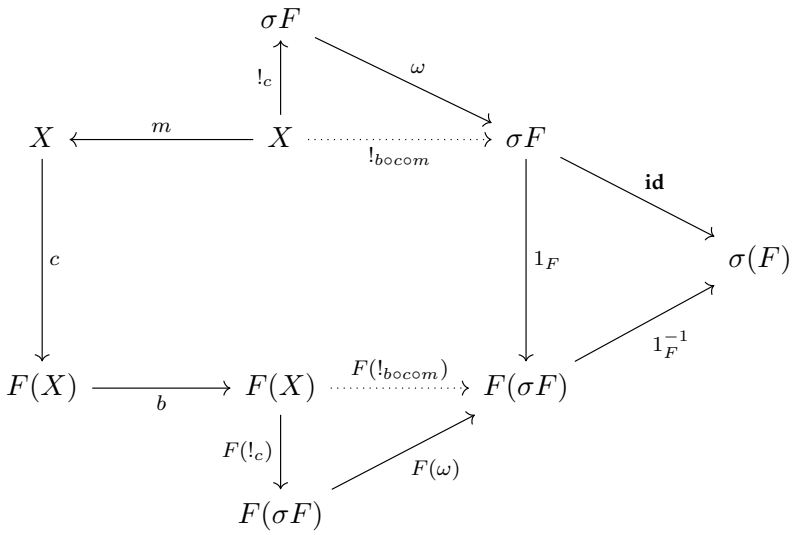
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN



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 (LBA) is a method to study changes in the observable behaviour of a transition system when in the presence of an external force, e.g., an attack or a fault. In LBA, we model these external forces using spatial transformations, while preserving the original transition dynamics of the system. We use the framework of universal coalgebra to model the transition system as an F-coalgebra $(X, c\colon X \to F(X))$ for some functor F, and the external force as a spatial transformation $m\colon X \to X$, which lets us seamlessly combine the two into the latent F-coalgebra $(X, c \circ m)$. The latent F-coalgebra $(X, c \circ m)$ implements the transition system in the presence of the force modelled by m. Latent coalgebras are compatible with the usual verification and testing methods for coalgebras and transition systems, and the properties satisfied by latent coalgebras prove properties about properties about the original system: if the latent coalgebra $(X, c \circ m)$ satisfies some property P, then the original system (X, c) satisfies P in the presence of the force modelled by m.

Although using spatial transformations to reveal latent coalgebras is a simple idea, it has a variety of practical applications. In this thesis, we show how to use LBA to approach three cybersecurity problems. The first problem is the *classification of attacker models*; we provide a method to systematically generate and compare attacker models for a system based on their potentially harmful effects by modelling their attacks using spatial transformations. The second problem is the *quantification of robustness in Cyber-physical Systems (CPSs)*; we propose a methodology for the systematic generation of attacks in CPSs to quantify the robustness of a system, and we propose a notion of latent robustness which considers the use of counterattacks (also modelled using spatial transformations). Finally, we study the problem of *timing side-channel repair*, for which we propose using a spatial transformation to enforce constant memory access patterns and preserves the functionality of the program.

Publications

Rothstein-Morris E., Murguia C. G. , and Ochoa M. (2017). Design-time Quantification of Integrity in Cyber-physical Systems. In Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security. PLAS '17. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/3139337.3139347

Rothstein-Morris E., Sun J. and Chattopadhyay S. (2020) Systematic Classification of Attackers via Bounded Model Checking. In: Beyer D., Zufferey D. (eds) Verification, Model Checking, and Abstract Interpretation. VMCAI 2020. Lecture Notes in Computer Science, vol 11990. Springer, Cham. https://doi.org/10.1007/978-3-030-39322-9_11

Rothstein-Morris E., Sun J., Chattopadhyay S. (2021) ORIGAMI: Folding Data Structures to Reduce Timing Side-Channel Leakage. Submitted to the 35th IEEE Computer Security Foundations Symposium. CSF 2022.

Rothstein-Morris E., Castellanos J.H., Ochoa M. and Chattopadhyay S. (2021). Improving the Robustness of Cyber-Physical System Models via Latent Behaviour Analysis and Testing. Submitted to the ACM Transactions on Privacy and Security.

Acknowledgements

I'm still writing this :-)

I will list all of them, indeed. They are important to me.

Thank you Maria Fernanda García, dear Mafe, for always checking on me! Your messages always bring joy and a smile to my face, and they keep me motivated. I'm very grateful to be your friend :)

Contents

List of Figures

List of Tables

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

Chapter 1

Introduction

1.1 Motivation: Classifying Attackers

To protect our systems, we need to understand what attackers can do to them. Basin and Cremers [**KnowYourEnemy**] quantify the power attackers in security protocols by defining a security protocol hierarchy with respect to a fixed security property and a set of attacker models. The position of a protocol in the hierarchy depends on its level of assurance with respect to the confidentiality property that the attackers aim to break: the protocols higher in the hierarchy can tolerate the presence of more powerful attackers (i.e., attackers with more capabilities). More precisely, the protocol hierarchy with respect to a property P is a directed acyclic graph whose nodes are pairs of a protocol Π and set of attacker capabilities K; there is an edge from a node (Π_1, K_1) to a node (Π_2, K_2) if Π_1 satisfies the property P in the presence of an attacker with capabilities K_1, but adding additional capabilities to the attacker would break the property P in the protocol Π_1, whilst the property P is still satisfied in the protocol Π_2 despite the addition of those capabilities.

To check if a protocol satisfies a confidentiality property in the presence of adversaries, Basin and Cremers model the protocol as symbolic transition system, and they model an adversary as a set of compromise rules that enable previously inexistent transitions in the protocol. These new transitions expand the set of reachable states, which now may contain a state where the knowledge of the attacker has enough sensitive information to break the property being verified. The idea of an attacker adding new transitions to an existing system is simple yet powerful, because it allows us to study exceptional behaviours of the system. However, to which extend can an attacker add new transitions to a system? Can they add arbitrary transitions? Can they also remove existing transitions? All these are pertinent questions, and their answers have interesting implications. Basin and Cremers do not allow attackers to create arbitrary transitions; instead, attackers can only create these transitions based on the compromise rules that are available to them. These compromise rules span over the following three aspects: *which* kind of data is compromised, *whose* data is it, and *when* the compromise occurs.

The attackers described by Basin and Cremers use symbolic compromise rules that abstract the concrete implementation of attacks and focus exclusively on their effects. More precisely, a transition generated by a compromise rule changes the state of the knowledge of the attacker (e.g. the attacker now knows the long term keys). Since these transitions are symbolic, we do not know how the attacker may concretely obtain such

data; we simply assume that they can and that they do. While arbitrary transitions are theoretically possible, they may be practically infeasible since it is impossible to create secure systems against arbitrary state changes. For example, an attacker could, in theory, create a transition from a state where they have no knowledge about any secrets to a state where they have knowledge of all of the secrets. Such attacker could clearly break any confidentiality protocol, but the realism of such attacker model is questionable.

1.2 Beyond Confidentiality and Symbolic Attackers

The work by Basin and Cremers verifies protocols with respect to confidentiality properties, but more precisely with respect to *safety* properties, since it uses a reachability approach. It is possible to study similar safety properties using their method, even if the properties address domains beyond confidentiality, i.e., integrity or availability. We are particularly interested in studying the *robustness* of systems, a concept which is closer to integrity than it is to confidentiality.

Informally, a *robust system* preserves its integrity in the presence of adversaries: despite the effect of attacks, the system does not reach any critical states. We are interested both in quantifying how robust a system is and also in the concrete strategies that attackers use when attacking the system. Since Basin and Cremers describe attackers with symbolic capabilities, the details of how information is revealed to attackers are abstracted away. To learn how attackers may compromise robustness, we are going to lower the level of abstraction of attackers: we define attackers as sets of concrete actions (e.g., the attacker replaces the value of a variable x by its complement) instead of a set of symbolic transition rules (e.g., the attacker learns the short-term secret keys). We formalise the actions of the attacker using spatial transformations. A *spatial transformation* is an endofunction in the set of states of a transition system; i.e. it is a function that maps states to states.

1.3 Idea: Attackers as Sets of Spatial Transformations

We now illustrate the effects of a spatial transformation in the behaviour of a transition system: consider the set of integer number pairs $\mathbb{Z} \times \mathbb{Z}$ and the transition function $\delta\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ that maps $(x, y) \in \mathbb{Z}$ to (x^2, y). The pair $(\mathbb{Z} \times \mathbb{Z}, \delta)$ defines a transition system, which we represent in Figure **??**.

The behaviour of $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ is the sequence that results from the iterated application of δ starting from (x, y); i.e., $(x^2, y), (x^4, y), (x^8, y)$, etc. Now, consider the endofunction $m\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ that maps $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ to $(-y, x)$. The transformation m also generates a transition system, shown in Figure **??**. The transitions δ and m do not commute in this scenario: applying δ first and then m maps (x, y) to $(-y, x^2)$ (whose transition system we show in Figure **??**) while applying m first and then δ maps (x, y) to (y^2, x) (whose transition system we show in Figure **??**).

Since $(\mathbb{Z} \times \mathbb{Z}, \delta)$ is the current transition system, we interpret m as a spatial transformation whose application reveals the system shown in Figure **??**. The behaviour of

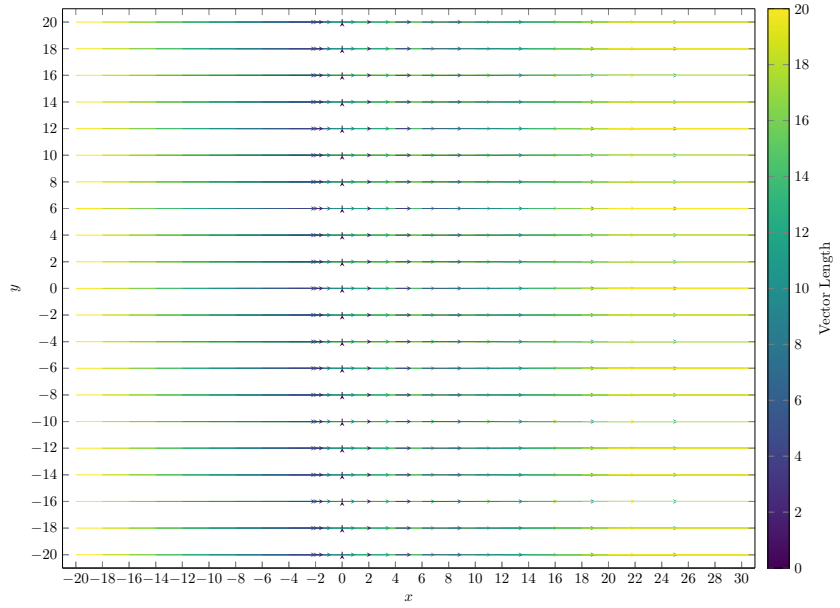


FIGURE 1.1: Transition system $(\mathbb{Z} \times \mathbb{Z}, \delta)$ in the interval $[-20, 20] \times [-20, 20]$. The behaviour of pairs (x, y) preserves the value of y. In this diagram and in the following ones, the length of the transition is encoded through the colour.

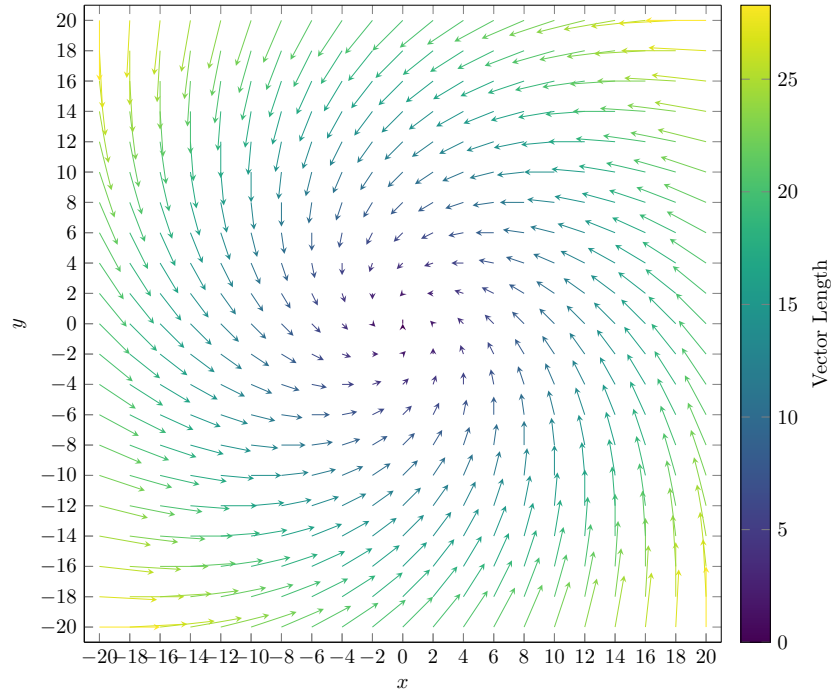


FIGURE 1.2: The spatial transformation m as a transition system in the interval $[-20, 20] \times [-20, 20]$. The function m rotates pairs by 90 degrees counter-clockwise.

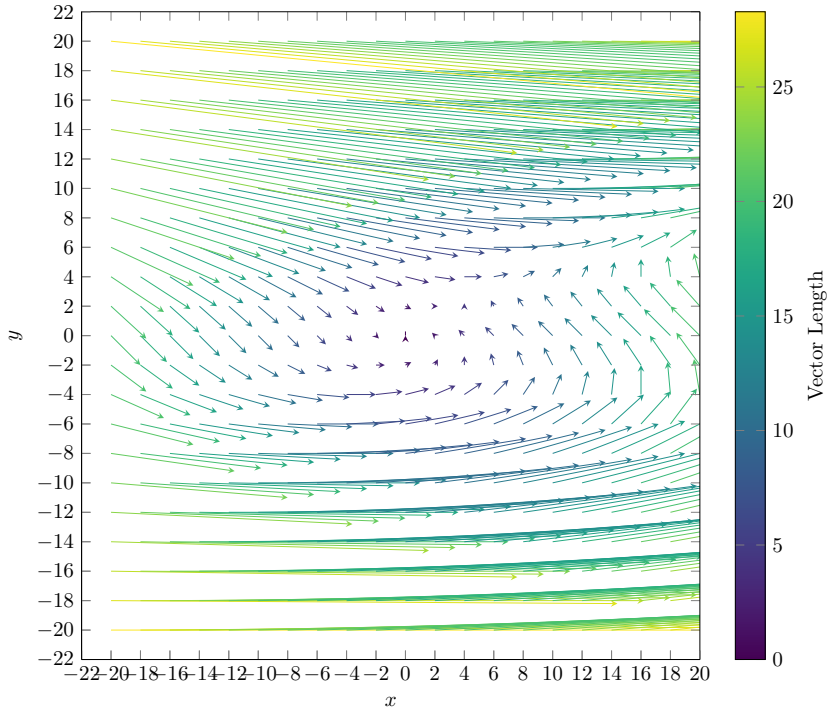


FIGURE 1.3: Latent behaviour of $(\mathbb{Z} \times \mathbb{Z}, \delta)$ revealed by the spatial transformation m in the interval $[-20, 20] \times [-20, 20]$.

a pair (x, y) is now very different: the iterated application of the function $\delta \circ m$ returns $(-y, x^2), (-x^2, y^2), (-y^2, x^4)$, etc. Although the behaviour of states in the systems $(\mathbb{Z} \times \mathbb{Z}, \delta)$ and $(\mathbb{Z} \times \mathbb{Z}, \delta \circ m)$ are quite different, their sole difference is the presence of the spatial transformation m.

In this thesis, we intend to model attackers of a system as sets of spatial transformations. Using these spatial transformations, the attackers cause the system to reveal new behaviours only by affecting the state space, leaving the transition function intact. This implicitly creates a "game" between attacker and system: the attacker moves first and transforms the initial state, the system then moves by applying its transition function, and then the attacker moves again, transforming the resulting states. The system moves again, and this process iterates. We do not consider cases where the attacker takes two turns in a row or where the system takes two turns in a row, since we can emulate them by an intercalated game. More precisely, an attacker that takes two turns in a row to transform a state x into a state y and then to a state z is equivalent to an attacker that takes only one turn and transforms x into z directly. A system that takes two turns in a row by mapping a state a to a state b and then to a state c is equivalent to a turn-based game between an attacker and system where the attacker used an identity transformation to map a to itself and b to itself.

An attacker that can only apply an identity spatial transformation (i.e., one that maps a state to itself) has only trivial capabilities, and fits a passive attacker model that can only watch as the system executes. An attacker that can apply any spatial transformations can manipulate the state space of the transition system as they wish, rendering

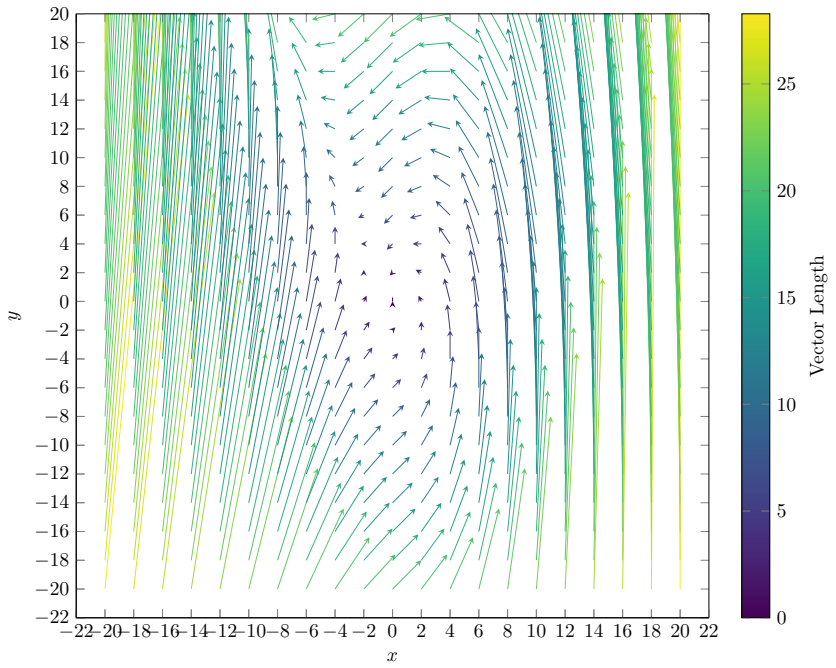


FIGURE 1.4: Dual latent behaviour of the system $(\mathbb{Z} \times \mathbb{Z}, m)$ revealed by δ in the interval $[-20, 20] \times [-20, 20]$. It is a rotation of the behaviour shown in Figure **??**.

the dynamics of the system obsolete because they can be bypassed via spatial transformations. We define realistic attacker models by imposing restrictions over the spatial transformations that are available to an attacker. For example, the attackers described by Basin and Cremers in [**KnowYourEnemy**] cannot transform every state of the protocol; they can only transform the states that satisfy the premises of the compromise rules available to them, and the transformation must implement the effects modelled by the compromise rule.

