

We now propose two sound heuristics in an attempt to improve Algorithm 1: the first technique aims to reduce the size of the SAT formula, while the other heuristic aims to record and propagate the results of previous verifications among the set of attacker models so that some calls to the SAT solver are avoided.

4.4.3 Heuristics: Isolation and Monotonicity

Isolation is a strategy that uses data dependency analysis to prove that is impossible for a given attacker model to break some requirement. Informally, a requirement r is isolated from an attacker model A if no flow of information can occur from the coordinates in A to the expression that defines r. To formally capture isolation, we first extend the notion of IOC to attacker models. The *IOC of an attacker model* A, denoted $\blacktriangle(A)$, is defined by the union of IOCs of the coordinates in A; more precisely,

$$\blacktriangle(A) \triangleq \bigcup \{\blacktriangle(c) | c \in A\} . \tag{4.12}$$

Isolation happens whenever the IOC of A is disjoint from the COI of r, implying that A cannot interact with r.

Proposition 4.4.2 (Isolation)**.** Let A be an attacker model and r be a requirement that is satisfied by the AIG. If $\blacktriangle(A) \cap \blacktriangledown(r) = \emptyset$, then an A-attacker does not break r.

Proof. For an A-attacker to break the requirement r, there must be a coordinate $c \in \blacktriangledown(r)$ whose behaviour is affected by the presence of such A-attacker, causing r to fail. In such a case, there must be a dependency between the variables directly controlled by the A-attacker and c, since A-attackers only choose actions over the coordinates they control; implying that $c \in \blacktriangle(A)$. This contradicts the premise that the IOC of A and the COI of r are disjoint, so the coordinate c cannot exist. □ □

Isolation reduces the SAT formula by dismissing attacker models that are outside the COI of the requirement to be verified. Isolation works similarly to *COI reduction* (see [Cab+18; CCQ16; Bie+09; Cla+18; CN11]), and it transforms Equation 4.11 into

$$\texttt{check}(r,t) \triangleq \texttt{goal}(r,t) \wedge \bigwedge_{c\in(\blacktriangledown(r)-W)} \left(\bigwedge_{k=0}^{t} \texttt{encode}(c,k) \right) \tag{4.13}$$

Monotonicity is a strategy that uses the following relation between capabilities and power of attacker models: if an attacker model B has at least the same capabilities as an attacker model A (i.e. $A \subseteq B$), then anything that an A-attacker can do, a B-attacker can do as well.

Proposition 4.4.3 (Monotonicity)**.** For attacker models A and B and a set of requirements R, if $A \subseteq B$, then $A[R] \subseteq B[R]$.

Proof. If $A \subseteq B$, then a B-attacker can always choose the same attack strategies that an A-attacker uses to break the requirements in $A[R]$; however, there may be strategies that a B-attacker uses that an A-attacker cannot, e.g., if B has a coordinate that A does not. Consequently, $A[R] \subseteq B[R]$. □

We use monotonicity to define a notion of minimal (successful) attacker model for a requirement r: an attacker model A is a *minimal attacker model* for a requirement r if and only if an A-attacker breaks r, and there is no attacker model $B \subset A$ such that a B-attacker breaks r. In the following sections, we describe a methodology for attacker model classification that focuses on the identification of these minimal attacker models.

4.4.4 Minimal (Successful) Attacker Models

By monotonicity, any attacker model that is more capable than a minimal attacker models for a requirement r breaks r, and any attacker model that is less capable than a minimal attacker model for r cannot break r; otherwise, this less capable attacker model would be a minimal attacker model. In this sense, the minimal attacker models for r not only partition the set of attacker models into those that break r and those who do not, they create a frontier in the lattice of attacker models. For example, on the left of Figure 4.2, the minimal attacker models for requirement r_2 are $\{v_1\}$ and $\{g_2\}$; the frontier (shown by the red arrows) is formed because any attacker model that includes them also breaks r_2. For these reasons, we reduce the problem of attacker model classification to the problem of finding the minimal attacker models for all requirements.

Existence of a Minimal Attacker.

A requirement r that is isolated from an attacker model A remains safe in the presence of an A-attacker. Thus, for each requirement $r \in R$, if an attacker model A breaks r, then A must be a subset of $\blacktriangledown(r) - W$. Out of all the attacker models that could break r, the most capable attacker model is $\blacktriangledown(r) - W$. For succinctness, we henceforth denote the attacker $\blacktriangledown(r) - W$ by r^{max}. We check the satisfiability of $\texttt{check}(r,t)$ with respect to the attacker model r^{max} to learn whether there exists any other attacker model that breaks

r in t steps; if check(r, t) is not satisfied, then no attacker model breaks r, otherwise we contradict monotonicity.

Corollary 4.1. From monotonicity and isolation (cf. Propositions 4.4.3 and 4.4.2), if attacker r^{max} cannot break the requirement r, then there are no minimal attacker models for r.

```
Data: system $S$, a requirement $r$, and a time step $t \geq 0$.
Result: set $M$ of minimal attacker models for $r$, bounded by $t$.
1  if check$(r, t)$ is not satisfiable while assuming $c^{\downarrow}$ for all $c \in r^{max}$ then
2      return $\emptyset$;
3  end
4  Set: $P = \{\emptyset\}$, $M = \emptyset$;    //($P$ contains the empty attacker model $\emptyset$)
5  while $P$ is not empty do
6      extract $A$ from $P$ such that $A$ is minimal with respect to set inclusion;
7      if not (exists $B \in M$ such that $B \subseteq A$) then
8          if check$(r, t)$ is satisfiable when assuming $c^{\downarrow}$ for all $c \in A$ then
9              insert $A$ in $M$;
10         else
11             foreach $c \in (r^{max} - A)$ do
12                 insert $A \cup \{c\}$ in $P$;
13             end
14         end
15     end
16 end
17 return $M$;
```

Algorithm 2: The `MinimalAttackerModels` algorithm.