Not all systems have their dynamics described in terms of endofunctions. For example, reactive systems require an external inputs to continue execution. To define a unified methodology for the application of spatial transformations in transition systems, we use *universal coalgebra*.

1.4 Universal Coalgebra and Spatial Transformations

To soundly extend the applicability of the “game" between spatial transformations and deterministic systems to other models of computation, we adopt the framework of *universal coalgebra* [**UniversalCoalgebra**]. Several authors have used universal coalgebra to generalise results previously exclusive to certain computational models; e.g., the *powerset construction*, a standard procedure to discretise a non-deterministic finite automaton, generalises to other computational effects beyond non-determinism [**GeneralisingDeterminat**
or e.g., Kleene’s Theorem [**KleenesTheorem**], which states that every deterministic finite automaton has an equivalent regular expression, generalises to Mealy machines [**KleeneCoalgebra**]

In universal coalgebra, a dynamical system is often represented as a pair (X, c) where X is a set, called the *carrier*, and $c\colon X \to F(X)$ is a function where $F(X)$, informally, is the set of the features and computational effects that the elements of X may have (e.g., continuation, termination, input/output, etc.). Using the framework of universal coalgebra, our definitions naturally adapt from one computational model to another, widening their applicability.

1.5 Goal of the Thesis and Outline

The main objective of the thesis is to illustrate the usefulness and versatility of parametrising the study of transition systems with a spatial transformation that reveals latent behaviours, a process that we call *latent behaviour analysis* (LBA). These latent behaviours capture extraordinary semantics that are realisable by the systems under the effect of state corruption or alterations; using LBA we can systematically study the behaviour of systems in the presence of adversaries or under the influence of property enforcers. We use universal coalgebra to define LBA in an abstract setting, and we illustrate its applicability in three practical scenarios: the classification of attacker models of a reactive system, the quantification and improvement of robustness in cyber-physical systems (CPS), and the enforcement of timing side-channel freedom in LLVM-IR programs.

We divide this thesis into two major modules: *theory* and *applications*. The theory module presents LBA in the framework of universal coalgebra (Chapter **??**), and the application module shows three different instances of the application of LBA to three practical problems: the classification of attacker models in Chapter **??**, the quantification and repair of robustness in CPS in Chapter **??**, and the enforcement of timing side-channel freedom with respect to memory access patterns for LLVM-IR programs in Chapter **??**.

1.6 Research Questions and Summary of Contributions

To accomplish the goal of this thesis, we study the following research questions. When appropriate, we analyse these questions in the context of the application scenarios for LBA that we present in the following chapters.

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

We approach this question by modelling attacker models with sets of capabilities that create spatial transformations that model the attacks available for attackers fitting that model. We discuss this question in detail in Chapters **??**, **??**, and **??**.

Research Question 1.2. How do we efficiently quantify the harmful effects of an attacker fitting a given attacker model on a system?

The presence of an attacker does not necessarily imply a security risk, e.g., if the attacker can only slightly deviate the behaviour of the system without breaking any security requirements. To address this question, we propose a quantification method that uses bounded model checking in Chapter **??** and that uses testing in Chapter **??**.

Research Question 1.3. How do we compare attacker models and attackers concerning the impact that they potentially have on a given system?

Attackers that have more interaction points with a system are not necessarily the most dangerous. We take inspiration from Basin and Cremers [**KnowYourEnemy**] to approach this question. To determine which attackers are more threatening, we use the quantification methods proposed in Chapter **??** and Chapter **??** to obtain a partial order of attackers relative to their harmful effects on systems. In particular, thanks to our modelling of attackers via sets of spatial transformations, the partial order of attackers that naturally arises from set inclusion has a monotonicity property for the harmful effects of attackers on a system. We explain this monotonicity property in Chapters **??** and **??**.

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

Finally, we approach this broad question from two directions: first, we show how LBA can suggest counterattacks for identified attacks in Chapters **??**, and second, we explain how to use spatial transformations to enforce a security property (i.e., timing side-channel freedom) in Chapter **??**.

Chapter 2

Preliminaries and Notation

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 $f(x) = 1$. We denote the set of subsets of X by 2^X and by $\mathscr{P}(X)$.

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 $y \in Y$.

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 $(f, g)(x) = (f(x), g(x))$.

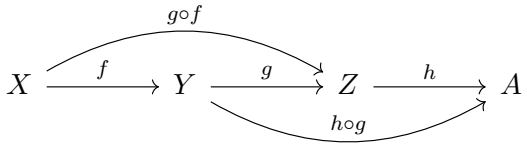


FIGURE 2.1: Commutative diagram describing the equality $(h \circ g) \circ f = h \circ g \circ f = h \circ (g \circ f)$

2.1.12 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 **??**.

2.2.3 Terminal Objects

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

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 $f : X \to Y$, f is mapped to a function of type $g : F(X) \to F(Y)$,
- $\mathtt{id}_X$ is mapped to $\mathtt{id}_{F(X)}$; i.e. $F(\mathtt{id}_X) = \mathtt{id}_{F(X)}$.
- if $f : X \to Y$ and $g : Y \to Z$, 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 range set 1, i.e. $\{0\}$, is the final object in $\mathbf{Set}$.

2.3.1 Partial Application

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

2.3.2 Monoids and Sequences

A *monoid* is a set X paired with a sum function $+ : X \to X \to X$ 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 $\mathtt{id}_X$.

2.3.3 Languages

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

2.3.4 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, $\delta : X \times A \to X$ is a transition function, and $F : X \to 2$ 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.

2.4 Universal Coalgebra Preliminaries and Notation

2.4.1 F-coalgebras

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*.

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 $x_0\colon 1 \to X$ is a function where $x_0(0)$ 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

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

F-homomorphisms are embeddings that preserve behavioural properties.

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 of functions $((\cdot)?, (\cdot)')$ as 1_F. The pair $(\sigma F, 1_F)$ is the *final F-coalgebra*, since it is the final object in the category of F-coalgebras $\mathbf{Coalg_F}$.

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 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 that the automaton would recognise if x were the initial state.

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 denotational semantics.

2.4.7 Coalgebraic Specification Language

Given a functor F , a *coalgebraic specification language* for F is an F-coalgebra $(\mathcal{E}, c)$ where $\mathcal{E}$ is a set of expressions whose operational semantics are given by c, and its denotational 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^*.

2.4.8 Completeness

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

2.4.9 Soundness

Given a coalgebraic specification language $L = (\mathcal{E}, c)$ for a functor F 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* with respect to $\equiv$ if and only if, for $e_1, e_2 \in \mathcal{E}$, if $e_1 \equiv e_2$, then $e_1 \sim e_2$. In other words, if two expressions are equivalent, then they have the same behaviour.

2.4.10 Consistency

Given a coalgebraic specification language $L = (\mathcal{E}, c)$ for the functor F 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 *consistent* with respect to $\equiv$ if and only if, for $e_1, e_2 \in \mathcal{E}$, if $!_c(e_1) = !_c(e_2)$ then $e_1 \equiv e_2$. More precisely, if two expressions have the same denotational semantics, they must be considered equivalent.

2.4.11 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 $(P \circ !_c)(x) = 1$.

We use behavioural properties to formalise properties of a system. Behavioural properties come in all shapes and forms, and are often described using logics like LTL [**LTL**]. CTL [**CTL**], and μ calculus [**MuCalculus**].

Chapter 3

Foundations of Latent Behaviour Analysis

3.1 Introduction

A *latent behaviour* is an unintended functionality of a system, which the system typically does not display. Still, given extraordinary circumstances, the system displays such functionality. To reveal a latent behaviour, we transform the state space of the system. Since the state space is usually not transformed, we model "extraordinary circumstances" using these transformations. This concept is probably well illustrated by *return-oriented programming* (ROP) as described in [**ROP**], where the attacker, to alter the behaviour of the running program, does not modify the code of the running program and instead transforms the state of the call stack.

In ROP, the adversary manipulates the call stack to string code gadgets together and arbitrarily execute code. However, if the program is small and does not have enough gadgets, then the attacker's computational power is restricted since it becomes impossible to create certain behaviours. This restriction is similar to the limitations 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 **??**. This diagram, which we refer to as *the Arrow*, illustrates how spatial and dynamics transformations affect the behaviour of an F-coalgebra (X, c) by changing the semantic map from $!_c$ to $!_{b \circ c \circ m}$. The intuitive meaning of 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 best, we can conclude 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 $(\sigma F, 1)$ where ω maps the original behaviours of the system into the new behaviours revealed by m and b.

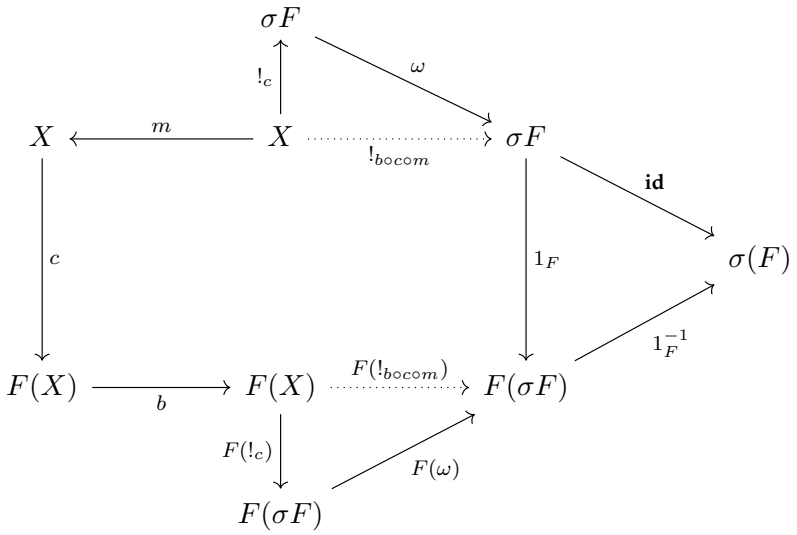


FIGURE 3.1: "The Arrow of Latent Behaviour Analysis." This commutative diagram summarises the effect of spatial and dynamics transformations over the F-coalgebra (X, c), respectively modelled by m and b: the semantic map changes from $!_c$ to $!_{bocom}$. Through this work, we assume $b = \texttt{id}$.

3.2 Motivational Example

Consider the automaton in Figure **??**, 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 implements 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 **??** shows an implementation 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 **??** is the system that is revealed by 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 $\delta_i \colon \vec{X} \to \vec{X}$ 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

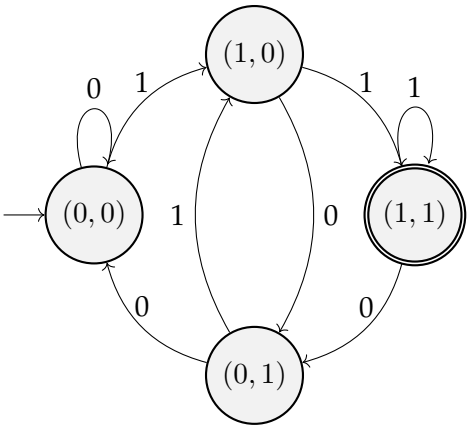


FIGURE 3.2: An automaton which recognises the language $(0|1)^*11$.

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

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

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

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

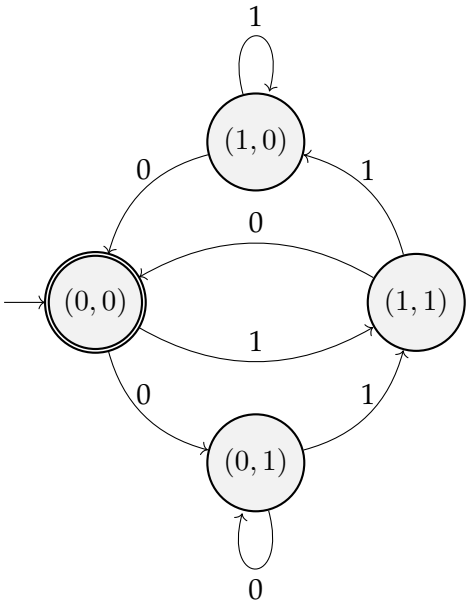


FIGURE 3.3: Automaton that implements the transformed automaton, which now recognises $\varepsilon|(0|1)^*10$.

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 in Figure **??** by "copying" the original behaviour from images of m to their preimages. Figure **??** 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 following: 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 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

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

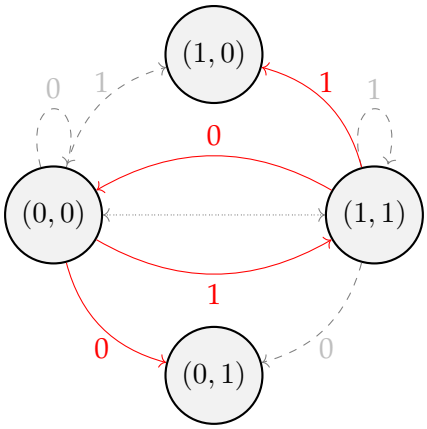


FIGURE 3.4: Composition of the fault m and the original behaviour for the states $(0, 0)$ and $(1, 1)$. The dotted, gray, bidirectional arrow in the centre captures the effect of the fault m. The original behaviours appear as dashed, grey lines. The behaviour which results from the composition appears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

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 *sound and consistent*.

Definition 3.3.1 (Sound and Consistent Spatial Transformations)**.** Given an F-co-algebra (X, c), any function of type $m\colon X \to X$ is a *spatial transformation*. A spatial transformation m is *sound* if and only if $x \sim_c y$ implies $m(x) \sim_c m(y)$, for all $x, y \in X$. Similarly, a spatial transformation m is *consistent* if and only if $m(x) \sim_c m(y)$ implies $x \sim_c y$, for all $x, y \in X$.

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

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

Example 3.1. Consider the automaton from Section **??** 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 behaviour of state $(0, 0)$ is the language $(0|1)^*11$, but its latent behaviour under m is the language $\varepsilon|(0|1)^*10$.

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 $(X, X \xrightarrow{c} F(X))$. We use LBA to understand the behavioural changes that the system (X, c) undergoes due to the application of the transformation m, as illustrated by the following example.

Example 3.2. In the context of Example **??**, consider the behavioural property $Q\colon 2^{2^*} \to 2$, where, $\phi \in 2^{2^*}$, $Q(\phi)$ is true if and only if $\phi \subseteq \varepsilon|(0|1)^*1$, i.e., if ϕ accepts a sequence $w \in 2^*$, then w ends in 1 or w is empty.

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

We can repair the system with respect to the property Q if we use m again in the latent coalgebra $(2 \times 2, c \circ m)$ to reveal the latent coalgebra $(2 \times 2, c \circ m \circ m)$ as $m \circ m = \mathtt{id}$. Thus, if m is an attack, then a second application of m is a valid counterattack with respect to Q.

Our choice to focus on spatial transformations may seem arbitrary since 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 in the "Arrow" in Figure **??**. 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 **??** 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 **??** 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

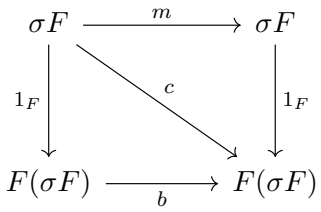


FIGURE 3.5: Every F-coalgebra $(\sigma F, c)$ can be revealed from the final coalgebra $(\sigma F, 1_F)$ by means of a transformation m or a transformation b. In other words, it suffices to use spatial transformations m and the final F-coalgebra to reveal all F-coalgebras of σF, so dynamics transformations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

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 four aspects of systems: *formalisation of integrity attackers*, *quantifying and improving robustness*, *classification of attacker models*, and *side-channel repair*.

3.5.1 Formalisation of Integrity Attacks and Attacker Models

LBA studies the effect of spatial transformations. Henceforth, when we refer to an attack, we implicitly mean a spatial transformation. In the particular case of final F-coalgebras, their spatial transformations attacks coincide with their dynamics transformations.

3.5.2 Classification of Attackers

In Chapters **??** 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 $\langle\alpha\rangle[\mathcal{R}]$ broken by attackers fitting the model $\langle\alpha\rangle$, and we compare these sets of requirements by set inclusion to compare attacker models 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 Quantifying and Improving Robustness

Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a set of behavioural properties that describes both functional and security requirements of a system defined by an F-coalgebra (X, c) with an initial state $x_0 \in X$, and let $m\colon X \to X$ be a spatial transformation that implements an attack or a fault; informally, we say that the system (X, c) is *robust* with respect to m if x_0 still satisfies all the properties in $\mathcal{R}$ despite the effects of the spatial transformation m. To check for robustness with respect to m, we reveal the latent coalgebra $(X, c \circ m)$ and check if this new system satisfies all the behavioural properties in $\mathcal{R}$ using standard verification and testing techniques.