After having confirmed that at least one non-isolated attacker model breaks r, we focus on finding minimal attacker models. Our strategy consist of starting with the smallest possible sets of capabilities (i.e. non-isolated attacker models), and systematically increasing those sets of capabilities when they fail to break the requirement r; we continue increasing them until they do, or we find that they are a superset of a different minimal attacker model. Algorithm 2 describes this empowering procedure to computes the set of minimal attacker models for a requirement r, which we call `MinimalAttackerModels`. As mentioned, we first check to see if a minimal attacker model exists (Lines 1-3); then we start evaluating attacker models in an orderly fashion by always choosing the least capable attacker model in the set of pending attacker models P (Lines 5-16). Line 7 uses monotonicity to discard the attacker model A if there is a known successful attacker model B with $B \subseteq A$. Line 8 checks if the attacker model A breaks r in t steps (or less); if so, then A is a minimal attacker model for r and is included in M (Line 9); otherwise, we empower A with a new coordinate c, and we add these new attacker models to P (Lines 11-13). We note that Line 11 relies on isolation, since we only add coordinates that belong to the COI of r.

Example 4.1. We recall the motivational example from Section 4.2. Consider the computation of `MinimalAttackerModels` for requirement r_2. In this case, r_2^{max} is $\{g1, g2, v_1\}$, which breaks r_2. The model r_2^{max} confirms the existence of a minimal non-isolated attacker model (Lines 1-3). We start to look for minimal attacker models by checking the

```
Data: system $S$, a time step $t \geq 0$, and a set of requirements $R$.
Result: Set of all minimal attacker models $\mathcal{M}$ and an initial classification map $\mathcal{H}$.
1 Set: $\mathcal{M} = \emptyset$;
2 Map: $\mathcal{H}$;
3 for $r \in R$ do
4     for $A \in$ MinimalAttackerModels$(S, t, r)$ do
5         insert $r$ in $\mathcal{H}(A)$;
6         insert $A$ in $\mathcal{M}$;
7     end
8 end
9 return $(\mathcal{M}, \mathcal{H})$;
```

Algorithm 3: The `AllMinimalAttackerModels` algorithm.

attacker model $\emptyset$ (Lines 5-8); after we see that it fails to break r_2, we conclude that $\emptyset$ is not a minimal attacker model and that we need to increase its capabilities. We then derive the attacker models $\{g_1\}, \{g_2\}$ and $\{v_1\}$ by adding one non-isolated coordinate to $\emptyset$, and we put them into the set of pending attacker models (Lines 11-13). We know that the attacker models $\{v_1\}$ and $\{g_2\}$ break the requirement r_2, so they get added to the set of minimal attacker models, and they are not empowered (Line 9). Unlike $\{v_1\}$ and $\{g_2\}$, the attacker model $\{g_1\}$ fails to break r_2, so we increase its capabilities and we generate the attacker models $\{v_1, g_1\}$ and $\{g_1, g_2\}$. Finally, these two latter attacker models are supersets of minimal attacker models that we already identified, i.e., $\{v_1\}$ and $\{g_2\}$; this causes the check in Line 7 to fail, and they are dismissed from the set of pending attacker models as they are not minimal. The algorithm finishes with $M = \{\{v_1\}, \{g_2\}\}$.

Algorithm 3 applies Algorithm 2 to each requirement; it collects all minimal attacker models in the set $\mathcal{M}$ and initialises the attacker model classification map $\mathcal{H}$, which maps attacker models to the sets of requirements that they break. Finally, Algorithm 4 exploits monotonicity to compute the classification of each attacker model A by aggregating the requirements broken by the minimal attacker models that are subsets of A.

Example 4.2. For the motivational example in Section 4.2, Algorithm 3 returns $\mathcal{M} = \{\emptyset, \{v_1\}, \{g_2\}\}$ and $\mathcal{H} = \{(\emptyset, \{r_1\}), (\{v_1\}, \{r_2, r_3\}), (\{g_2\}, \{r_2, r_3\})\}$. From there, Algorithm 4 completes the map $\mathcal{H}$, and returns

$$\begin{aligned}\mathcal{H} = \{&(\emptyset, \{r_1\}), (\{v_1\}, \{r_1, r_2, r_3\}), (\{g_1\}, \{r_1\}), (\{g_2\}, \{r_1, r_2, r_3\}),\\ &(\{v_1, g_2\}, \{r_1, r_2, r_3\}), (\{v_1, g_2\}, \{r_1, r_2, r_3\}),\\ &(\{g_1, g_2\}, \{r_1, r_2, r_3\}), (\{v_1, g_1, g_2\}, \{r_1, r_2, r_3\})\},\end{aligned}$$

where each pair corresponds to $(A, A[R])$ with A being an attacker model and $A[R]$ being the set of requirements that A-attackers break.

4.4.5 On Soundness and Completeness

Just like any bounded model checking problem, if the time parameter t is below the completeness threshold (see [KS03]), the resulting attacker model classification up to t

```
Data: system $S = (W, V, G)$, a time step $t \geq 0$, a set of requirements $R$.
Result: A map $\mathcal{H}$ that maps the attacker model $A$ to $A[R]$.
$(\mathcal{M}, \mathcal{H})$ = AllMinimalAttackerModels$(S, t, R)$;
foreach $A \subseteq (V \cup G)$ do
    foreach $A' \in \mathcal{M}$ do
        if $A' \subseteq A$ then
            insert all elements of $\mathcal{H}(A')$ in $\mathcal{H}(A)$;
        end
    end
end
return $\mathcal{H}$;
```

Algorithm 4: Improved classification algorithm. We assume that $\mathcal{H}$ initially maps every attacker model A to the empty set.

steps could be *incomplete*. Informally, the completeness threshold is a time parameter which states that any bounded model checking beyond the completeness threshold never yields any new results. This means that an attacker model classification up to t steps may prove that an attacker model A cannot break some requirement r with a strategy up to t steps, while in reality an A-attacker breaks r by using a strategy whose length is strictly greater than t. Nevertheless, there are practical reasons for using a time parameter that is lower than the completeness threshold: 1) computing the exact completeness threshold is often as hard as solving the model-checking problem [Cla+18], and 2) the complexity of the classification problem growths exponentially with t because the size of the SAT formulae also grow with t; moreover, there is an exponential number of attacker models that need to be classified by making calls to the SAT solver. A classification method that uses a time bound t below the completeness threshold is nevertheless *sound*, i.e., the method does not falsely report that an attacker model breaks a requirement when in reality it cannot. In Section 4.6 we discuss possible alternatives to overcome incompleteness and provide results with more coverage.

Another source of incompleteness occurs when we artificially restrict minimal attacker models to a maximum size. This would be the case if we are interested in only verifying if there exists an attacker model that can break the requirements of the system by using only up to z coordinates. The results of a classification whose minimal attacker models are bounded is also sound but incomplete, since does not identify minimal attacker models that have more than z elements. We show in Section 4.5 that, even with restricted minimal attacker models, it is possible to obtain a high coverage of the universe of attacker models because we can extrapolate soundly those results using monotonicity.

4.4.6 Requirement Clustering

Property clustering [Cab+18; Bie+09; CN11] is a state-of-the-art technique for the model checking of multiple properties. Clustering allows the SAT solver to reuse information when solving a similar instance of the same problem, but under different assumptions. To create clusters for attacker model classification, we combine the SAT problems whose COI is similar (i.e., requirements that have a Jaccard index close to one), and incrementally enable and disable properties during verification. More precisely, to use clustering, instead of computing $\texttt{goal}(r,t)$ for a single requirement, we compute $\texttt{goal}(Y,t)$ for a cluster Y of requirements, defined by

$$\texttt{goal}(Y,t) \triangleq \bigwedge_{r\in Y} (\neg r^{\downarrow} \vee \texttt{goal}(r,t)). \tag{4.14}$$

where $r^{\downarrow}$ is a new literal that plays a similar role to the ones used for gates and latches; i.e., we assume $r^{\downarrow}$ when we want to find the minimal attacker models for r, and we assume $\neg y^{\downarrow}$ for all other requirements $y \in Y$.

The SAT problem for checking whether the cluster of requirements Y is safe up to t steps is

$$\texttt{check}(Y,t) \triangleq \texttt{goal}(Y,t) \wedge \bigwedge_{c\in(\blacktriangledown(Y)-W)} \left(\bigwedge_{k=0}^{t} \texttt{encode}(c,k) \right), \tag{4.15}$$

where $\blacktriangledown(Y) = \bigcup\{\blacktriangledown(r) | r \in Y\}$.

4.5 Evaluation

In this section, we perform experiments to evaluate the effectiveness of the isolation and monotonicity heuristics for the classification of attacker models, and we evaluate the completeness of partial classifications for different time steps. We use a sample of AIG benchmarks for evaluation; these benchmarks appeared in past Hardware Model-Checking Competitions (see [Hwma; Hwmb]) in the multiple-property verification track. Each benchmark has an associated list of invariants to be verified, which we interpret as the set of security requirements for the purposes of this evaluation. As of 2014, the benchmark set contained 230 different instances, coming from both academia and industrial settings [Cab+15]. We quote from [Cab+15]:

> "Among industrial entries, 145 instances belong to the SixthSense family (6s*, provided by IBM), 24 are Intel benchmarks (intel*), and 24 are Oski benchmarks. Among the academic related benchmarks, the set includes 13 instances provided by Robert (Bob) Brayton (bob*), 4 benchmarks coming from Politecnico di Torino (pdt*) and 15 Beem (beem*). Additionally, 5 more circuits, already present in previous competitions, complete the set."

We run experiments on a quad core MacBook with 2.9 GHz Intel Core i7 and 16GB RAM, and we use the SAT solver CaDiCaL version 1.0.3 [Cad]. The source code of the artefact is available at [Aig].

Evaluation Plan. We separate our evaluation in two parts: 1) a comparative study where we evaluate the effectiveness of using of monotonicity and isolation for attacker model classification in several benchmarks, and 2) a case study, where we apply our classification methodology to a single benchmark –`pdtvsarmultip`– and we study the results of varying the time parameters for partial classification.