To quantify the robustness of a system, we systematically generate a set of spatial transformations $\mathcal{M} = \{m_1, \ldots, m_p\}$ based on an attacker model, and we 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.*

Spatial transformations can be used to repair systems as well. Given a set of spatial transformations $\mathcal{K} = \{k_1, \ldots, k_q\}$, we say that $k \in \mathcal{K}$ *repairs* (X, c) with respect to a property R if (X, c) does not satisfy R but $(X, c \circ k)$ does. The set $\mathcal{M}$ and $\mathcal{K}$ are both sets of spatial transformations, but the former describes an attacker model, while the latter describes a counterattacker model.

We study the process of system repair in more detail in Chapters **??**. **??**.

3.5.4 Side-Channel Repair

The last application of LBA that we consider is of program repair. We briefly mentioned the idea in Section **??**: 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

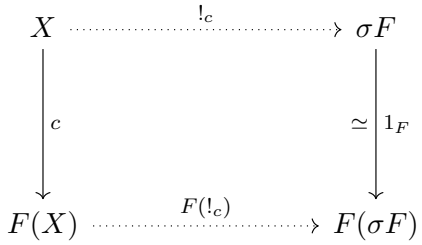


FIGURE 3.6: Typical commuting diagram for the semantic map $!_c$ in universal coalgebra.

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.

3.6 The Arrow: a Summary of LBA

Normally, for a functor F that has a final F-coalgebra $(\sigma F, 1_F)$, any F-coalgebra (X, c) satisfies the commutative diagram in Figure **??**. When we compose a function $m\colon X \to X$ on the right of c and a function $b\colon F(X) \to F(X)$ on the left of c, we obtain the pair $(X, b \circ c \circ m)$, which is an F-coalgebra. Since $(X, b \circ c \circ m)$ is an F-coalgebra, it has its own diagram similar to the one in Figure **??** exists, replacing c by $b \circ c \circ m$. However, the relation between the semantic maps $!_c$ and $!_{b \circ c \circ m}$ is not so simple: in general, the semantic map $!_{b \circ c \circ m}$ is not equal to the function $(!_{b \circ c}) \circ m$, so we cannot directly lift the effects of a given spatial transformation m to the behaviours of an F-coalgebra. At best, if m is sound, then we can create a map $\omega\colon \sigma F \to \sigma F$ that maps the old behaviours of the system to the new ones revealed by m.

This impossibility to soundly lift the effects of m in X to a function in σF indicates that, in general, we cannot deduce properties of latent coalgebras from their original systems, and we should study latent coalgebras as if they were independent systems. However, the properties proved in latent coalgebras act as a proof of behavioural properties of their original systems, as we show in the following chapters. For example, an F-coalgebra (X, c) is robust with respect to a behavioural property Q and a set of spatial transformations M if, for all $m \in M$, the latent coalgebra $(X, c \circ m)$ satisfies the behavioural property Q.

3.7 Related Work

The theory of LBA stems from universal coalgebra [**UniversalCoalgebra**]. The use of the theory of coalgebras has rapidly grown in the past decade, and has found notorious applicability in the domain of program semantics [**GKAT**; **GKATCoequations**]. However, the theory of coalgebras mainly focuses on the correctness and functionality of systems, dismissing the security problem a dual of correctness [**JacobsBook**]; more precisely, if correctness is the description of desired behaviour, then security is

the description of undesired behaviour. This view unfortunately restricts attackers of coalgebras to be external actors that can only interact with the system via clearly defined interfaces. In other words, the user of the system modelled by the coalgebra is the attacker, and only the user can be the attacker. Our proposal of modelling attackers as sets of spatial transformations gives us more flexibility when describing the effects of attackers, while still retaining compatibility with the theory of coalgebras. By modelling an attack as a spatial transformation $m\colon X \to X$ acting on an F-coalgebra (X, c), we naturally incorporate the effects of the attack into the behaviour of the coalgebra.

Chapter 4

Systematic Classification of Attacker Models

4.1 Introduction

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) [**Gunther1990**; **Menezes1996**]. 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.

The Research Questions **??**, **??** and **??** pose an interesting problem: we want to systematically generate attacker models like Basin and Cremers did in [**KnowYourEnemy**], and we want to compare them to obtain a "minimal attacker" that causes harm to the system with minimal assumptions. For the first practical scenario where we use LBA, we consider systems whose components are all boolean. The choice of boolean state variables heavily restricts how we define spatial transformations: if an attacker controls some component, then they only have two options – to set it to `false` or to `true`. In this chapter, we study the problem of *attacker model classification in and-inverter graphs*.

An *And-inverter grap* is a directed graph that models a system of equations [**AIGs**; **AIGs2**] with boolean variables, and are useful to describe hardware models at the bit-level [**AIGER**], which is why they have attracted the attention of industry partners including IBM and Intel [**HWMCC2014BM**]. AIGs present a convenient system model

to study the problem of attacker model classification using LBA because we model attacks as spatial transformations, and spatial transformations in AIGs are boolean functions. To be more precise, an attacker controlling a coordinate π has two options when it comes to transforming a state $\vec{x}$: the attacker sets $\vec{x}[\pi]$ to 0 or sets it to 1.

We characterise the attacker models for an AIG whose set of coordinates is Π using subsets of Π, meaning that there are $2^{|\Pi|}$ distinct attacker models. An attacker model $\langle\alpha\rangle$, which controls the set of coordinates $\alpha \subseteq \Pi$, has $2^{|\alpha|}$ spatial transformations that they can use to manifest latent behaviours, since each coordinate in α is assigned either 0 or 1 by the effect of attacks in $\langle\alpha\rangle$. Given a set of security requirements R, the attacker model $\langle\alpha\rangle$ poses a threat to the system if there exists at least one attack within its capabilities that breaks at least one requirement in R.

Computing the set of requirements that $\langle\alpha\rangle$ breaks is not a trivial process. Each attacker model has an exponential number of attacks available to them, so verifying one by one is an impractical approach. Moreover, extending this analysis to all attacker models only exponentiates the size of this problem. A brute force algorithm would enumerate all attacker models by systematically choosing a subset α in the set of coordinates Π, and then 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. In this chapter, we present two strategies to produce meaningful results in a sensible amount of time. First, 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). Second, 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 summary, we present a bounded model checking strategy that uses these two observations, and provides a possible answer to Research Questions **??**, **??** and **??** by illustrating how this model checking strategy efficiently maps attackers to the set of requirements that they break.

4.2 AIG Preliminaries

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

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 $\vec{w}(t)$.

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 $e_1, e_2 \in E$.

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.

Example 4.1. Consider the AIG shown on the left of Figure **??** with $W = \{w_1, w_2\}$, $V = \{v_1\}$ and $G = \{g_1, g_2\}$. This AIG models the system of equations

$$v_1(0) = 1, \qquad v_1(t+1) = \neg g_2(t),$$
$$g_1(t) = \neg w_1(t) \wedge \neg w_2(t), \qquad g_2(t) = g_1(t) \wedge \neg v_1(t).$$

The set of states of this system is $\vec{V} = 2^{\{v_1\}}$, its set of inputs is $\vec{W} = 2^{\{w_1, w_2\}}$, 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.

Cone of Influence (COI) and its dual. 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 $\blacktriangledown(e) = \{w\}$;

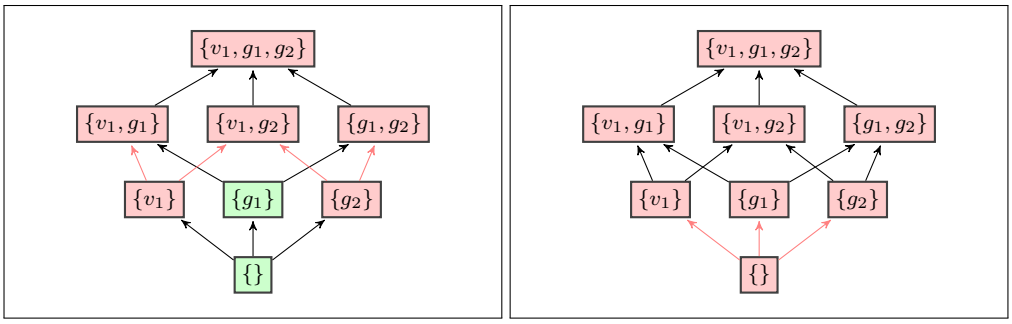


FIGURE 4.1: Left: classification of attackers for requirements r_2 and r_3. Right: classification of attackers for requirement r_1. A green attacker model cannot break the requirement, while a red attacker model can. The red arrows originate from minimal attacker models.

- if $e = v$ 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 $\blacktriangledown(r) \triangleq \blacktriangledown(e)$.

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 $\blacktriangle(c) \triangleq \{c' \in C | c \in \blacktriangledown(c')\}$.

Example 4.2. For the AIG in Figure **??**, the COI of the requirements are $\blacktriangledown(\Box g_1) = \{g_1, w_1, w_2\}$, and $\blacktriangledown(\Box g_2) = \blacktriangledown(\Box g_3) = \{g_1, g_2, v_1, w_1, w_2\}$.

4.3 Motivational Example

Consider the AIG in Figure **??**; assume that an attacker A controls the gate g_2 of the AIG shown on the left of Figure **??**. 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.

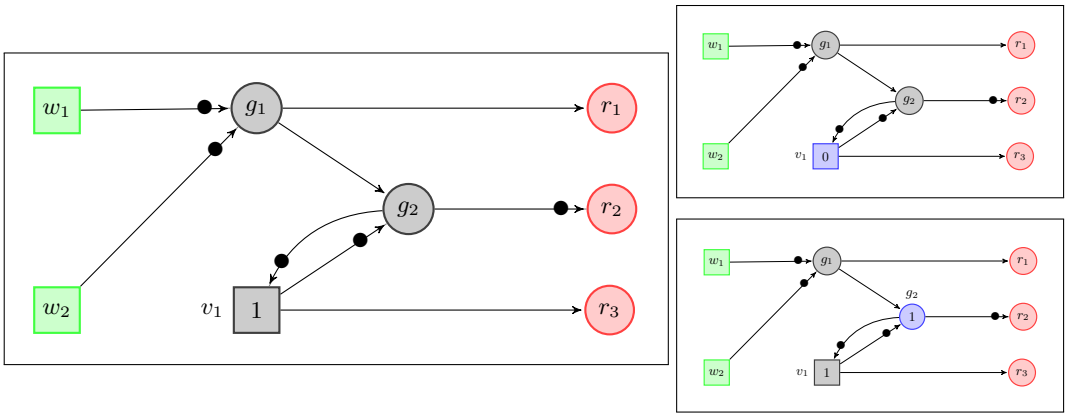


FIGURE 4.2: **Left:** An AIG with two inputs w_1 and w_2 (green boxes), one latch v_1 with initial value 1 (grey box), two gates g_1 and g_2 (gray circles), and three invariant requirements $r_1 = \Box g_1$, $r_2 = \Box \neg g_2$ and $r_3 = \Box v_1$ (red circles). The arrows represent logical dependencies, 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.

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 **??** 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.

4.4 AIGs as F-coalgebras

If we go by the strict definition of AIGs (see Section **??**), then only latches are part of the states of AIGs; inputs and gates are external. We, however, include gates into the state when we model AIGs as F-coalebras. 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.

4.5 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.

4.5.1 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

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

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 4.5.1 (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 4.5.2 (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 **??** 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 4.5.3 (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.

4.5.2 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.* **[BMCWithoutBDDs]**, we transform the equations into a Conjunctive Normal Form formula (CNF) that the SAT solver can work with using *Tseitin encoding* **[TseitinEncoding]**. 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

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

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

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

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 4.5.1. 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 **??**, 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

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 $A[R]_t$.

```
Map H;
foreach r ∈ R do
    foreach A such that A ⊆ (V ∪ G) do
        if check(r, t) is satisfiable while assuming c↓ for all c ∈ A then
            insert r in H(A);
        end
    end
end
return H;
```

Algorithm 1: Naive attacker model classification algorithm.

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 $k \leq t$). □

Algorithm **??** 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 **??** 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 $\mathcal{O}(|W \cup V \cup G| \times t)$ on the worst case.

We now propose two sound heuristics in an attempt to improve Algorithm **??**: 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.

4.5.3 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 4.5.2 (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 [**ToSplitOrToGroup**; **GraphLabelingForEfficientCOIComputation**; **HandbookOfSatisfiability**; **HandbookOfModelChecking**; **OptimizedModelCheckingOfMultipleProperties**]), and it transforms Equation **??** 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 4.5.3 (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.

4.5.4 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 **??**, 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 $\texttt{check}(r, t)$ is not satisfied, then no attacker model breaks r, otherwise we contradict monotonicity.

Corollary 4.1. From monotonicity and isolation (cf. Propositions **??** and **??**), 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↓ for all c ∈ r^max then
2  |  return ∅;
3  end
4  Set: P = {∅}, M = ∅;    //(P contains the empty attacker model ∅)
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 ∈ M such that B ⊆ A) then
8  |  |  if check(r, t) is satisfiable when assuming c↓ for all c ∈ A then
9  |  |  |  insert A in M;
10 |  |  else
11 |  |  |  foreach c ∈ (r^max − A) do
12 |  |  |  |  insert A ∪ {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 **??** 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 4.3. We recall the motivational example from Section **??**. 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 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 **??** applies Algorithm **??** 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 **??** 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 4.4. For the motivational example in Section **??**, Algorithm **??** 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 **??** 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.

4.5.5 On Soundness and Completeness

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

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 $\mathcal{H}$.

```
Set: M = ∅;
Map: H;
for r ∈ R do
    for A ∈ MinimalAttackerModels(S, t, r) do
        insert r in H(A);
        insert A in M;
    end
end
return (M, H);
```

Algorithm 3: The `AllMinimalAttackerModels` algorithm.

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 $A[R]$.

```
(M, H) = AllMinimalAttackerModels(S, t, R);
foreach A ⊆ (V ∪ G) do
    foreach A' ∈ M do
        if A' ⊆ A then
            insert all elements of H(A') in H(A);
        end
    end
end
return H;
```

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

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 [**HandbookOfModelChecking**], 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 **??** 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 **??** 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.

4.5.6 Requirement Clustering

Property clustering [**ToSplitOrToGroup**; **HandbookOfSatisfiability**; **OptimizedModelCheckingOfMul** 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 $\blacktriangledown(Y) = \bigcup\{\blacktriangledown(r) | r \in Y\}$.

4.6 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 [**HWMCC2011**; **HWMCC2013**]) 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 this evaluation. As of 2014, the benchmark set contained 230 different instances, coming from both academia and industrial settings [**HWMCC2014BM**]. We quote from [**HWMCC2014BM**]:

> "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 [**Cadical**]. The source code of the artefact is available at [**aig-ac-asset**].

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.

4.6.1 Evaluating Monotonicity and Isolation

Given a set of competing classification methodologies $\mathcal{M}_1, \ldots, \mathcal{M}_n$ (e.g., Algorithm **??** and Algorithm **??**), 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 4.6.1 (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.

4.6.2 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 **??**, which uses both isolation and monotonicity
2. $(+IS, -MO)$: Algorithm **??** but removing the check for monotonicity on Algorithm **??**, Line 7;
3. $(-IS, +MO)$: Algorithm **??** but using Equation **??** instead of Equation **??** to remove isolation while preserving monotonicity; and
4. $(+IS, +MO)$ Algorithm **??**, 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 **??** illustrates the average coverage for the four different methodologies, for each of the seven benchmarks. The exact coverage values are reported in Tables **??** and **??**. 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 **??** 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 **??**.

4.6.3 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.

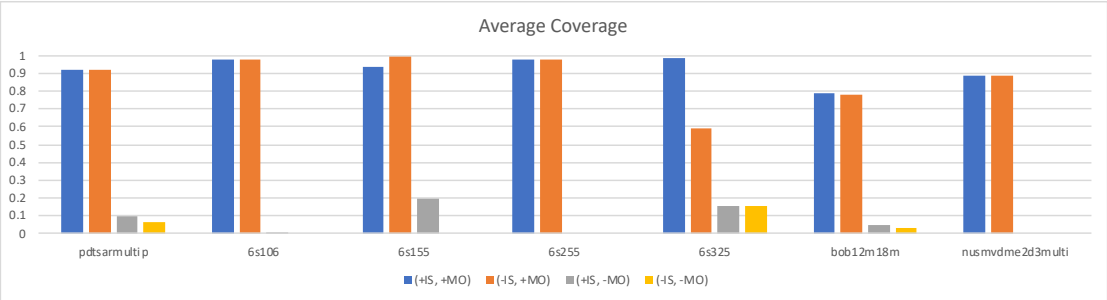


FIGURE 4.3: Average requirement coverage per benchmark. A missing bar indicates a value that is approximately 0.

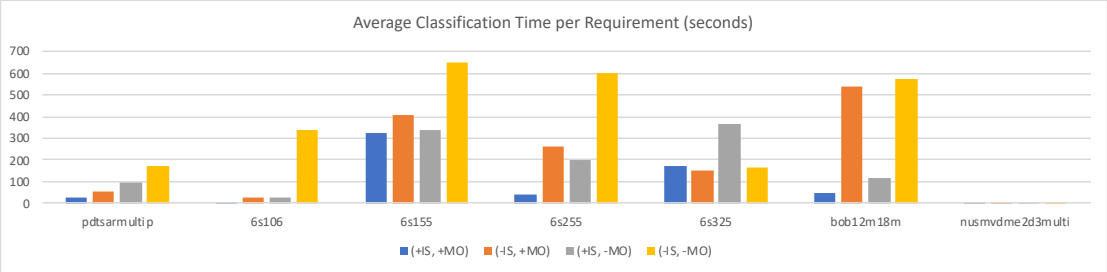


FIGURE 4.4: Average classification time per requirement per benchmark. A missing bar indicates a value that is approximately 0.

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 **??**.