4.5.1 Evaluating Monotonicity and Isolation

Given a set of competing classification methodologies $\mathcal{M}_1, \ldots, \mathcal{M}_n$ (e.g., Algorithm 1 and Algorithm 4), each methodology is given the same set of benchmarks $S_1, \ldots, S_m$, each with its respective set of requirements $r_1, \ldots, r_m$. To evaluate a methodology $\mathcal{M}$ on a benchmark $S = (W, V, G)$ with a set of requirements R, we allow $\mathcal{M}$ to "learn" for about 10 minutes per requirement by making calls to the SAT solver, and produce a (partial) attacker model classification $\mathcal{H}$. Afterwards, we compute the *coverage* of $\mathcal{M}$.

Definition 4.5.1 (Coverage)**.** Let $\mathscr{P}(C)$ be the set of all attacker models, and let $\mathcal{H}$ be the attacker model classification produced by the methodology $\mathcal{M}$. The relation $\mathcal{H}$ maps each attacker model A to a set of requirements that an A-attacker breaks, with $\mathcal{H}(A) \subseteq A[R]$, for each attacker A (the classification is sound and complete iff $\mathcal{H}(A) = A[R]$). The *attacker model coverage obtained by methodology $\mathcal{M}$ for a requirement r* is the percentage of attacker models $A \in \mathscr{P}(C)$ for which we can correctly determine whether A-attackers break r only by computing $r \in \mathcal{H}(A)$ (i.e.,do not allow guessing and we do not allow making new calls to the SAT solver).

We measure the execution time of the classification per requirement; i.e., the time it takes for the methodology to find minimal attacker models, capped at about 10 minutes per requirement. We force stop the classification for each requirement if a timeout occurs, but not while the SAT solver is running (i.e., we do not interrupt the SAT solver), which is why the reported time sometimes exceeds 10 minutes.

4.5.2 Effectiveness of Isolation and Monotonicity

To test the effectiveness of isolation and monotonicity, we select a small sample of seven benchmarks. For each benchmark, we test four variations of our methodology:

1. $(+IS, +MO)$: Algorithm 4, which uses both isolation and monotonicity
2. $(+IS, -MO)$: Algorithm 4 but removing the check for monotonicity on Algorithm 2, Line 7;
3. $(-IS, +MO)$: Algorithm 4 but using Equation 4.13 instead of Equation 4.15 to remove isolation while preserving monotonicity; and
4. $(+IS, +MO)$ Algorithm 1, which does not use isolation nor monotonicity.

Setup. The benchmarks we selected have an average of 173 inputs, 8306 gates, 517 latches, and 80 requirements. We arbitrarily define the time step parameter t to be ten. Now, following our formulation of attacker models, which uses $V \cup G$ as the set of coordinates, we observe that these benchmarks have on average 2^{8823} attacker models. However, since an attacker that controls a gate g can be emulated by an attacker that controls all latches in the sources of g, we choose to restrict attacker models to be comprised of only latches; i.e. the set of coordinates used by attacker models is now V. This restriction reduces the size of the set of attacker models from 2^{8823} to 2^{517} on average per benchmark.

We arbitrarily restrict the number of coordinates that minimal attacker models may control to a maximum of three. This restriction represents the arbitrary decision of accepting the risk of not defending against attackers controlling more than four gates (possibly due to the improbability of those attackers gaining control over those components in the first place). This decision implies that we would need to make at most $80 \times \sum_{k=0}^{3} \binom{517}{k}$ calls to the SAT solver per benchmark.

Results. Figure 4.3 illustrates the average coverage for the four different methodologies, for each of the seven benchmarks. The exact coverage values are reported in Tables 4.4 and 4.5. We see that the methodology which uses isolation and monotonicity consistently obtains the best coverage of all the other methodologies, with the exception of benchmark `6s155`, where the methodology that removes isolation performs better. We attribute this exception to the way the SAT solver reuses knowledge when working incrementally; it seems that, for $(-IS, +MO)$, the SAT solver can reuse more knowledge than for $(+IS, +MO)$, which is why $(-IS, +MO)$ can discover more minimal attacker models in average than $(+IS, +MO)$.

We observe that the most significant element in play to obtain a high coverage is the use of monotonicity. Methodologies that use monotonicity always obtain better results than their counterparts without monotonicity. Isolation does not show a trend for increasing coverage, but has an impact in terms of classification time. Figure 4.4 presents the average classification time per requirement for the benchmarks under the different methodologies. We note that removing isolation often increases the average classification time of classification methodologies; the only exception –benchmark `6s325`– reports a smaller time because the SAT solver ran out of memory during SAT solving about 50% of the time, which caused an early termination of the classification procedure. This early termination also reflects on the comparatively low coverage for the method $(-IS, +MO)$ in this benchmark, reported on Figure 4.3.

4.5.3 Partial Classification of the `pdtvsarmultip` Benchmark

The benchmark `pdtvsarmultip` has 17 inputs, 130 latches, 2743 gates, and has an associated list of 33 invariant properties, out of which 31 are different (three requirements are equal). We interpret those 31 invariants as the list of security requirements. Since we are only considering attacker models that control latches, there are a total of 2^{130} attacker models that need to be classified for the 31 security requirements.

We consider six scenarios for the partial classification up to t steps. We allow t to take values in the set $\{0, 1, 5, 10, 20, 30\}$. We obtain the execution time of classification (ms), the size of the set of source latches for the requirement (#C), the number of minimal attacker models found (#Min.), the total number of calls to the SAT solver (#SAT), the average number of coordinates per minimal attacker (#C./Min) and the coverage for the requirement (Cov.) for each requirement. We present the average of these measures in Table 4.1.

Normally, the attacker model classification behaves in a similar way to what is reported for requirement $\square\neg g_{2177}$, shown in Table 4.2. More precisely, coverage steadily increases and stabilises as we increase t. However, we like to highlight two interesting phenomena that may occur: 1) coverage may *decrease* as we increase the time step (e.g.,

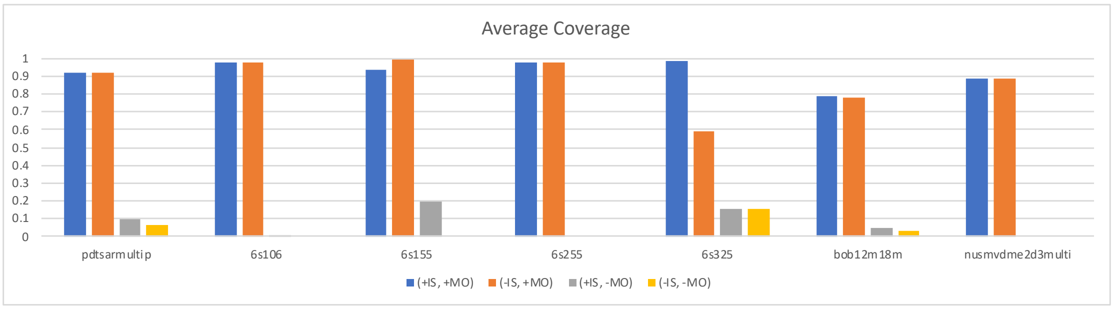

FIGURE 4.3: Average requirement coverage per benchmark. A missing bar indicates a value that is approximately 0.

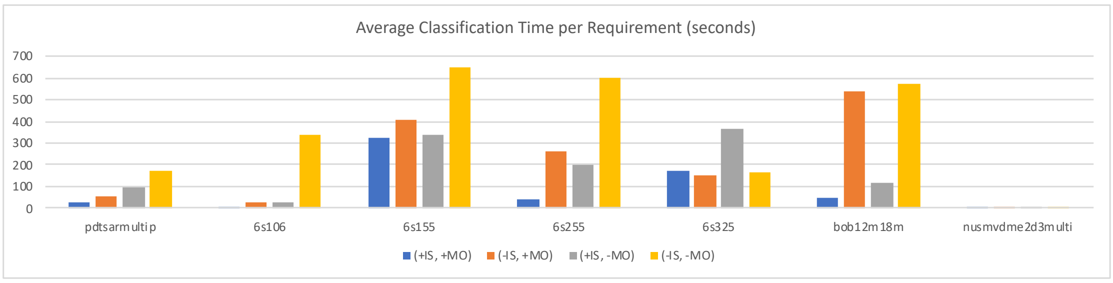

FIGURE 4.4: Average classification time per requirement per benchmark. A missing bar indicates a value that is approximately 0.

as shown in Table 4.3), and 2) the number of minimal attacker models decreases while the coverage increases, as in Tables 4.2 and 4.3.

The coverage may decrease as we increase the time step occurs because the set of attacker models whose actions affect requirements at time 0 is rather small, i.e., 2^6, because the effect of actions by other attackers do not have time to propagate. When we consider one step of propagation, the set of attacker models whose actions affect the requirements at times 0 and 1 has size 2^{26}. The size of this set increases with time until it stabilises at 2^{66}, which is the size of the set of attackers that cannot be dismissed by isolation.

The number of minimal attacker models may decrease as the coverage increases because the minimal attacker models that are found for smaller time steps may require more components on average to break requirements than the minimal attacker models that are found when we give more time for effect of attacks to propagate. More precisely, those minimal attacker models control a relatively large set of coordinates, which they need to be successful in breaking requirements, as shown in Step 5, column #C./Min in Tables 4.2 and 4.3. By considering more time steps, we are allowing attackers that control less coordinates to further propagate their actions through the system, which enables attack strategies that were unsuccessful for smaller choices of time steps.

By taking an average over all requirements, we observe that coverage seems to steadily increase as we increase the number of steps for the classification, as reported in Table 4.1, column Cov. The low coverage for small t is due to the restriction on the size of

TABLE 4.1: Average measures for all requirements per time steps.

Steps	ms	#C.	#Min.	#SAT	#C./Min.	Cov.
0	683.12903	34.967741	2.4516129	16328.774	1.4	0.5272775
1	2387.5483	46.225806	6.3870967	24420	1.6502324	0.5725332
5	5229.9354	58.935483	44.903225	28355.290	1.6396456	0.849569
10	24967.129	58.935483	151.19354	25566.548	1.4608692	0.9189732
20	13632.516	58.935483	17.677419	20849.709	1.1762725	0.9793542
30	12208.258	58.935483	15.935483	20798.161	1.1045638	0.9793542

TABLE 4.2: Coverage for $\Box\neg g_{2177}$.