Normally, the attacker model classification behaves in a similar way to what is reported for requirement $\Box\neg g_{2177}$, shown in Table **??**. 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., as shown in Table **??**), and 2) the number of minimal attacker models decreases while the coverage increases, as in Tables **??** and **??**.

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.

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

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 **??** and **??**. 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 **??**, column Cov. The low coverage for small t is due to the restriction on the size of 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 **??** 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 $(-IS, -MO)$.

4.7 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 [**KnowYourEnemy**] 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 [**Giraldo2018**] model attackers as sets of components (e.g., some sensors or actuators), while other works like [**IFCPSSec**; **Cardenas2011**; **Urbina2016**] 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 [**CPSProfiles**] 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 [**KnowYourEnemy**] and [**Giraldo2018**]. The authors of [**KnowYourEnemy**] 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 [**Giraldo2018**]), 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 [**GraphLabelingForEfficientCO** Cabodi et. al [**ToSplitOrToGroup**], and Cabodi and Nocco [**OptimizedModelCheckingOfMultiplePro** 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.* [**JustAssume**] 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 **??** into Equation **??**. 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 [**Interpolation**]), but such optimisations are beyond the scope of this work.

On Completeness. As mentioned in Section **??**, 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. [**ProvingMorePropertiesWithBMC**]) to incorporate attacker models. Alternatively, methods that compute a good approximation of the completeness threshold (e.g. [**EfficientComputationOfRecurrenceDiameters**]) should not only improve coverage in general but also the reliability of the resulting attacker model classifications. Interpolation [**Interpolation**] could also help finding a guarantee of completeness.

Finally, if we consider alternative verification techniques to BMC, then IC3 [**IC3**; **IC32**] and PDR [**IC3**], 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 [**BMCWithoutBDDs**; **lmcs:2236**]. 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.

4.8 Discussion

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

Finding minimal attacker models with only one coordinate is relatively simple since AIGs are systems where a fault quickly propagates through the system. Informally, this means that the systems we studied were not very robust: all it takes is for one of several components to fail or be attacked to cause these systems to fail their security requirements. In retrospective, we believe that using SAT solvers and BMC is reasonable for the computation of the power of a single attacker with several capabilities; however, BMC might have overcomplicated our approach to the problem of systematic attacker classification, where we study several attacker models. Maybe it would have sufficed to study these multiple attackers using a testing approach instead of a verification approach. Consequently, we switch from verification to testing in Chapter **??**.

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 **??**)

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 (Continued from Figure **??**)

Chapter 5

Improving the Robustness of Cyber-Physical System Models

We now consider a practical scenario where we use spatial transformations to model both attacks and counterattacks. From the perspective of LBA, attacks and counterattacks are distinguishable only by intent: an attack is a spatial transformation that aims to damage a system, while a counterattack is a spatial transformation that tries to fix it. Recall the automaton from Section **??**; the spatial transformation that affected the system was a symmetry m that is its own inverse (i.e., $m \circ m = \mathtt{id}$), so we can counter m by applying it a second time.

The inclusion of counterattacks alters the interaction between attacker and system. Up until now, the LBA of a system (X, c) with respect to a spatial transformation $m\colon X \to X$ that reveals the latent system $(X, c \circ m)$, which models the the system (X, c) under the effect of the attack m. In this chapter, we consider a second spatial transformation $w\colon X \to X$ that reveals the system $(X, c \circ w \circ m)$, which models the system (X, c) under the effect of the attack m, but also under the effect of the counterattack w.

5.1 Introduction

A *cyber-physical system* (CPS) is a networked specialised computer network that monitors and controls physical components. Some examples of CPSs include autonomous vehicles, aircraft, water treatment plants, industrial control systems, and critical infrastructures. A subgroup of CPS, called *infrastructure* or *industrial control systems* (ICS), are responsible for critical services in modern societies, such as access to clean water and transportation. Security violations in infrastructure have significant consequences in the physical world, *e.g.*, in late 2007 or early 2008, the Stuxnet attack against an Iranian control system allegedly sabotaged centrifuges in uranium enrichment plants, causing them to rapidly deteriorate [**StuxnetWeb**; **Stuxnet**]. In 2014, hackers struck a steel mill in Germany and disrupted the control system, which prevented a blast furnace from properly shutting down, causing massive damage to the facility [**WiredArticle**; **Lagebericht2014**]. Other infrastructures affected by cyberattacks are a power grid in Ukraine [**cherepanov2017industroyer**] and oil systems in the middle-east [**johnson2018attackers**].

Protecting the integrity of CPSs is paramount. This is especially true for ICSs; their security priority is not the protection of confidential data, but the protection of their

physical assets. Gollmann and Krotofil reinforce this view in [**CPSSec**], stating that the traditional CIA (Confidentiality-Integrity-Availability) triad should be reversed when studying the security of CPSs. They argue that the enforcement of integrity in CPSs should not only consider the "traditional approach for IT systems", i.e., protecting component logic and communication, but that we must also protect the integrity of observations; more precisely, we have to protect the veracity of sensor data and check its plausibility.

A *robust* CPS can tolerate the presence of external perturbations (i.e., attacks or faults). In other words, the integrity of the system is preserved in the presence of adversaries. Traditionally, the enforcement of CPS integrity has used control-theoretic methodologies, specifically fault-detection techniques. Control theory researchers model attacks on CPSs as time-series with specific structures affecting sensor measurements and/or control commands. Depending on their capabilities, attackers have the power to control when, where, and how attacks enter the system. Research on attack detection and mitigation has mainly focused on the so-called *integrity attacks*, i.e., attacks that put at risk the proper operation and physical integrity of CPSs. Integrity attacks include stealthy attacks [**CPSStealthAttacks**], message replay [**CPSReplayAttacks**], covert attacks [**CPSCovertAttacks**], and false-data injection [**CPSDataInjectionAttacks**], among others. Due to their focus on fault-detection techniques, many results based on control theory aim to protect CPSs from integrity attacks in two ways: one, by first verifying that their system satisfies a series of security guarantees during normal operation (i.e., attack-free operation), and second, by relying on monitoring the physics of the system to detect anomalies (see, e.g., [**CPSInvariantsForDetection**; **Urbina2016**; **CPSAttackDetection**; **CPSAttacksAgainstPCS**; **CPSIntegrityAttacks**; **CPSDetectingIntegrityAttacksS**

From the computer science perspective, we can estimate the robustness of software systems by systematic *fault injection* through techniques like fuzzing (see [**Fuzzing**]). During fault injection, we provide semi-random inputs to the system in the hopes of creating new behavioural traces that may be insecure. Normally, an insecure systems is a system with abnormal termination, but we can extend this notion to other security properties if we have a decision procedure to determine when an input is successful at breaking the security properties. In our CPS scenario, the inputs that we use to quantify robustness are the attacks that correspond to some given attacker model. We also assume the existence of a set of security requirements $\mathbb{P}$ that we can evaluate in the CPS. Henceforth, we consider a CPS to be robust with respect to a set of attacks $\mathcal{M}$ and a set of requirements $\mathbb{P}$ if no attack in $\mathcal{M}$ can break any security requirement in $\mathbb{P}$.

Following the LBA methodology, we model attacks as spatial transformations, and we compose these transformations with the original system to create a latent system. We test the latent systems to quantify the robustness of the original system: if a latent system fails a security requirement, then the CPS is vulnerable to the attack that reveals the latent system. Similar to the control theoretical approaches, we expect that the system under normal circumstances satisfies all the security requirements in $\mathbb{P}$. However, it is not unexpected that the system fails to satisfy one or more security requirement while it is under attack.

To protect the CPS in scenarios that are not attack free, we propose the use of *counterattacks*. A counterattack is a second spatial transformation that aims to counter the effect of one or more attacks. More precisely, if we model a CPS whose set of states is Π and whose transition function is $[\![\cdot]\!]\colon \Pi \to \Pi$, we can compose $[\![\cdot]\!]$ with a state transformation $m\colon \Pi \to \Pi$ that models an attack and reveal a the latent CPS $(\Pi, m \circ [\![\cdot]\!])$. To counter m, we introduce a second spatial transformation $w\colon \Pi \to \Pi$ to reveal a latent CPS $[\![\cdot]\!] \circ w \circ m$ that satisfies the following property: if the system $(\Pi, [\![\cdot]\!] \circ w \circ m)$ satisfies all security requirements that the original CPS does, then w counters the attack m.

Theoretically, for a system $(\Pi, [\![\cdot]\!])$, it is possible to arbitrarily choose values for the spatial transformations m (attack) and w (counterattack) from the set of endofunctions Π^{Π}. However, in the case of m, this set of transformations models an attacker that is capable of arbitrarily morphing the state of the CPS, including both cyber and physical aspects, in any form and at any time. This attacker is trivially too powerful to defend against, not to mention impossible, since it would be capable of bypassing physical invariants (e.g. Boyle's law). To model realistic attackers, we use attacker models that are a combination of simple actions that affect individual components (e.g., set a fixed value in a sensor). Now, since attacks and counterattacks are duals, we model realistic counterattackers by using a combination of simple actions that affect individual variables in the controllers of the CPS. We also illustrate how, based on the interactions between attacks and counterattacks, the designers of CPSs can derive possible repairs for the CPS such that the robustness of the system provably increases.

Problem statement: To summarize, in this work we address the following questions: how can we systematically generate realistic attacks and counterattacks for a CPS model? How can we improve the design of the CPS at design time and justify our models formally with respect to a quantitative notion of robustness?

Summary of our Approach: We propose to model a CPS as a composition of the control logic and the physical system in terms of a deterministic transition system. We then formally quantify the impact of various attackers using the following testing approach: using LBA, we obtain a family of systems that model the CPS in the presence of attackers and counterattackers, which we test via simulation to identify successful attacks and counterattacks. We quantify the robustness of the system by counting the number of successful attacks, and we quantify a potential for improvement by counting the number of successful attacks that can be countered by at least one counterattack. We note that a sound implementation of the counterattacks requires the existence of an attack monitor that has strong precision and correctness requirements, so, alternatively, by observing the interactions between attacks and counterattacks, we propose a repair, i.e., a modification of the program of the controller of the CPS. To assess the success of the repair, we compare the robustness of the repaired version to the robustness of the original system; if the robustness of the repaired system is strictly higher than the robustness of the original, then the repair is successful.

Contributions: We make the following contributions **a)** We present a logical framework to formally reason about robustness in CPS models using spatial transformations, reconciling testing techniques and control theoretical aspects. **b)** We discuss how our approach can be used to model attacks, counterattacks, and quantify the robustness of

CPSs. **c)** We illustrate the usefulness of our approach by means of simple but realistic models concerning a water distribution CPS; we show that we can identify non-trivial attacks and counterattacks, and we show how to implement a repair that provably improves the robustness of the original system.

5.2 Control Theory Preliminaries

Gollmann and Krotofil highlight, "to work on cyber-physical systems, one has to combine the devious mind of the security expert with the technical expertise of the control engineer" [**CPSSec**]. Unfortunately, this expertise is "not commonly found in the IT security community" [**CPSSecVinyl**]. Moreover, the language barrier between control theory and IT security disciplines worsens this lack of expertise. Thus, in this section, we present the model-based techniques for CPSs security broadly used by the systems and control community.

5.2.1 Linear Time-Invariant Models.

During the last decade, there has been an increasing tendency to use physics-based models of CPSs to detect and quantify the effect of attacks on the system performance [**CPSAttacksAgainstPCS**; **Urbina2016**; **CPSDetectingIntegrityAttacksScada**; **CPSIntegrityAttacks**; **Carlos_Justin1**; **Carlos_Justin2**; **ReachableSets**]. These physics-based models focus on the normal operation of the CPS and work as prediction models that are used to confirm that control commands and measurements are valid and plausible. Often, dynamical models of physical systems are approximated around their operation points or approximated using input-output data. These lead to approximated models which are often linear and time-invariant.

Following the work in [**CPSAttacksAgainstPCS**; **Urbina2016**; **CPSDetectingIntegrityAttacksScada**; **CPSIntegrityAttacks**; **Carlos_Justin1**; **Carlos_Justin2**; **ReachableSets**], here, we only consider Linear-Time Invariant (LTI) models, although the same ideas are employed when considering more complicated dynamics. In particular, a model of a CPS that uses LTI stochastic difference equations is of the form

$$\begin{cases} x(k+1) = Ax(k) + Bu(k) + \vartheta(k), \\ y(k) = Cx(k) + \eta(k), \end{cases} \tag{5.1}$$

with $k \in \mathbb{N}$, physical state of the system $x \in \mathbb{R}^n$ (i.e., an n-dimensional vector of physical variables associated with the dynamics of the CPS), sensor measurements $y := (y_1, \ldots, y_m)^T \in \mathbb{R}^m$, control commands $u := (u_1, \ldots, u_m)^T \in \mathbb{R}^l$, real-valued matrices A, B, and C of appropriate dimensions, and i.i.d. multivariate zero-mean Gaussian noises $\vartheta \in \mathbb{R}^n$ and $\eta \in \mathbb{R}^m$ with covariance matrices $R_1 \in \mathbb{R}^{n\times n}$ and $R_2 \in \mathbb{R}^{m\times m}$, respectively. The matrix A describes the *dynamic evolution of the physical process*, B is used to model the effect of *actuators* on the system dynamics, and the matrix C models the part of the state x_k available from *sensor measurements*.

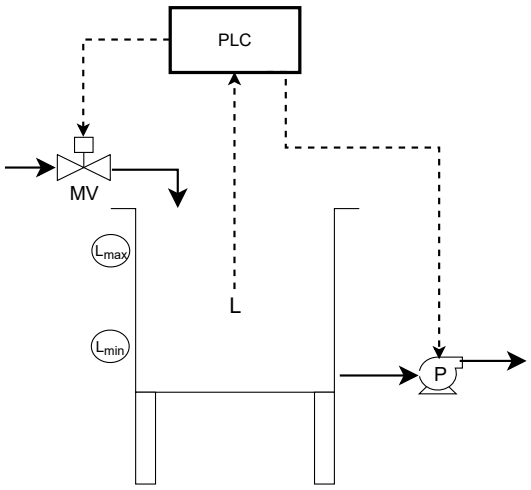


FIGURE 5.1: A water tank. The PLC sends control commands to the motor valve (MV) and the pump (P) so that the level of the tank (L) stays within the range L_{min} – L_{max}.

5.2.2 Attacks on Models of CPSs

At the time-instants $k \in \mathbb{N}$, the output of the process $y(k)$ is sampled and transmitted over a communication network. The received output is used to compute control actions $u(k)$ which are sent back to the physical process. The complete control-loop is assumed to be performed instantaneously, i.e., the sampling, transmission, and arrival time-instants are supposed to be equal. In between transmission and reception of sensor data and control commands, an attacker may replace the values coming from the sensors to the controller and from the controller to the actuators, acting as a man-in-the-middle. Thus, after each transmission and reception, the attacked output $\bar{y}$ and attacked input $\bar{u}$ take the form

$$\begin{cases} \bar{y}(k) := y(k) + \delta^y(k), \\ \bar{u}(k) := u(k) + \delta^u(k), \end{cases} \tag{5.2}$$

where $\delta^y(k) \in \mathbb{R}^m$ and $\delta^u(k) \in \mathbb{R}^l$ denote *additive sensor and actuator attacks*, respectively.

A system under attack is modelled by

$$\begin{cases} x(k+1) = Ax(k) + B(u(k) + \delta^u(k)) + \vartheta(k), \\ y(k) = Cx(k) + \eta(k) + \delta^y(k). \end{cases} \tag{5.3}$$

5.3 Motivational Example: A Water Tank

We now provide a simple example to motivate the problem of quantifying robustness via testing in CPS models. We also illustrate the concepts behind the modelling formalisms presented in the following sections, especially LBA.

Consider the following model of a CPS containing a water tank with a water level (L), a water pump (P), a motor valve (MV), and a programming logic controller (PLC), as shown in Figure **??**. This system has a physical process state vector $\vec{x}$, a cyber controller state vector $\vec{q}$, a sensor state vector $\vec{y}$, and an actuator state vector $\vec{u}$.

The physical state vector $\vec{x}$ has coordinates MV, L and P, where $\vec{x}[MV]$ corresponds to the physical state of the motor valve, $\vec{x}[P]$ corresponds to the physical state of the pump, and $\vec{x}[L]$ corresponds to the physical state of the water level of the tank. The variable $\vec{x}[MV]$ can take four values: `closed`, `opening`, `open`, or `closing`. The variable $\vec{x}[P]$ can take two values: `on` and `off`. Finally, the variable $\vec{x}[L]$ can take any value between 0 L and 1200 L (the maximum capacity of the tank). Depending on the state of the motor valve $\vec{x}[MV]$, different amounts of water flow into the tank. In turn, the state of the valve $\vec{x}[MV]$ changes when it receives the control input $\vec{u}[MV]$, which can take the values `closed`, `opening`, `open`, or `closing`.

The behaviour of the physical process is captured by the following linear time invariant system:

$$\vec{x}[L](k+1) = \vec{x}[L](k) + Inflow(k) - Outflow(k), \quad \text{where} \tag{5.4}$$

$$Inflow(k) = \begin{cases} 0.31, & \text{if } \vec{x}[MV](k) = \texttt{open} \\ 0, & \text{if } \vec{x}[MV](k) = \texttt{closed} \\ 0.26, & \text{otherwise} \end{cases} \tag{5.5}$$

$$Outflow(k) = \begin{cases} 0.15, & \text{if } \vec{x}[P](k) = \texttt{on} \\ 0, & \text{otherwise} \end{cases} \tag{5.6}$$

$$\vec{x}[MV](k+1) = \vec{u}[MV](k) \tag{5.7}$$

$$\vec{x}[P](k+1) = \vec{u}[P](k) \tag{5.8}$$

$$\vec{y}[L](k) = \vec{x}[L](k) \tag{5.9}$$

The value of $\vec{y}[L]$ is a measurement of $\vec{x}[L]$, but since it is a cyber component, it can take values above 1200 (although in normal circumstances it remains in the range $[0..1200]$). We leave the initial conditions of the process abstract for now.

The values of the control command for the motor valve $\vec{u}[MV]$ are defined by the controller, which uses $\vec{y}[L]$ to make decisions. The controller itself has a state $\vec{q}$ that has four coordinates: the presumed state the valve MV, denoted by $\vec{q}[MV]$, the presumed state of the pump P, denoted by $\vec{q}[P]$, the presumed water level of the tank L, denoted by $\vec{q}[L]$, and a timer τ that the controller uses to approximate the time it takes for the motor valve MV to open or close, denoted $\vec{q}[\tau]$.

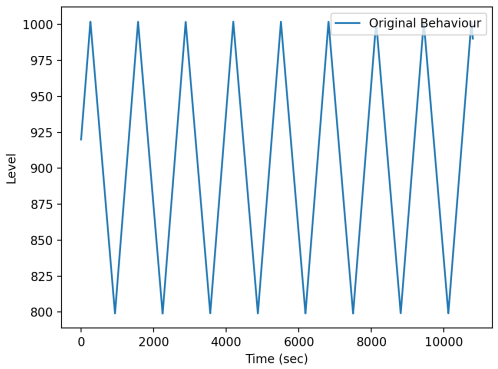


FIGURE 5.2: Normal steady state behaviour of the water level of the water tank.

The controller updates its state using $\vec{y}[L]$, and produces the control values $\vec{u}[MV]$ and $\vec{u}[P]$ using $\vec{q}(k)$, with

$$\vec{q}[MV](k+1) = \begin{cases} \texttt{opening}, & \text{if } \vec{q}[MV](k) = \texttt{closed} \text{ and } \vec{y}[L](k) \leq L_{min}, \\ \texttt{open}, & \text{if } \vec{q}[MV](k) = \texttt{opening} \text{ and } \vec{q}[\tau](k) > T, \\ \texttt{closing}, & \text{if } \vec{q}[MV](k) = \texttt{open} \text{ and } \vec{y}[L](k) \geq L_{max}, \\ \texttt{closed}, & \text{if } \vec{q}[MV](k) = \texttt{closing} \text{ and } \vec{q}[\tau_3](k) > T_3, \\ MV & \text{otherwise;} \end{cases} \tag{5.10}$$

$$\vec{q}[\tau](k+1) = \begin{cases} 1 + \vec{q}[\tau](k), & \text{if } \vec{q}[MV](k) = \texttt{opening} \text{ or } \vec{q}[MV](k) = \texttt{closing} \\ 0, & \text{otherwise;} \end{cases} \tag{5.11}$$

$$\vec{q}[L](k+1) = \vec{y}[L](k) \tag{5.12}$$

$$\vec{q}[P](k+1) = \texttt{on} \tag{5.13}$$

$$\vec{u}[MV](k+1) = \vec{q}[MV](k+1) \tag{5.14}$$

$$\vec{u}[P](k+1) = \vec{q}[P](k+1) \tag{5.15}$$

where $L_{min} = 800$, $L_{max} = 1000$ and $T = 7$. We leave the initial conditions of $\vec{q}$ and $\vec{u}$ abstract for now.

Now that we have a description of the model of the CPS, we define a set of security requirements that determine whether the behaviour of the system is safe or not.

Security Requirements

We establish two simple security requirements for the water tank: the tank is never empty and the tank never overflows. These requirements protect the integrity of the system: if the tank empties, the pump suffers physical damage due to cavitation, and if the tank overflows, then water spills out, causing a work hazard. Formally, these

requirements correspond to the invariants $\Box\vec{x}[L] < 1200$ and $\Box\vec{x}[L] > 0$, defined by

$$\Box\vec{x}[L] < 1200 \triangleq \forall k \geq 0.\vec{x}[L](k) < 1200,$$
$$\Box\vec{x}[L] > 0 \triangleq \forall k \geq 0.\vec{x}[L](k) > 0.$$

Under normal circumstances, the water level of the tank exhibits the behaviour displayed in Figure **??**. The system in its attack-free state satisfies the security requirements $\Box\vec{x}[L] < 1200$ and $\Box\vec{x}[L] > 0$.

Testing and Quantifying Robustness

Now, consider an attacker model where the attacker can replace at any time the value $\vec{y}[L]$ by a value of their choice, or use the real value. This attacker model generates a universe of 64 different *representative classes of attacks*, which define when and what values to use to replace $\vec{y}[L]$. These 64 attacks classes correspond to the paths that the attacker can trigger in the controller by varying the values of $\vec{y}[L]$. More precisely, the attacker has four options: they can introduce a fake low level value (e.g., 500 L), a fake ok level value (e.g. 800 L), a fake high level value (e.g., 1100 L) or preserve the real water level value. The attacker can choose to use different actions in three main scenarios: when the water level is high, normal, or low. This combination of four “what" values over three “when" dimensions defines $4^3 = 64$ combinations. We explain in detail the generation of attack universes in Section **??**.

We measure the robustness of the system with respect to this attacker model by computing the number of attacks that do not break any of the security requirements. To test an attack in the system, we simulate several operation hours of the system while concurrently simulating the effect of the attack until either the simulation timeouts or the system fails a requirement. After testing, we see that 30 out of 64 attacks break at least one requirement, so the robustness of this system is 34/64 with respect to an attacker that has control of $\vec{y}[L]$. We explain the process of computing robustness in detail in Section **??**.

Counterattackers

We now rely on the fundamental observation that an attack m on the original version of the tank reveals a latent system, which we denote T_m, and that a transformation w on the latent system T_m reveals another latent system T_m^w, which hopefully satisfies all security requirements. In such a case, we say that w *is a counterattack of* m.

A *counterattacker model* is dual to an attacker model. It also requires a “what" and a “when" to act. In this scenario, we assume that the counterattacker directly changes the physical state of the pump and the valve, i.e., an operator implements this counterattacker, and they can manually set the coordinate $\vec{x}[P]$ to `on` or `off` and the coordinate $\vec{x}[MV]$ to `open` or `close`, overriding the commands of the controller. We also assume that the counterattacker can observe the value $\vec{y}[L]$ (which is controlled by the attacker in this scenario). From this counterattacker model, we generate a universe of 729 counterattacks (we explain how we obtain the size of the counterattack universe in Section **??**).

All 40 successful attacks can be countered, i.e., for each attack m there exists a counterattack w such that the system T_m^w satisfies all the security requirements. Consequently, we say that the system has a *latent robustness* of 64/64; i.e., it is theoretically possible for a counterattacker fitting the counterattacker model to counter every attack of an attacker fitting the attacker model, given that it is possible for the counterattacker to accurately identify that: 1) the system is under attack, and 2) which exact attack is occurring, since apparently similar attacks could have different counterattacks. Assuming the existence of such monitor is impractical; which is why we instead consider a modification of the program of the PLC based on the information that we obtain from the LBA to improve the robustness of the system. In Section **??**, we perform LBA on a system that is more complex than the water tank, and we show how to use the relations between attacks and counterattacks obtained via LBA to propose repairs that improve the robustness of the model.

5.4 Cyber-Physical Systems, Physical Requirements and Attacker Models

The major difference between CPSs and IT systems is the interaction with the physical world. We share the view of Gollmann *et al.* [**CPSSecVinyl**], who state that CPSs security is specifically concerned with attacks that cause some physical impact; i.e., attack that affect the integrity of the physical process beyond repair. The robustness of a system is a measure of tolerability to those attacks that target integrity.

In this section, we present a framework to quantify the robustness of CPSs. This framework consists of several parts:

1. a discrete model for CPSs and their semantics;
2. a description of physical security requirements;
3. a formalisation of attacker models and attacks for our CPS models;
4. a formulation of counterattacker models and counterattacks, dual to those of attackers and attacks;
5. a quantitative notion of robustness and latent robustness based on physical security requirements, attacker models and counterattacker models.

5.4.1 A Discrete Model for CPSs

Figure **??** presents an extended version of the *supervisor model* [**doi:10.1137/0325013**], which serves as the starting point for our modelling framework. Our CPS model consists of four subsystems: the *controller*, an open, discrete cyber system; the *process*, a linear-time invariant physical system; and two sets of cyber-and-physical interface systems: the *sensors* and the *actuators*. The vector $\vec{q}$ models the current state of the controller, and $\vec{x}$ models the current state of the process. The vector $\vec{i}$ models cyber input measurements and the vector $\vec{o}$ models cyber control outputs; they respectively

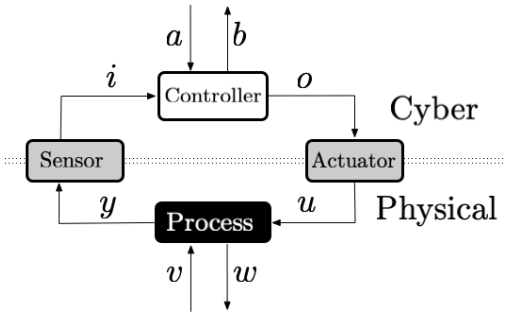


FIGURE 5.3: Extended supervisor model accounting for external inputs and outputs in both cyber and physical components.

model the internal communication channels between controller and sensors, and between controller and actuators. Similarly, the vector $\vec{u}$ models the physical control inputs and the vector $\vec{y}$ models physical measurements; these vectors respectively model the internal communication channels between the process and actuators, and between the process and sensors.

The cyber network input $\vec{a}$ and the cyber network output $\vec{b}$ enable communication between controllers, and enable CPS composition on the cyber side. Similarly, the physical network input $\vec{v}$, and the physical network output $\vec{w}$ model physical channels that connect modules of a larger physical system, enabling the composition of CPS on the physical side. For example, when two water tanks T_1 and T_2 are connected by a pipe, the outgoing flow from T_1 to T_2 (a component of $\vec{w}$ for T_1) should translate into the incoming flow from T_1 to T_2 (a component of $\vec{v}$ for T_2).

Definition 5.4.1 (States of a CPS)**.** A *state of a CPS* π is the coupling of the components of such CPS; formally,

$$\pi = (\vec{a}, \vec{b}, \vec{i}, \vec{o}, \vec{u}, \vec{y}, \vec{v}, \vec{w}, \vec{q}, \vec{x}). \tag{5.16}$$

Let $\vec{A}$ be the *range* of $\vec{a}$; i.e., the set of values that $\vec{a}$ can take. Similarly, we define $\vec{B}$, $\vec{I}$, $\vec{O}$, $\vec{U}$, $\vec{Y}$, $\vec{V}$, $\vec{W}$ $\vec{Q}$, and $\vec{X}$ as the ranges of $\vec{b}$, $\vec{i}$, $\vec{o}$, $\vec{u}$, $\vec{y}$, $\vec{v}$, $\vec{w}$, and $\vec{q}$, respectively. We defined the set Π of the CPS states by

$$\Pi \triangleq \vec{A} \times \vec{B} \times \vec{I} \times \vec{O} \times \vec{U} \times \vec{Y} \times \vec{V} \times \vec{W} \times \vec{Q} \times \vec{X}. \tag{5.17}$$

Finally, we informally define the set of *coordinates* $\mathbb{C}$ to be the set of elements in the CPS model that have no subcomponents. For example, the vector $\vec{u}$ from Section **??** has two subcomponents, $\vec{u}[MV]$ and $\vec{u}[P]$, so $\vec{u}$ is not a coordinate, but both $\vec{u}[MV]$ and $\vec{u}[P]$ individually are. We refer to the elements of $\mathbb{C}$ as the *coordinates* of the CPS.

Informally, the semantics of a CPS is the following: the process, whose state is $\vec{x}$, is observed by the sensor, yielding the physical observation $\vec{y}$; the sensor transforms $\vec{y}$ into the cyber input observation $\vec{i}$. The controller, whose state is $\vec{q}$, receives $\vec{i}$ and a cyber network input $\vec{a}$, and produces a cyber network output $\vec{b}$ and a cyber control command $\vec{o}$. The cyber command $\vec{o}$ is then transformed into a physical control command $\vec{u}$ by the actuator. The process, whose state is $\vec{x}$, reacts to $\vec{u}$ and to some physical network input

$\vec{v}$, returning a physical network output $\vec{w}$ while updating its state. The cycle repeats by having the sensor observe the physical state $\vec{x}$ to produce $\vec{y}$ once more. We now formally define how a state of the CPS evolves over time.

Definition 5.4.2 (Single-Cycle Semantics)**.** Let $\Pi \triangleq \vec{A}\times\vec{B}\times\vec{I}\times\vec{O}\times\vec{U}\times\vec{Y}\times\vec{V}\times\vec{W}\times\vec{Q}\times\vec{X}$ be the set of states of the CPS; we assume the existence of the following functions,

- $\delta_{\vec{Q}}\colon \vec{Q}\times\vec{A}\times\vec{I}\to\vec{Q}$, the transition function of the controller,
- $\beta_{\vec{Q}}\colon \vec{Q}\to\vec{B}$, the network output function of the controller,
- $\theta_{\vec{Q}}\colon \vec{Q}\to\vec{O}$, the cyber control output function of the controller,
- $\theta_{\vec{Y}}\colon \vec{Y}\to\vec{I}$, the translation function of the sensor,
- $\theta_{\vec{O}}\colon \vec{O}\to\vec{U}$, the translation function of the actuator,
- $\delta_{\vec{X}}\colon \vec{X}\times\vec{U}\times\vec{V}\to\vec{X}$, the transition function of the process,
- $\beta_{\vec{X}}\colon \vec{X}\to\vec{W}$, the physical effect function of the process,
- $\theta_{\vec{X}}\colon \vec{X}\to\vec{Y}$, the physical output function of the process.

We define the single-cycle semantics function $[\![\cdot]\!]\colon \Pi\to\Pi$ that describes the evolution of the state $\pi = (\vec{a},\vec{b},\vec{i},\vec{o},\vec{u},\vec{y},\vec{v},\vec{w},\vec{q},\vec{x})$ of the CPS by

$$[\![\pi]\!] \triangleq \left((\vec{a'},\vec{b'},\vec{i'},\vec{o'},\vec{u'},\vec{y'},\vec{v'},\vec{w'},\vec{q'},\vec{x'})\right), \tag{5.18}$$

where $\vec{a'}$ and $\vec{v'}$ are external inputs, and

$$\begin{gathered}\vec{o'} = \theta_{\vec{Q}}(\vec{q}),\quad \vec{q'} = \delta_{\vec{Q}}(\vec{q},\vec{a},\vec{i}),\quad \vec{b'} = \beta_{\vec{Q}}(\vec{q}),\\ \vec{x'} = \delta_{\vec{X}}(\vec{x},\vec{u},\vec{v}),\quad \vec{y'} = \theta_{\vec{X}}(\vec{x}),\quad \vec{w'} = \beta_{\vec{X}}(\vec{x}),\\ \vec{u'} = \theta_{\vec{O}}(o),\quad \vec{i'} = \theta_{\vec{Y}}(\vec{y}).\end{gathered}$$

The evolution over $k+1$ cycles is defined by the iteration of $[\![\cdot]\!]$, i.e., $[\![\cdot]\!]_{k+1} = [\![\cdot]\!]\circ[\![\cdot]\!]_k$. Finally, we define $[\![\pi]\!]_0 = \pi$ for all states $\pi\in\Pi$. Under this definition, the *normal behaviour* of the CPS is characterised by the infinite sequences $(\pi_0,\pi_1,\ldots)$ where $\pi_{k+1}\triangleq [\![\pi_k]\!]$ and π_0 is an initial state.

Because a CPS itself does not create values for $\vec{a}$ and $\vec{v}$, we need a stream of network inputs $(\vec{a_0},\vec{a_1},\ldots)$ and a stream of physical effects $(\vec{v_0},\vec{v_1},\ldots)$ to advance the system. Usually, $\vec{a}$ is the translation of a network output $\vec{b}$ given by another CPS; similarly, $\vec{v}$ is the translation of a physical output $\vec{w}$ from another system. If the functions $\delta_{\vec{Q}}$ and $\delta\vec{X}$ respectively ignore $\vec{a}$ and $\vec{v}$ for their computation, then we say that the CPS is *closed*. Henceforth, we assume that the CPS (or composition of CPSs) we are working with is closed. We study a closed CPSs composition in Section **??**.

From the perspective of coalgebras, this definition of latent behaviour implies that the whole state is observable from the modelling perspective. Implicitly, we are working with a functor $F(X) = \Pi\times X$, modelling the CPS using the F-coalgebra $(\Pi, (\mathtt{id}, [\![\cdot]\!]))$.

In the following, we define a notion of integrity based on security requirements over physical components. We also discuss the attacker models we consider in this work, and a method to systematically generate attacks.

5.4.2 Physical Integrity Requirements

We use *physical integrity requirements* to determine when the behaviour of the process $\vec{x}$ has deviated critically, and becomes unsafe, independently of whether the system is under attack or not. Formally, physical integrity requirements model *safety properties* over the physical state of the CPS.

Definition 5.4.3 (Physical Integrity Requirement)**.** Let Π be the set of states of the CPS. A *physical integrity requirement* is a finite state automaton $\mathcal{P} = (S, \delta, F, s_0)$, where S is a set, $\delta \colon S \times \vec{X} \to S$ is the transition function, $F \subseteq S$ marks critical states, and $s_0 \in S$ is the initial state. Given a possibly infinite sequence of states of the CPS $\omega = (\pi_0, \pi_1 \ldots)$, we say that ω *fails the physical integrity requirement* $\mathcal{P}$ if and only if there exists a finite prefix of ω, say $(\pi_0, \ldots \pi_n)$, such that the sequence $(\pi_0[\vec{x}], \ldots, \pi_n[\vec{x}])$ is rejected by $\mathcal{P}$. Henceforth, we refer to physical integrity requirements simply as *physical requirements*.