$\Box\neg g_{2177}$						
Steps	ms	#C	#Min.	#SAT	#C./Min.	Cov.
0	895	59	0	34281	–	5.94E-14
1	2187	66	10	47378	2	0.499511
5	1735	66	205	12476	1.912195	0.999997
10	968	66	27	9948	1	0.999999
20	1275	66	27	9948	1	0.999999
30	1819	66	27	9948	1	0.999999

TABLE 4.3: Coverage for $\Box\neg g_{2220}$.

$\Box\neg g_{2220}$						
Steps	ms	#C.	#Min.	#SAT	#C./Min.	Cov.
0	1	6	1	28	1	0.90625
1	86	26	1	2628	1	0.500039
5	4511	67	17	47664	2.588235	0.852539
10	3226	67	6	37889	1	0.984375
20	3355	67	6	37889	1	0.984375
30	3562	67	6	37889	1	0.984375

minimal attacker models; if we allowed more coordinates for minimal attackers, then the effect of their attacks may propagate faster to the target requirements. More precisely, for small t, attackers can only use short strategies, which limits their interaction with the system. In this sense, we expect minimal attacker models to control a large number of coordinates if they want to successfully influence a requirement in a single time step, and since we restricted our search to attacker models of maximum size three, those larger attacker models remain unexplored (e.g., as reported in Table 4.2 for Step 0).

We conclude that experimental evidence favours the use of both monotonicity and isolation for the classification of attacker models, although some exceptions may occur for the use of isolation. In general, monotonicity and isolation help the classification methodology $(+IS, +MO)$ consistently obtain significantly better coverage compared to the naive methodology $(-IS, -MO)$.

4.6 Related Work

On Defining Attackers. Describing an adequate attacker model to contextualise the security guarantees of a system is not a trivial task. Some attacker models may be adequate to provide guarantees over one property, but not for a different one (e.g., the Dolev-Yao attacker model is sensible for confidentiality properties, but not so much for integrity properties). Additionally, depending on the nature of the system and the security properties being studied, it is sensible to describe attackers at different levels of abstraction. For instance, in the case of security protocols, Basin and Cremers define attackers in [BC14] as combinations of compromise rules that span over three dimensions: *whose* data is compromised, *which* kind of data it is, and *when* the compromise occurs. In the case of Cyber-physical Systems (CPS), works like [Gir+18] model attackers as sets of components (e.g., some sensors or actuators), while other works like [Wee+16; C+11; Urb+16] model attackers that can arbitrarily manipulate any control inputs and any sensor measurements at will, as long as they avoid detection. In the same context of CPS, Rocchetto and Tippenhauer [RT16] model attackers more abstractly as combinations of quantifiable traits (e.g., insider knowledge, access to tools, and financial support); these traits, when provided a compatible system model, ideally fully define how the attacker can interact with the system.

Our methodology for the definition of attackers combines aspects from [BC14] and [Gir+18]. The authors of [BC14] define symbolic attackers and a set of rules that describe how the attackers affect the system, which is sensible since many cryptographic protocols are described symbolically. Our methodology describes attackers as sets of coordinates (staying closer to the definitions of attackers in [Gir+18]), and has a lower level of abstraction since we describe the semantics of attacker actions in terms of how they change the functional behaviour of the AIG, and not in terms of what they ultimately represent. This lower level of abstraction lets us systematically and exhaustively generate attacker models for a given system by using the definition of such system, but those models do not automatically translate to other systems. Basin and Cremers compare the effect of attackers across different protocol implementations because their attacker models have the same abstract semantics. If we had an abstraction function from sets of gates and latches to abstract effects (e.g., "gates in charge of encryption", or "latches in charge of redundancy"), then it could be possible to compare results amongst different AIGs.

In Chapter **??**, we expand the notion of attacker models that we used in this chapter, i.e., attacker models as sets of coordinates, by adding a new component: *attack preconditions*. If coordinates determine *which* components are compromised and *how*, then attack preconditions define *when* they are compromised.

On Efficient Classification. The works by Cabodi, Camurati and Quer [CCQ16], Cabodi et. al [Cab+18], and Cabodi and Nocco [CN11] present several useful techniques that may improve the performance of model checking when verifying multiple properties, including COI reduction and property clustering. We also mention the work by Goldberg et al. [Gol+18] where they consider the problem of efficiently checking a set of safety properties P_1 to P_k by individually checking each property while assuming that all other properties are valid. These works inspired us to incrementally check requirements in the same cluster, helping us transform Equation 4.11 into Equation 4.15. Nevertheless, we note that all these techniques are described for model checking systems in the absence of attackers, which is why we needed to introduce the notions of isolation and monotonicity to account for them. We believe that it may be possible to use or incorporate other techniques that improve the efficiency of BMC in general (e.g., interpolation [McM03]), but such optimisations are beyond the scope of this work.

On Completeness. As mentioned in Section 4.4.5, if the time parameter for the classification is below the completeness threshold, the resulting attacker model classification is most likely incomplete. To guarantee completeness, it may be possible to adapt existing termination methods (e.g. [AS04]) to incorporate attacker models. Alternatively, methods that compute a good approximation of the completeness threshold (e.g. [KS03]) should not only improve coverage in general but also the reliability of the resulting attacker model classifications. Interpolation [McM03] could also help finding a guarantee of completeness.

Finally, if we consider alternative verification techniques to BMC, then IC3 [Bra11; Bra12] and PDR [Bra11], which have seen some success in hardware model checking, may address the limitation of boundedness.