The *behaviours of the CPS* are the infinite sequences $(\pi_0, \pi_1, \ldots)$ where $\pi_{k+1} = [\![\pi_k]\!]$ and π_0 is an initial state. The CPS *satisfies* $\mathcal{P}$ iff all its behaviours satisfy $\mathcal{P}$. We expect the attack-free CPS to satisfy all physical requirements.

5.4.3 Attacker Models

Clarkson and Schneider state in [**QuantitativeIntegrity**] that they know of no widely accepted definition of integrity, but they remark that an "informal definition seems to be the 'prevention unauthorized modification of information'." Clarkson and Schneider use two formal notions to characterise and quantify *corruption*, i.e., the damage to integrity. One notion is *contamination*, a generalisation of taint analysis that tracks information flows from untrusted inputs to outputs that are supposed to be trusted. The other notion is *suppression*, which occurs when implementation outputs omit information about correct outputs with respect to the specification. Contamination is dual to information leakage under the Biba duality; however, there is no apparent Biba dual to suppression [**BibaIntegrity**]. In this work, we want to test whether any corruption generated by contamination of information affects the integrity of the behaviour of the system. To that end, we give attackers the ability to either: 1) alter the value of the coordinates they control by means of arbitrary substitution, or 2) leave the current value be. We only impose a minimal type-checking restriction: the input provided by the attacker must be in the range of the corrupted coordinate, or the behaviour of the system becomes ill-defined. We allow attackers to have some view over the current state of the system, so they can decide when and to corrupt the components they control. Attackers that contaminate information using other methods besides arbitrary substitutions are beyond the scope of this work, e.g., attackers that contaminate data by adding noise (although they are compatible with LBA, since these attacks can be described as spatial transformations).

Following the methodology of LBA, we aim to compose the single cycle semantics function $[\![\cdot]\!] \colon \Pi \to \Pi$ from Definition **??** with a spatial transformation $m \colon \Pi \to \Pi$ (which models some attack) to reveal the transition function $[\![\cdot]\!] \circ m \colon \Pi \to \Pi$ of a latent system which models the CPS under the attack m. In the following, we show how to describe attacks as both vectors and spatial transformations, and we show how to systematically generate them from an attacker model.

Attacker Models

We now show how an attacker model –which describes how an attacker may interact with the system and when– generates a set of attacks. Each attacker model is composed by a *basis* and a set of *coefficients*. The *basis* reflects the visibility of the attacker over the current state of the system, helping them decide "when" to attack. The *coefficients* determine the actions that the attacker uses when interacting with the system, and they define the "how" an attacker attacks.

Definition 5.4.4 (Attack Basis)**.** Let Π be the set of states of the CPS. An *attack basis* is a *finite* partition $\sum_{i=1}^{n} \Pi_i$ of Π; i.e., for $1 \leq i, j \leq n$, Π_i is non-empty, Π_i and Π_j are pairwise disjoint, and the union of all Π_i is Π. We often refer to an element of the partition Π_i by its characteristic predicate; i.e., we write $\Pi_i(\pi)$ instead of $\pi \in \Pi_i$.

The most granular base is given by the individual elements of Π, and it models an attacker with absolute visibility: this attacker always knows the value of every component of the current state, and can act differently at every state if they so wish. The least granular base is given by Π itself, and it models an attacker with no visibility over the current state; this attacker repeats the same action over and over, independently of the current state.

Example 5.1. For the water tank from Section **??**, we define the base $[LL, OK, HH]^T$, where

$$\begin{aligned} LL &\triangleq \{\pi | \pi[\vec{x}[L]] < L_{min}\}, \\ HH &\triangleq \{\pi | \pi[\vec{x}[L]] > L_{max}\}, and \\ OK &\triangleq \{\pi | L_{min} \leq \pi[\vec{x}[L]] \leq L_{max}\}. \end{aligned}$$

The base $[LL, OK, HH]^T$ models the visibility of an attacker that can always determine whether the current state of the system satisfies LL, in OK or in HH. These bases can bypass normal observation assumptions: in this example, the attacker does not observe the value measured by the sensor $\pi[\vec{y}[L]]$; instead, we assume that they observe the real level of the water tank $\pi[\vec{x}[L]]$ by some alternative means.

Informally, the coefficients provided by an attacker model are actions that may affect multiple coordinates simultaneously, and they define how the attacker interacts with the system. We want attackers to be capable of setting any values on the components they control. Unfortunately, we face a typical coverage problem of system testing: many different values exercise the same execution paths, and some paths may be exercised only by very specific inputs. For our examples, we manually choose these

representative values to maximise the coverage of testing, but a selection process for the values that maximise coverage is beyond the scope of this work.

Before we formally define attack coefficients, we introduce a two auxiliary concepts: constant transformations of coordinates and the idempotent monoid generated by combining those constant transformations.

Definition 5.4.5 (Constant Transformations of a Coordinate)**.** Let $c \in \mathbb{C}$ be a coordinate of the CPS whose range is C and let v be a representative value in C; we define the *constant transformation of c to v*, denoted $\Delta_v^c \colon \Pi \to \Pi$, for all $\pi \in \Pi$ and $c' \in \mathbb{C}$ by

$$\Delta_v^c(\pi)[c'] \triangleq \begin{cases} v, & \text{if } c' = c, \\ \pi[c'] & \text{otherwise.} \end{cases} \tag{5.19}$$

Definition 5.4.6 (Idempotent Monoid Generated by Constant Transformations of Coordinates $\mathrm{IM}\langle\Gamma_{\mathcal{K}}\rangle$)**.** Let $\mathcal{K} \subseteq \mathbb{C}$ be a set of coordinates, and let $\Gamma_{\mathcal{K}}$ be a *finite* set of constant transformations of the coordinates in $\mathcal{K}$. We generate the monoid $(\mathrm{IM}\langle\Gamma_{\mathcal{K}}\rangle, \circ, \mathtt{id})$ by closing $\Gamma_{\mathcal{K}}$ under function composition; the unit of the monoid is the identity function $\mathtt{id}$, and the operation is function composition $\circ$. The elements of this monoid are our *attack coefficients*, and they describe the actions that the attacker can perform.

Each transformation $t \in \mathrm{IM}\langle\Gamma_{\mathcal{K}}\rangle$ is finitely generated and unambiguously pairs each coordinate $c \in \mathbb{C}$ with either $\mathtt{id}$ or with some constant transformation Δ_v^c in $\Gamma_{\mathcal{K}}$. We denote this association by $t[c]$. If $t[c] = \mathtt{id}$, then t does not act on the coordinate c, and if $t[c] = \Delta_v^c$, then $t(\pi)[c] = v$ for all $\pi \in \Pi$.

An attacker model is a pair of a set of constant transformations of coordinates and a basis.

Definition 5.4.7 (Attacker Model)**.** Let $\Gamma_{\mathcal{K}}$ be a set of constant transformations of coordinates for some set of coordinates $\mathcal{K}$, and let $\sum_{i=1}^{n} \Pi_i$ be a basis; the pair $\mathcal{A} = (\Gamma_{\mathcal{K}}, \sum_{i=1}^{n} \Pi_i)$ defines the *attacker model* for attackers that have visibility $\sum_{i=1}^{n} \Pi_i$ over the CPS and that are capable of applying the transformations in $\mathrm{IM}\langle\Gamma_{\mathcal{K}}\rangle$ to the states of the CPS, *but only those transformations*.

The attacks generated by the attacker model $(\Gamma_{\mathcal{K}}, \sum_{i=1}^{n} \Pi_i)$ are the vectors whose coefficients are elements of $\mathrm{IM}\langle\Gamma_{\mathcal{K}}\rangle$ and whose basis is $\sum_{i=1}^{n} \Pi_i$.

Definition 5.4.8 (Attack)**.** Given an attacker model $\mathcal{A} = (\Gamma_{\mathcal{K}}, \sum_{i=1}^{n} \Pi_i)$, an *attack* $\vec{m} \colon \Pi \to \Pi$ *generated by* $\mathcal{A}$ is a linear combination

$$\vec{m} = t_1\Pi_1 + t_2\Pi_2 + \ldots + t_n\Pi_n, \tag{5.20}$$

where $t_1, \ldots, t_n \in \mathrm{IM}\langle\Gamma_{\mathcal{K}}\rangle$ is a choice of n coefficients. The attack $\vec{m}$ implements a function defined for $\pi \in \Pi$ by

$$\vec{m}(\pi) = \begin{cases} t_1(\pi), & \text{if } \pi \in \Pi_1; \\ \ldots & \ldots \\ t_n(\pi), & \text{if } \pi \in \Pi_n. \end{cases} \tag{5.21}$$

For convenience, we write $\vec{m}$ using vectors as follows:

$$\vec{m}(\pi) = \begin{bmatrix} t_1 \\ \vdots \\ t_n \end{bmatrix} \cdot \begin{bmatrix} \Pi_1 \\ \vdots \\ \Pi_n \end{bmatrix}. \tag{5.22}$$

We can scale an attack $\vec{m} \colon \Pi \to \Pi$ by $t \in \mathbb{IM}\langle\Gamma_{\mathcal{K}}\rangle$ to obtain the attack $t\vec{m} \colon \Pi \to \Pi$, defined for $\pi \in \Pi$ by

$$t\vec{m}(\pi) \triangleq \begin{bmatrix} t \circ t_1 \\ \vdots \\ t \circ t_n \end{bmatrix} \cdot \begin{bmatrix} \Pi_1 \\ \vdots \\ \Pi_n \end{bmatrix}. \tag{5.23}$$

The set of all attacks generated by the attacker model $\mathcal{A}$ is $\langle\mathcal{A}\rangle$.

We can also combine attacks –i.e., attack addition– using function composition, which is *not commutative*. The sum of attacks using the same basis corresponds to the composition of their coefficients. However, the sum of attacks with different bases requires a common basis: given attacks $\vec{m}_1$ and $\vec{m}_2$ whose respective bases are $\sum_{i=1}^{n} \Pi_i$ and $\sum_{i=1}^{m} \Pi'_i$, we define the common basis $(\sum_{i=1}^{n} \Pi_i)(\sum_{i=1}^{m} \Pi'_i)$ by

$$\left(\sum_{i=1}^{m} \Pi'_i\right)\left(\sum_{i=1}^{n} \Pi_i\right) \triangleq \sum_{j=1}^{m}\sum_{i=1}^{n}(\Pi'_j \cap \Pi_i); \tag{5.24}$$

so the sum of $\vec{m}_2$ and $\vec{m}_1$ is defined by

$$(\vec{m}_2 + \vec{m}_1)(\pi) \triangleq (\vec{m}_2 \circ \vec{m}_1)(\pi) \triangleq \begin{bmatrix} t'_1 \circ t_1 \\ \vdots \\ t'_1 \circ t_n \\ \vdots \\ t'_m \circ t_1 \\ \vdots \\ t'_m \circ t_n \end{bmatrix} \cdot \begin{bmatrix} \Pi'_1 \cap \Pi_1 \\ \vdots \\ \Pi'_1 \cap \Pi_n \\ \vdots \\ \Pi'_m \cap \Pi_1 \\ \vdots \\ \Pi'_m \cap \Pi_n \end{bmatrix}. \tag{5.25}$$

Finally, if the states of the CPS must satisfy a physical invariant $\mathcal{I} \colon \vec{x} \to \mathbb{B}$, then we say that an attack m is *physically impossible* if there exists some state $\pi \in \Pi$ such that $\pi[\vec{x}]$ satisfies $\mathcal{I}$, but $\vec{m}(\pi)[\vec{x}]$ does not. If we do not impose this restriction, then attackers that act on physical components would be able to break the laws of physics. This restriction does not apply for cyber components, but it illustrates that we sometimes need to filter out attacks generated from a model. Since we focus on cyber attackers, we do not exclude any attacks generated by our attacker models.

Example 5.2. In general, the pair $(\Gamma_\emptyset, \Pi)$ models a passive attacker, since the only attack of this model is $\vec{id} = [\texttt{id}] \cdot [\texttt{True}]$; i.e. an attack that does not change the state.

Now, consider the attacker model $(\Gamma_{\vec{y}[L]}, \{LL, OK, HH\})$ in the context of the water tank of Section **??**, where

$$\Gamma_{\vec{y}[L]} = \left\{ \Delta_{500}^{\vec{y}[L]}, \Delta_{800}^{\vec{y}[L]}, \Delta_{1100}^{\vec{y}[L]} \right\}. \tag{5.26}$$

We give the attacker model the representative values 500 L, 800 L and 1100 L for $\bar{y}[L]$ because each of these values triggers different paths in the controller: 500 L activates the paths where the water level is low, 800 L activates those paths where the water level is normal, and finally the value 1100 L covers the paths where the water level is high.

This attacker model generates attacks that affect $\vec{y}[L]$ depending on whether the state satisfies LL, OK or HH; however, their effects over $\vec{y}[L]$ are limited to $\text{IM}\langle\Gamma_{\vec{y}[L]}\rangle = \left\{ \texttt{id}, \Delta_{500}^{\vec{y}[L]}, \Delta_{800}^{\vec{y}[L]}, \Delta_{1100}^{\vec{y}[L]} \right\}$. Each attack is a combination of the basis $[LL, OK, HH]^T$ and the coefficients in $\text{IM}\langle\Gamma_{\vec{y}[L]}\rangle$, so this attacker model generates 64 attacks, given by the combination $\left|\text{IM}\langle\Gamma_{\vec{y}[L]}\rangle\right|^{|\{LL,OK,HH\}|} = 4^3$. For example, we define $\vec{m}$ by

$$\vec{m} = \begin{bmatrix} \texttt{id} \\ \Delta_{500} \\ \Delta_{500} \end{bmatrix} \cdot \begin{bmatrix} LL \\ OK \\ HH \end{bmatrix}. \tag{5.27}$$

The transformation $\vec{m}$ describes an attack where the attacker forces the value of $\vec{y}[L]$ to be 500 L, but only when the current state of the CPS satisfies either OK or HH; otherwise, the attacker preserves the current value of $\vec{y}[L]$. This attack causes the tank to overflow after about 876 cycles because the controller always believes that the level of water of the tank is low, and it continuously tries to fill the tank.

5.4.4 Required Capabilities

The inclusion of the identity transformation as an action available to all attacker models implies that attacker models generate attacks that do not use all their altering capabilities. In general, for a given spatial transformation m, the *required capabilities* to execute m is the minimal set of changes in coordinates that an attacker needs to perform to apply m in any state of the CPS.

Definition 5.4.9 (Required Capabilities)**.** Let $m \colon \Pi \to \Pi$; we define the *capabilities required to execute* m, denoted $|m|$, to be the set

$$|m| \triangleq \left\{ \Delta_{m(\pi)[c]}^{c} \mid \pi[c] \neq m(\pi)[c], \text{ for } \pi \in \Pi \text{ and } c \in \mathbb{C}. \right\} \tag{5.28}$$

In general, computing the required capabilities of an arbitrary transformation $m \colon \Pi \to \Pi$ is a hard problem, since it requires us to apply m to all elements of Π to check which components change. However, when we generate spatial transformations using attacker models, we can compute their required capabilities directly from their definition, as shown in the following Corollaries **??** and **??**.

Corollary 5.1. Since every $t \in \mathrm{IM}\langle\Gamma_{\mathcal{K}}\rangle$ pairs each component c in $\mathbb{C}$ to some Δ_v^c or to `id`, the set of required capabilities to execute t is

$$|t| = \{t[c] \mid t[c] \neq \texttt{id} \text{ and } t[c] = \Delta_v^c, \text{for } c \in \mathbb{C}\}\,. \tag{5.29}$$

Corollary 5.2. If $\vec{m} = [t_1 \dots t_n]^T \cdot [\Pi_1, \dots, \Pi_n]^T$, then the set of capabilities required for the attack $\vec{m}$ is the union of the capabilities required for its coefficients. Formally,

$$|\vec{m}| = \bigcup_{i=1}^{n} |t_i|. \tag{5.30}$$

The attacker with the minimal capabilities that can execute an attack $\vec{m}$ is the *free attacker model of* $\vec{m}$.

Definition 5.4.10 (Free Attacker Model)**.** For any attack $\vec{m}$ defined over some base $\sum_{i=1}^{n} \Pi_i$, the pair $(|\vec{m}|, \sum_{i=1}^{n} \Pi_i)$ is the *free attacker model of* $\vec{m}$, and it is the attacker model with minimal capabilities that can execute $\vec{m}$.

5.5 Improving Robustness via Latent Behaviour Analysis

When we compose the single-cycle semantics $[\![\cdot]\!]$ with an arbitrary state transformation $m\colon \Pi \to \Pi$, we reveal a transition function $[\![\cdot]\!] \circ m\colon \Pi \to \Pi$. We call $[\![\cdot]\!] \circ m$ the *latent dynamics* revealed by m, and we use $[\![\cdot]\!] \circ m$ to compute *latent behaviours* revealed by m.

Definition 5.5.1 (Latent Behaviours of a Closed CPS)**.** Given an attack $\vec{m}\colon \Pi \to \Pi$, the *latent behaviour revealed by* $\vec{m}$ at some initial state π_0 is the infinite sequence of states $(\pi_0^{\vec{m}}, \pi_1^{\vec{m}} \dots)$ inductively defined, for $k \geq 0$, by

$$\pi_{k+1}^{\vec{m}} \triangleq [\![\vec{m}(\pi_k^{\vec{m}})]\!], \quad \text{with } \pi_0^{\vec{m}} \triangleq \pi_0. \tag{5.31}$$

5.5.1 Robustness

When a system is under attack, the single-cycle semantics changes, and the system reveals latent behaviours. Given an attacker model $\mathcal{A}$, a set of requirements $\mathbb{P}$, and a CPS $(\Pi, [\![\cdot]\!])$, we assume that the CPS satisfies all the requirements in $\mathbb{P}$ (see Definition **??**). Given an attack $\vec{m}\colon \Pi \to \Pi$ generated by the attacker model $\mathcal{A}$, and a requirement $\mathcal{P} \in \mathbb{P}$, LBA states that if $(\Pi, [\![\cdot]\!] \circ m)$ fails $\mathcal{P}$, then $\vec{m}$ *breaks* $\mathcal{P}$ in $(\Pi, [\![\cdot]\!])$, so the CPS is vulnerable to $\mathcal{A}$. We define the *robustness* of the system with respect to a given attacker model to be the fraction of attacks generated by the attacker model that fail to break any physical requirement.

Definition 5.5.2 (Robustness)**.** Given an attacker model $\mathcal{A}$, and a set of requirements $\mathbb{P}$, let $\mathcal{A}(\mathbb{P})$ be the set of attacks generated $\mathcal{A}$ that break at least one requirement in $\mathbb{P}$. We define the *robustness* of the system by

$$\frac{|\langle\mathcal{A}\rangle| - |\mathcal{A}(\mathbb{P})|}{|\langle\mathcal{A}\rangle|}$$

Example 5.3. In the context of the attacker model of the water tank from Section **??** where the basis is $[LL, OK, HH]^T$ and the coefficients are $\Gamma_{\vec{y}[L]} = \{\Delta_{500}, \Delta_{800}, \Delta_{1100}\}$, there are 64 attacks. We confirm via simulation of the revealed latent systems that 30 of those attacks break at least one physical requirement, so the robustness of the water tank with respect to this attacker model is 34/64.

5.5.2 Counterattacks and Latent Robustness

LBA uses spatial transformations to alter the single-cycle semantics of the CPS, but it is possible for us to keep transforming the revealed latent behaviours with other spatial transformations: *counterattacks*. In the LBA framework, counterattacks and attacks only differ only by intent: we use attacks to reveal compromised systems, and we use counterattacks to reveal systems where attackers and counterattackers are interacting. Since attacks and counterattacks are both transformations of the system, applying a counterattack in the absence of attacks is equivalent to attacking the system using the counterattack.

Counterattacker models are also dual to attacker models. A counterattacker model is also a pair of a set of constant transformations of coordinates and a basis that partitions the state space. However, we assume that that the controller or the operator are going to play the role of the counterattacker, so the coordinates that they affect should be specially impactful with respect to the behaviour of the system. More precisely, we believe it is sensible to describe the counterattacker model using transformations that directly affect the output of the controller or the physical components the system.

Example 5.4. In the context of the water tank from Section **??**, we define the counterattacker model $\left(\Gamma_{\{\vec{x}[MV],\vec{x}[P]\}}, \left\{\widehat{LL}, \widehat{OK}, \widehat{HH}\right\}\right)$. The actions of the counterattacker are generated by $\Gamma_{\{\vec{x}[MV],\vec{x}[P]\}}$, where

$$\begin{aligned}\Gamma_{\{\vec{x}[MV],\vec{x}[P]\}} &= \Gamma_{\vec{x}[MV]} \cup \Gamma_{\vec{x}[P]} \\ &= \left\{\Delta^{\vec{x}[MV]}_{\texttt{open}}, \Delta^{\vec{x}[MV]}_{\texttt{closed}}\right\} \cup \left\{\Delta^{\vec{x}[P]}_{\texttt{on}}, \Delta^{\vec{x}[P]}_{\texttt{off}}\right\};\end{aligned}$$

i.e., the counterattacker can change the physical state of the motor valve MV and the pump P by, e.g., manual operation. Since this counterattacker acts directly on the physical state of the components, it is a rather influential force over the behaviour of the CPS.

The basis $\left\{\widehat{LL}, \widehat{OK}, \widehat{HH}\right\}$ is defined for $\pi \in \Pi$ by

$$\begin{aligned}\widehat{LL}(\pi) &\triangleq (\pi[\vec{y}[L]] < L_{min}), \\ \widehat{HH}(\pi) &\triangleq (\pi[\vec{y}[L]] > L_{max}), and \\ \widehat{OK}(\pi) &\triangleq (L_{min} \leq \pi[\vec{y}[L]] \leq L_{max}).\end{aligned}$$

This counterattacker model generates 729 different counterattacks, since

$$\left|\text{IM}\langle\Gamma_{\{\vec{x}[MV],\vec{x}[P]\}}\rangle\right|^{\left|\{\widehat{LL},\widehat{OK},\widehat{HH}\}\right|} = 9^3 = 729$$

Note that the visibility of this counterattacker is controlled by attacker from Example **??**. More precisely, when deciding $\widehat{LL}, \widehat{OK}$ and $\widehat{HH}$, the counterattacker uses the value of coordinate $\vec{y}[L]$ (which is controlled by this attacker), and not on the physical (real) water level $\vec{x}[L]$. These non-trivial dependencies result in curious interactions: the counterattack $\vec{w}$ defined by

$$\vec{w} = \begin{bmatrix} \Delta^{\vec{x}[MV]}_{\texttt{closed}} \circ \Delta^{\vec{x}[P]}_{\texttt{off}} \\ \texttt{id} \\ \texttt{id} \end{bmatrix} \cdot \begin{bmatrix} \widehat{LL} \\ \widehat{OK} \\ \widehat{HH} \end{bmatrix}. \tag{5.32}$$

counters the attack $\vec{m}$ from Example **??**, because the attack $\vec{m}$ tries to trick the controller into thinking that the water level of the tank is low, and this counterattack closes the motor valve and shuts off the pump. This takes the system into a state where water does not flow in or out of the tank, so the attack $\vec{m}$ cannot overflow it.

We now provide a formal notion of what it means for an attack to be countered by a counterattack.

Definition 5.5.3 (Attack Countering)**.** We say that a counterattack $\vec{w}$ *counters* an attack $\vec{m}$ if and only if the latent behaviour $[\![\cdot]\!] \circ \vec{m}$ fails at least one physical requirement, but $[\![\cdot]\!] \circ (\vec{w} \circ \vec{m})$ does not fail any.

The *latent robustness* of a system is characterised by the set of attacks that fail to break any requirement, plus the set of attacks that can be countered.

Definition 5.5.4 (Latent Robustness)**.** Given an attacker model $\mathcal{A}$, a counterattacker model $\mathcal{B}$, and a set of physical requirements $\mathbb{P}$, we define $\mathcal{B}(\mathcal{A}(\mathbb{P}))$ to be the set of attacks in $\mathcal{A}$ that *cannot be countered* by counterattacks in $\mathcal{B}$. We define the *latent robustness* of the system by

$$\frac{|\langle\mathcal{A}\rangle| - |\mathcal{B}(\mathcal{A}(\mathbb{P}))|}{|\langle\mathcal{A}\rangle|}$$

Example 5.5. In the context of the attacker model of the water tank from Section **??**, the latent robustness of the water tank with respect of the attacker model from Example **??** and the counterattacker model from Example **??** is 64/64, since every attack in the attacker model can be countered by a counterattack in the counterattacker model.

5.5.3 Improving Robustness with Repairs

If a counterattack $\vec{w}$ counters an attack $\vec{m}$, then we can, in theory, improve the robustness of the CPS by triggering $\vec{w}$ when $\vec{m}$ is affecting the system, and only then. This method has the following two assumptions: one, there exists some sort of monitor that can always correctly identify which attack is currently affecting the system (if any), and two, the counterattack can be properly launched; i.e., the attacker does not interfere with the counterattacking mechanism. While condition two can be enforced by isolating the attacker and the counterattacker models such that they do not share coordinates, the first condition may be practically impossible to satisfy. Thus, we consider an alternative method: changing the program of the controller to incorporate the effects

of counterattacks in the system, guarded by conditions that signal abnormal behaviour, which we obtain from the data resulting from LBA.

Changing the program of the controller is not a fail-proof process to improve robustness. We study the distribution of attacks and counterattacks that we obtain from LBA and dig for relations between them, and we can see patterns that guide us in the process of changing the controller. To confirm that our changes are a repair, we compare the new system to the original system using their robustness.

Example 5.6. To increase the robustness of the CPS from Section **??** with respect to the attacker model from the same example, we change the program of the controller to replicate the actions taken by the counterattack from Example **??** and the conditions that trigger them. The repair consists of shutting off P and closing MV when the level of water is low. This change improves the robustness of the system from $34/64$ to $58/64$, so it classifies as a repair. The improvement is due to a forced stabilisation of the system: once the level of water goes below L_{min}, the repair prevents any more water from flowing in or out of the tank. If the specification of the CPS required the level of the tank to constantly go from L_{min} to L_{max} and vice-versa, this repair would not be acceptable since it would break a functional requirement.

5.5.4 Choosing Attacker and Counterattacker Models

The choice of attacker and counterattacker models defines the boundaries of the LBA. If the attacker model is too complex, then the resulting set of generated attacks is not only large, but several of its attacks might not have a counterattack. Similarly, if the counterattacker model is not very capable, then LBA will conclude that many attacks cannot be countered. In this section, we provide small hints for the selection of attackers and counterattackers, based on partial orders that exist within the attacker models.

From Chapter **??**, we know that we can compare attacker models by both their capabilities and by the set of physical requirements that they break; i.e. their power. As expected, these partial orders are tightly related.

Definition 5.5.5 (Capabilities and Power)**.** Given two attacker models $\mathcal{A}_1 = (\Gamma_1, \sum \Pi_1)$ and $\mathcal{A}_1 = (\Gamma_2, \sum \Pi_2)$, we say that $\mathcal{A}_1$ is *less or equally capable* than $\mathcal{A}_2$, denoted $\mathcal{A}_1 \subseteq \mathcal{A}_2$, iff $\langle \mathcal{A}_1 \rangle \subseteq \langle \mathcal{A}_2 \rangle$.

Now, given a set of physical requirements $\mathbb{P}$, we denote the set of requirements that an attacker model $\mathcal{A}$ can break by $\mathbb{P}|_{\mathcal{A}}$. Formally, for a requirement $\mathcal{P} \in \mathbb{P}$, $\mathcal{P} \in \mathbb{P}|_{\mathcal{A}}$ if and only if there exists an attack in $\mathcal{A}(\mathbb{P})$ that breaks $\mathcal{P}$. We say that $\mathcal{A}_1$ is *less or equally powerful* than $\mathcal{A}_2$, denoted $\mathcal{A}_1 \leq \mathcal{A}_2$, iff $\mathbb{P}|_{\mathcal{A}_1} \subseteq \mathbb{P}|_{\mathcal{A}_2}$.

We highlight some relevant corollaries.

Corollary 5.3 (Monotonicity↑)**.** For all attacker models $\mathcal{A}_1$ and $\mathcal{A}_2$, if $\mathcal{A}_1 \subseteq \mathcal{A}_2$, then $\mathcal{A}_1 \leq \mathcal{A}_2$.

Proof. Every attack generated by $\mathcal{A}_1$ is an attack generated by $\mathcal{A}_2$, so every requirement that $\mathcal{A}_1$ can break, $\mathcal{A}_2$ can break using the same attacks as $\mathcal{A}_1$. □

Corollary **??** is practical because it motivates us to first run LBA with small attacker models. If we prove that the system is vulnerable with respect to an attacker model $\mathcal{A}$ that only modifies one coordinate, then the system is vulnerable with respect to any attacker model that is more capable than $\mathcal{A}$.

Corollary 5.4 (Monotonicity↓)**.** For all attacker models $\mathcal{A}_1$ and $\mathcal{A}_2$ with $\mathcal{A}_1 \subseteq \mathcal{A}_2$, if $\mathcal{A}_2$ cannot break a requirement $\mathcal{P}$, then $\mathcal{A}_1$ cannot either.

Proof. If $\mathcal{P} \notin \mathcal{A}_2(\mathbb{P})$, then there is no attack $\vec{m}$ generated by $\mathcal{A}_2$ that can reveal a latent behaviour that fails $\mathcal{P}$; since $\mathcal{A}_1 \subseteq \mathcal{A}_2$, the universe of attacks of $\mathcal{A}_1$ is included in the universe of attacks of $\mathcal{A}_2$, so an attack that breaks $\mathcal{P}$ cannot exist for $\mathcal{A}_1$, meaning that $\mathcal{P} \notin \mathcal{A}_1(\mathbb{P})$. □

Corollary **??** is also practical. If we have an intuition about the robustness of the system, and we characterise an attacker model $\mathcal{A}$ such that the robustness of the system with respect to $\mathcal{A}$ is high, then there is no need to check any attacker model that is less capable than $\mathcal{A}$ since the robustness will not decrease for those attackers.

Corollary 5.5 (Unsafe CPS)**.** If the trivial attacker model $(\emptyset, \Pi)$ is capable of breaking some requirement $\mathcal{P}$, then every attacker model is capable of breaking $\mathcal{P}$ by doing nothing (due to monotonicity); in other words, the CPS is unsafe.

Proof. The transformation $\mathtt{id}$, which maps every state to itself, is the only attack generated by the attacker model $(\emptyset, \Pi)$. Now, since the original CPS $(\Pi, [\![\cdot]\!])$ is equal to $(\Pi, [\![\cdot]\!] \circ \mathtt{id})$, if $(\Pi, [\![\cdot]\!] \circ \mathtt{id})$ fails any requirement $\mathcal{P}$, then the original system also fails $\mathcal{P}$. □

Finally, Corollary **??** is a special case of Corollary **??**, where the least capable attacker model (i.e., the attacker that does nothing) breaks one or more physical requirements.

We now illustrate how quantifying the robustness of CPS designs can help the process of redesigning the controller of a CPS by means of a simple, but not trivial, case study.

5.6 Case Study

In this section, we illustrate the process of improving the design of a composition of CPSs by using robustness as the evaluation measure which we obtain via LBA and testing.

5.6.1 SWaT – Secure Water Treatment Plat

SWaT [**SWat**] is a water treatment plant for security research. SWaT is a six-stage testbed that combines modern real-world filtration techniques to treat water for human consumption.

In a nutshell, the six-stage system operates as follows: (1) Stage1 and Stage2, the system takes raw water, and through pH, conductivity, and Oxidation-reduction potential analysers (ORP) determine how much chemicals (HCl, NaOCl, NaCl) add to the liquid.

(2) Stage3 and Stage4 filter the water using an Ultrafiltration system and de-chlorinates the water through the usage of UV lamps. (3) Stage5 and Stage6 feed the Reverse Osmosis (RO) system with filtered water. Water from the RO system cleans the membranes in the Ultrafiltration system. A set of interconnected PLCs control each stage. They read the system's physical properties and issue commands to influence the plant in a controlled manner. Operators monitor the whole system via a Supervisory Control and Data Acquisition (SCADA) system.

Figure **??** shows a diagram of the first three stages of SWaT and how they interact. Each stage has a pair of actuators: a motor valve MV to regulate the liquid in the tank, and a pump P that drains water out of each tank. In this work, we focus on the water flow and storage subprocess, i.e., Stages 2 and 3. Stage2 stores chemical–treated water and Stage3 stores the water after filtration. The security requirements state that the level of water in the two tanks, denoted by L2 and L3 respectively, must remain within safe ranges (i.e., the tanks should never be empty nor overflow).

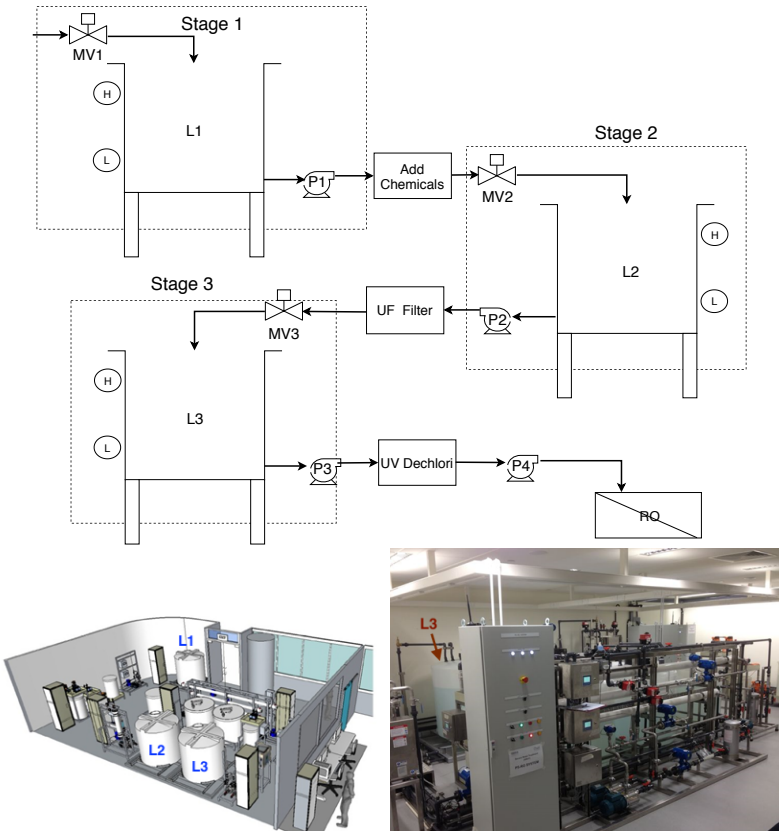


FIGURE 5.4: SWaT diagram

5.6.2 SWaT Stage 3

Recall the motivational example presented in Section **??** that models a water tank. That water tank is the Stage 3 submodule of SWaT. We reuse the dynamics presented in Section **??** (which we henceforth refer to as Stage 3). Since there are also pumps and

valves in other processes, we rename the components MV, L, and P from Section **??** to $MV3$, $L3$ and $P3$ respectively, matching the descriptions in Figure **??**.