On Verifying Non-Safety Properties. In this work, we focused our analysis exclusively on safety properties of the form $\Box e$. However, it is possible to extend this methodology to other types of properties, as it is possible to efficiently encode Linear Temporal Logic formulae for bounded model checking [Bie+99; Bie+06]. The formulations of the SAT problem change for the different formulae, but both isolation and monotonicity should remain valid heuristics; ultimately, monotonicity and isolation are about the sound extrapolation of the validity of attack strategies among attacker models that have related capabilities.

4.7 Conclusion

It surprised us that finding minimal attacker models that had only one coordinate was relatively simple. Informally, this means that the systems we studied were not robust: all it takes is for one single component to fail or be attacked, and the systems start to fail their security requirements. Since altering a single component already causes systems to fail, we conclude that using SAT solvers and BMC might have been a bit excessive for the purposes of attacker classification; it could have sufficed to systematically test the systems instead of checking them. In Chapter **??**, we switch from a verification to a testing methodology.

Benchmark	pdtsarmultip	6s106	6s155	6s255	6s325	bob12m18m	nusmvdme2d3multi
Average Coverage per Requirement							
(+IS, +MO)	0.918973269	0.977620187	0.93762207	0.977172852	0.986949581	0.785075777	0.8853302
(-IS, +MO)	0.916622423	0.977008259	0.99609375	0.9765625	0.590205544	0.781704492	0.8853302
(+IS, -MO)	9.95E-02	2.01E-03	0.1953125	1.73828E-11	0.156147674	4.61E-02	4.52E-15
(-IS, -MO)	6.45E-02	9.10E-36	6.00E-72	2.0838E-222	0.156146179	3.29E-02	4.52E-15
Average Classification Time per Requirement							
(+IS, +MO)	24967.12903	3088.882353	326372.2188	38396.125	174066.6047	49916.59211	3530.666667
(-IS, +MO)	56187.48387	25090.47059	408165.4063	263457.125	152425.3889	541870.25	3348.333333
(+IS, -MO)	97326.29032	28483.11765	340715.0313	197275.75	363913.8472	114330.0855	3890
(-IS, -MO)	170630.7742	341430.7059	646926.0625	602012.5	162959.5436	572080.0987	4262.333333
Average Number of Components to Build Attackers From per Requirement							
(+IS, +MO)	58.93548387	35.47058824	9	93.875	173.1196013	124.0460526	63
(-IS, +MO)	82	135	257	752	1668	261	63
(+IS, -MO)	58.93548387	35.47058824	9	93.875	173.1196013	124.0460526	63
(-IS, -MO)	82	135	257	752	1668	261	63

TABLE 4.4: Average metrics per requirement for the different benchmarks (Continued in Figure 4.5)

Benchmark	pdtsarmultip	6s106	6s155	6s255	6s325	bob12m18m	nusmvdme2d3multi
Average Number of Identified Successful Attackers per Requirement							
(+IS, +MO)	151.1935484	16.05882353	7.5	2.5625	237.3056478	34.93421053	129.3333333
(-IS, +MO)	146.0322581	16.05882353	8	2.4375	46.42424242	34.68421053	129.3333333
(+IS, -MO)	15860.25806	14693.17647	98	13792.9375	214293.01	22700.58553	1282
(-IS, -MO)	27207.54839	105842.7059	126133.2813	20657	2502.25641	40526.72368	1282
Average Number of Calls to SAT Solver per Requirement							
(+IS, +MO)	25566.54839	1978.647059	10.4375	11382.125	583123.7209	300825.9605	40576.33333
(-IS, +MO)	58420.83871	294477.9412	2022640.25	35252.1875	51411.24747	1607803.138	40576.33333
(+IS, -MO)	41275.6129	16655.76471	101	23455.875	550652.3721	322031.1513	41729
(-IS, -MO)	85390.80645	396510	1389339.563	49366.1875	56238.01026	1266308.526	41729
Average Minimal Number of Components Needed by Successful Attackers per Requirement							
(+IS, +MO)	1.460869285	1.004524887	1	0.464285714	1.649279888	1.142237928	2.843533741
(-IS, +MO)	1.460869285	1.004524887	1	0.4375	1	1.123402209	2.843533741
(+IS, -MO)	2.912532493	2.909934933	2.1953125	2.885495873	2.927835684	2.964760733	2.984195449
(-IS, -MO)	2.975171828	2.983304749	2.665659086	1.986925862	1.977314997	2.973301799	2.984195449

TABLE 4.5: Average metrics per requirement for the different benchmarks (Cont. of Figure 4.4)

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

[Aig] *And-Inverter Graph Attacker Classification*. URL: https://gitlab.com/asset-sutd/public/aig-ac.

[AS04] Mohammad Awedh and Fabio Somenzi. "Proving More Properties with Bounded Model Checking". In: *Computer Aided Verification*. Ed. by Rajeev Alur and Doron A. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 96–108. ISBN: 978-3-540-27813-9.

[BC14] David Basin and Cas Cremers. "Know Your Enemy: Compromising Adversaries in Protocol Analysis". In: *ACM Trans. Inf. Syst. Secur.* 17.2 (Nov. 2014), 7:1–7:31. ISSN: 1094-9224. DOI: 10.1145/2658996. URL: http://doi.acm.org/10.1145/2658996.

[BHW11] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond*. Tech. rep. FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[Bie+06] Armin Biere et al. "Linear Encodings of Bounded LTL Model Checking". In: *Logical Methods in Computer Science* Volume 2, Issue 5 (2006).

[Bie+09] A. Biere et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009. ISBN: 1586039296, 9781586039295.

[Bie+99] Armin Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-49059-3.

[Bra11] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18275-4.

[Bra12] Aaron R. Bradley. "Understanding IC3". In: *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*. SAT'12. Trento, Italy: Springer-Verlag, 2012, pp. 1–14. ISBN: 978-3-642-31611-1. DOI: 10.1007/978-3-642-31612-8_1. URL: http://dx.doi.org/10.1007/978-3-642-31612-8_1.

[C+11] Alvaro A. Cárdenas et al. "Attacks Against Process Control Systems: Risk Assessment, Detection, and Response". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. Hong Kong, China: ACM, 2011, pp. 355–366. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966959. URL: http://doi.acm.org/10.1145/1966913.1966959.

[Cab+15] Gianpiero Cabodi et al. "Hardware Model Checking Competition 2014: An Analysis and Comparison of Model Checkers and Benchmarks". In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2015), pp. 135–172.

[Cab+18] G. Cabodi et al. "To split or to group: from divide-and-conquer to sub-task sharing for verifying multiple properties in model checking". In: *International Journal on Software Tools for Technology Transfer* 20.3 (2018), pp. 313–325. ISSN: 1433-2787. DOI: 10.1007/s10009-017-0451-8. URL: https://doi.org/10.1007/s10009-017-0451-8.

[Cad] 2019. URL: http://fmv.jku.at/cadical/.

[CCQ16] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. "A graph-labeling approach for efficient cone-of-influence computation in model-checking problems with multiple properties". In: *Software: Practice and Experience* 46.4 (2016), pp. 493–511. DOI: 10.1002/spe.2321. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2321. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2321.

[Cla+18] Edmund M. Clarke et al., eds. *Handbook of Model Checking*. Springer, 2018.

[CN11] G. Cabodi and S. Nocco. "Optimized model checking of multiple properties". In: *2011 Design, Automation Test in Europe*. 2011, pp. 1–4. DOI: 10.1109/DATE.2011.5763279.

[Gir+18] Jairo Giraldo et al. "A Survey of Physics-Based Attack Detection in Cyber-Physical Systems". In: *ACM Comput. Surv.* 51.4 (July 2018), 76:1–76:36. ISSN: 0360-0300. DOI: 10.1145/3203245. URL: http://doi.acm.org/10.1145/3203245.

[Gol+18] E. Goldberg et al. "Efficient verification of multi-property designs (The benefit of wrong assumptions)". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 43–48. DOI: 10.23919/DATE.2018.8341977.

[Gün90] Christoph G. Günther. "An Identity-Based Key-Exchange Protocol". In: *Advances in Cryptology — EUROCRYPT '89*. Ed. by Jean-Jacques Quisquater and Joos Vandewalle. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 29–37. ISBN: 978-3-540-46885-1.

[Hel63] L. Hellerman. "A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits". In: *IEEE Transactions on Electronic Computers* EC-12.3 (1963), pp. 198–223. DOI: 10.1109/PGEC.1963.263531.

[Hwma] 2011. URL: http://fmv.jku.at/hwmcc11/index.html.

[Hwmb] 2013. URL: http://fmv.jku.at/hwmcc13/index.html.

[KS03] Daniel Kroening and Ofer Strichman. "Efficient Computation of Recurrence Diameters". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Lenore D. Zuck et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 298–309. ISBN: 978-3-540-36384-2.

[Kue+02] A. Kuehlmann et al. "Robust Boolean reasoning for equivalence checking and functional property verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.12 (2002), pp. 1377–1394. DOI: 10.1109/TCAD.2002.804386.

[McM03] K. L. McMillan. “Interpolation and SAT-Based Model Checking”. In: *Computer Aided Verification*. Ed. by Warren A. Hunt and Fabio Somenzi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–13. ISBN: 978-3-540-45069-6.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0849385237.

[RMSC20] Eric Rothstein-Morris, Jun Sun, and Sudipta Chattopadhyay. “Systematic Classification of Attackers via Bounded Model Checking”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Dirk Beyer and Damien Zufferey. Cham: Springer International Publishing, 2020, pp. 226–247. ISBN: 978-3-030-39322-9.

[RT16] Marco Rocchetto and Nils Ole Tippenhauer. “On Attacker Models and Profiles for Cyber-Physical Systems”. In: *Computer Security – ESORICS 2016*. Ed. by Ioannis Askoxylakis et al. Cham: Springer International Publishing, 2016, pp. 427–449. ISBN: 978-3-319-45741-3.

[Sha07] Hovav Shacham. “The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. URL: https://doi.org/10.1145/1315245.1315313.

[Tse83] G. S. Tseitin. “On the Complexity of Derivation in Propositional Calculus”. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28. URL: https://doi.org/10.1007/978-3-642-81955-1_28.

[Urb+16] David I. Urbina et al. “Limiting the Impact of Stealthy Attacks on Industrial Control Systems”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 1092–1105. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978388. URL: http://doi.acm.org/10.1145/2976749.2978388.

[Wee+16] S. Weerakkody et al. “Information flow for security in control systems”. In: *2016 IEEE 55th Conference on Decision and Control (CDC)*. 2016, pp. 5065–5072. DOI: 10.1109/CDC.2016.7799044.