Stage 3 receives water from Stage 2, and the controller of Stage 3 reports to the controller of Stage 2 the following information: the presumed level of water in the tank $T3$ and the presumed state of the valve $MV3$. To model the composition of Stage 3 with Stage 2, we define the behaviour of the communication channels $\vec{a}$, and $\vec{w}$ for Stage 2, and $\vec{b}$ and $\vec{v}$ for Stage 3. Let

$$\vec{b}[L3](k+1) \triangleq \vec{q}[L3](k) \tag{5.33}$$

$$\vec{b}[MV3](k+1) \triangleq \vec{q}[MV3](k) \tag{5.34}$$

$$\vec{w}[L2](k) \triangleq \vec{v}[L2](k), \tag{5.35}$$

$$Inflow(k) = \begin{cases} 0 & \text{if } \vec{w}[L2](k) = 0 \\ 0.31, & \text{if } \vec{x}[MV](k) = \texttt{open} \\ 0, & \text{if } \vec{x}[MV](k) = \texttt{closed} \\ 0.26, & \text{otherwise} \end{cases} \tag{5.36}$$

where $\vec{v}[L2](k)$ is part of the behaviour description of Stage 2. The inflow of water for $T3$ now depends on the physical state of $T2$; if there is no water in $T2$, there is no inflow.

In the following, we formally model Stage 2 and describe its composition with Stage 3. We then characterise an attacker model and a counterattacker model to measure the robustness of the composition. Finally, we use the results from the LBA to illustrate two changes to the composition: a repair that improves robustness, and a non-repair that does not.

5.6.3 A Formal Model of Stage 2

We provide a model of SWaT Stage 2. This model has three physical components $MV2$, $L2$ and $P2$, and uses the constants $L2min = 800$, $L2max = 1000$ and $T2 = 7$.

The Controller PLC2. This controller has eight modes which depend on the presumed states of the pump $P2$ and the valve $MV2$, and a timer. The vector $\vec{q}$ has three more coordinates $P2$, $MV2$, and $\tau 2$, and their respective ranges are

$$\begin{aligned} \vec{q}[P2] &\in \{\texttt{off}, \texttt{on}\}, \\ \vec{q}[MV2] &\in \{\texttt{closed}, \texttt{opening}, \texttt{open}, \texttt{closing}\}, \\ \vec{q}[\tau 2] &\in \mathbb{N} \end{aligned}$$

Through this section, we assume that every actuator and sensor simply translates values across the cyber and physical domains, i.e., $\vec{i}(k) = \theta_{\vec{Y}}(\vec{y}(k)) = \vec{y}(k)$ and $\vec{u}(k) = \theta_{\vec{O}}(\vec{o}(k)) = \vec{o}(k)$. The controller PLC2 receives a network input $\vec{a}$ and a sensor reading $\vec{y}$, and produces a control command $\vec{u}$. The network input $\vec{a}$ has two coordinates $L3$

and $MV3$, whose ranges are

$$\begin{aligned}\vec{a}[L3] &\in [0..1200],\\ \vec{a}[MV] &\in \{\texttt{closed}, \texttt{opening}, \texttt{open}, \texttt{closing}\}.\end{aligned}$$

The sensor reading $\vec{y}$ has one coordinate $L2$, where $\vec{y}[L2] \in [0..1200]$. The control command $\vec{u}$ has two coordinates, $MV2$ and $L2$, where

$$\begin{aligned}\vec{u}[P2] &\in \{\Delta_{\texttt{off}}, \Delta_{\texttt{on}}, \texttt{id}\},\\ \vec{u}[MV2] &\in \{\texttt{closed}, \texttt{opening}, \Delta_{\texttt{open}}, \Delta_{\texttt{closing}}, \texttt{id}\}.\end{aligned}$$

Let $L3_k = \vec{a}[L3](k)$, $MV3_k = \vec{a}[MV3](k)$, $MV2_k = \vec{q}[MV2](k)$, $P2_k = \vec{q}[P2](k)$, and $L2_k = \vec{y}[L2](k)$; we define the behaviour of the controller PLC2 as follows:

$$\vec{q}[P2](k+1) \triangleq \begin{cases}\texttt{off}, & \text{if } L3_k \leq L3min \text{ or } MV3_k \in \{\texttt{closed}, \texttt{closing}\},\\ \texttt{on}, & \text{if } L3_k \geq L3min \text{ or } MV3_k \in \{\texttt{open}, \texttt{opening}\}\\ P2_k & \text{otherwise;}\end{cases} \tag{5.37}$$

$$\vec{q}[MV2](k+1) \triangleq \begin{cases}\texttt{opening}, & \text{if } MV2_k = \texttt{closed} \text{ and } L2_k \leq L2min,\\ \texttt{open}, & \text{if } MV2_k = \texttt{opening} \text{ and } \tau 2_k > T_2,\\ \texttt{closing}, & \text{if } MV2_k = \texttt{open} \text{ and } L2_k \geq L2max,\\ \texttt{closed}, & \text{if } MV2_k = \texttt{closing} \text{ and } \tau 2_k > T_2,\\ MV2 & \text{otherwise;}\end{cases} \tag{5.38}$$

$$\vec{q}[\tau 2](k+1) \triangleq \begin{cases}\tau 2_k + 1 & \text{if } MV2_k \in \{\texttt{opening}, \texttt{closing}\}\\ 0 & \text{otherwise;}\end{cases} \tag{5.39}$$

$$\vec{u}[MV2](k+1) \triangleq \Delta_{MV2_k}, \tag{5.40}$$

$$\vec{u}[P2](k+1) \triangleq \Delta_{P2_k}. \tag{5.41}$$

The Process The state of the process $\vec{x}$ has three coordinates: $L2$, $MV2$ and $P2$. Their respective ranges are $\vec{x}[L2] \in [0..1200]$, $\vec{x}[MV2] \in \{\texttt{closed}, \texttt{opening}, \texttt{open}, \texttt{closing}\}$ and $\vec{x}[P2] \in \{\texttt{on}, \texttt{off}\}$.

Let $P2_k = \vec{x}[P2](k)$, $MV2_k = \vec{x}[MV2](k)$, and $L2_k = \vec{x}[L2](k)$, and let $\delta_{P2} = \vec{u}[P2](k)$, and $\delta_{MV2} = \vec{u}[MV2](k)$; we define the behaviour of the process by

$$\begin{aligned}
\vec{x}[P2](k+1) &\triangleq \delta_{P2}(P2_k) \\
\vec{x}[MV2](k+1) &\triangleq \delta_{MV2}(MV2_k) \\
\vec{x}[L2](k+1) &\triangleq L2_k + \partial(MV2_k, P2_k), \text{where} \\
\partial(MV2, P2) &\triangleq \begin{cases}
0 & \text{if } MV2_k = \texttt{closed} \text{ and } P2_k = \texttt{off} \\
0.46 & \text{if } MV2_k = \texttt{open} \text{ and } P2_k = \texttt{off} \\
0.29 & \text{if } MV2_k \in \{\texttt{opening}, \texttt{closing}\} \text{ and } P2_k = \texttt{off} \\
-0.29 & \text{if } MV2_k = \texttt{closed} \text{ and } P2_k = \texttt{on} \\
0.13 & \text{if } MV2_k = \texttt{open} \text{ and } P2_k = \texttt{on} \\
-0.28 & \text{if } MV2_k \in \{\texttt{opening}, \texttt{closing}\} \text{ and } P2_k = \texttt{on}
\end{cases}
\end{aligned}$$

The values for $\partial(MV2, P2)$ were computed by linear regression to approximate the flow equation $L2_{k+1} = L2_k + \texttt{Inflow}_k - \texttt{Outflow}_k$. We leave the initial conditions for $\vec{x}$ abstract for now.

The sensor reading $\vec{y}$ has only one coordinate $L2$, whose range is any positive value. The behaviour of $\vec{y}$ is given by

$$\vec{y}[L2](k) \triangleq \vec{x}[L2] \tag{5.42}$$

Finally, the physical state of the process of Stage 2 propagates to Stage 3 via the abstract channel $\vec{v}[L3]$

$$\vec{v}[L2](k) \triangleq \vec{x}[L2](k). \tag{5.43}$$

We use this relation to enforce the conservation of matter: if there is no water in T2, then no water should be flowing into T3, even if $MV3$ is open and $P2$ is on.

Figure **??** shows the attack-free, steady state behaviour of the water level in tank T2. Due to interdependence, the changes in the behaviour of the tank T2 affect the behaviour of tank T3 and vice versa. This composite CPS has four security requirements that model safe water levels for both tanks: $\Box\vec{x}[L2] > 0$, $\Box\vec{x}[L2] < 1200$, $\Box\vec{x}[L3] > 0$ and $\Box\vec{x}[L3] < 1200$.

5.6.4 Attacker Model

We now define an attacker model using Definitions **??**, **??**, and **??** for the quantification of robustness. We consider an attacker model that manipulates the control outputs of the controller PLC2 for pump $P2$ and valve $MV2$. We choose this attacker because they can override the commands sent by PLC2, giving them the same level of control over Stage 2 than PLC2 has. More precisely, we allow this attacker to set $\vec{u}[MV2](k)$ and $\vec{u}[P2](k)$ for all $0 \leq k \leq t$ given some testing time parameter t, which we set to 6 hours to give the effect of actions of the attacker enough time to propagate through the system. To define the attacker model, we reuse the base from Example **??**. The base

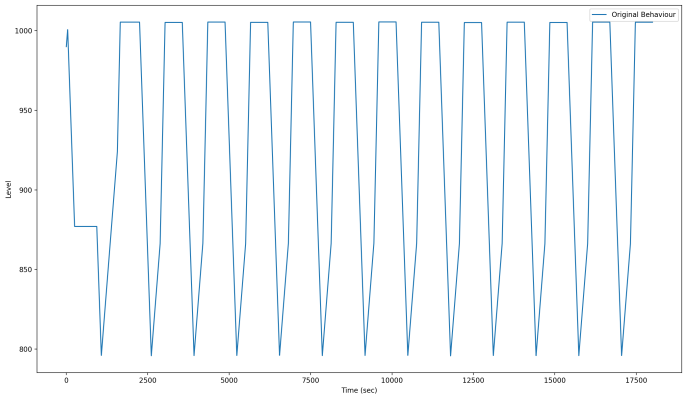


FIGURE 5.5: Normal behaviour of the water level of Stage 2 if Stage 3 is also operating normally.

for the attacker model is $[LL, OK, HH]^T$, where $LL(\pi) = \pi[\vec{x}[L3]] < L3min$, $HH(\pi) = \pi[\vec{x}[L3]] > L3max$ and $OK(\pi) = L3min \leq \pi[\vec{x}[L3]] \leq L3max$, for $\pi \in \Pi$. This implies that even though the attacker modifies components in Stage2, *they only have visibility of the tank T3 and not of T2*. Although this attacker is "blind" with respect to $T2$, their actions will definitely impact the behaviour of $T2$.

We manually choose the representative values used by the attacker model so that we maximise testing coverage. The representative values of $\vec{u}[MV2]$ are `open` and `closed`, which in turn define

$$\Gamma_{\vec{u}[MV2]} = \left\{ \Delta^{\vec{u}[MV2]}_{\texttt{open}}, \Delta^{\vec{u}[MV2]}_{\texttt{closed}} \right\}.$$

The representative values for $\vec{u}[P2]$ are `on` and `off`, which define

$$\Gamma_{\vec{u}[P2]} = \left\{ \Delta^{\vec{u}[P2]}_{\texttt{on}}, \Delta^{\vec{u}[P2]}_{\texttt{off}} \right\}.$$

An attack $\vec{m}$ generated by this model is of the form

$$\vec{m}(\pi) = \begin{bmatrix} t_{LL} \circ s_{LL} \\ t_{OK} \circ s_{OK} \\ t_{HH} \circ s_{HH} \end{bmatrix} \cdot \begin{bmatrix} LL \\ OK \\ HH \end{bmatrix},$$

where $t_i \in \left\{ \texttt{id}, \Delta^{\vec{u}[MV2]}_{\texttt{open}}, \Delta^{\vec{u}[MV2]}_{\texttt{closed}} \right\}$ and $s_i \in \left\{ \texttt{id}, \Delta^{\vec{u}[P2]}_{\texttt{on}}, \Delta^{\vec{u}[P2]}_{\texttt{off}} \right\}$, for $i \in \{LL, OK, HH\}$. The resulting attacker model has 729 attacks because

$$\left| \text{IM}\left\langle \Gamma_{\{\vec{u}[MV2], \vec{u}[P2]\}} \right\rangle \right|^{|\{LL,OK,HH\}|} = 9^3 = 729.$$

5.6.5 Counterattacker Model

We want the counterattacker of this system to be "in the controller PLC3"; i.e. we want the counterattacker to have control over $\vec{u}[MV3](k)$ and $\vec{u}[P3](k)$ for all $0 \leq k \leq t$, with $t = 6$ hours to let the actions of the attacker and counterattacker enough time to interact and reveal either stable safe behaviours of a broken requirement. We define the base for the counterattacker model to be $[LL', OK', HH']^T$, where $LL'(\pi) = \pi[\vec{y}[L3]] < L3min$, $HH'(\pi) = \pi[\vec{y}[L3]] > L3max$ and $OK'(\pi) = L3min \leq \pi[\vec{y}[L3]] \leq L3max$, for $\pi \in \Pi$ (unlike the base of the attacker, which uses $\vec{x}[L3]$ instead of $\vec{y}[L3]$).

We set the representative values of $\vec{u}[MV3]$ to be `open`, and `closed` and the representative values of $\vec{u}[P3]$ to be `on`, and `off`; more precisely,

$$\Gamma_{\vec{u}[MV3]} = \left\{\Delta^{\vec{u}[MV3]}_{\texttt{open}}, \Delta^{\vec{u}[MV3]}_{\texttt{closed}}\right\}.$$
$$\Gamma_{\vec{u}[P3]} = \left\{\Delta^{\vec{u}[P3]}_{\texttt{on}}, \Delta^{\vec{u}[P3]}_{\texttt{off}}\right\}.$$

The counterattacks of this model $\vec{w}$ are then of the form

$$\vec{w}(\pi) = \begin{bmatrix} t_{LL'} \circ s_{LL'} \\ t_{OK'} \circ s_{OK'} \\ t_{HH'} \circ s_{HH'} \end{bmatrix} \cdot \begin{bmatrix} LL' \\ OK' \\ HH' \end{bmatrix},$$

where $t_i \in \left\{\texttt{id}, \Delta^{\vec{u}[MV3]}_{\texttt{open}}, \Delta^{\vec{u}[MV3]}_{\texttt{closed}}\right\}$ and $s_i \in \left\{\texttt{id}, \Delta^{\vec{u}[P3]}_{\texttt{on}}, \Delta^{\vec{u}[P3]}_{\texttt{off}}\right\}$, for $i \in \{LL', OK', LL'\}$. This counterattacker model has 729 counterattacks.

5.6.6 Quantification of Robustness: Analysis of Results

Using LBA, we quantify the robustness of the system given the initial conditions $\vec{x}[L2] = 990$, $\vec{x}[MV2] = open$, $\vec{x}[P2] = on$, $\vec{x}[L3] = 920$, $\vec{x}[MV3] = open$, $\vec{x}[P3] = on$. The analysis computes a robustness factor of 162/729, meaning that are successful 567 attacks for the given attacker model. All attacks that break $\Box\vec{x}[L3] > 0$ or $\Box\vec{x}[L3] < 1200$ can always be countered. However, 136 attacks cannot be countered by the proposed counterattacker model, meaning that the latent robustness of the system is only $593/729$. The attacks without a counterattack always break either $\Box\vec{x}[L2] > 0$ or $\Box\vec{x}[L2] < 1200$, but some attacks that break those requirements can be countered. For example, the attack

$$\vec{m}(\pi) = \begin{bmatrix} \Delta^{\vec{u}[MV2]}_{\texttt{open}} \circ \Delta^{\vec{u}[P2]}_{\texttt{off}} \\ \texttt{id} \circ \texttt{id} \\ \texttt{id} \circ \texttt{id} \end{bmatrix} \cdot \begin{bmatrix} \pi[\vec{x}[L3]] < L3min \\ L3min \leq \pi[\vec{x}[L3]] \leq L3max \\ \pi[\vec{x}[L3]] > L3max \end{bmatrix},$$

breaks the requirement $\Box\vec{x}[L2] < 1200$, but it is countered by the counterattack

$$\vec{w}(\pi) = \begin{bmatrix} \texttt{id} \circ \texttt{id} \\ \texttt{id} \circ \texttt{id} \\ \texttt{id} \circ \Delta^{\vec{u}[P3]}_{\texttt{off}} \end{bmatrix} \cdot \begin{bmatrix} \pi[\vec{y}[L3]] < L3min \\ L3min \leq \pi[\vec{y}[L3]] \leq L3max \\ \pi[\vec{y}[L3]] > L3max \end{bmatrix}.$$

The counterattack works by preventing the conditions that trigger the attack; more precisely, the attack activates when $\pi[\vec{x}[L3]] < L3min$, and the counterattack $\vec{w}$ prevents such condition from happening (although the behaviour eventually causes the system to reach a state where water does not flow in or out of the tanks).

The following attack

$$\vec{m}'(\pi) = \begin{bmatrix} \mathtt{id} \circ \mathtt{id} \\ \Delta^{\vec{u}[MV2]}_{\mathtt{open}} \circ \Delta^{\vec{u}[P2]}_{\mathtt{off}} \\ \mathtt{id} \circ \Delta^{\vec{u}[P2]}_{\mathtt{off}} \end{bmatrix} \cdot \begin{bmatrix} \pi[\vec{x}[L3]] < L3min \\ L3min \leq \pi[\vec{x}[L3]] \leq L3max \\ \pi[\vec{x}[L3]] > L3max \end{bmatrix}$$

causes tank T2 to overflow, and it cannot be countered using the current counterattacker model. This and similar attacks cannot be countered because the counterattacker in Stage 3 cannot compensate some effects of the attacker over T2 by influencing components under their control. In summary, we identified 333 attacks that cause T2 to overflow, 126 that cause T2 to empty and 108 that cause T3 to empty; no attack caused T3 to overflow because we stop analysis at the first broken requirement, and attacks that may overflow T3 also overflow T2, empty T2 or empty T3 faster than they could overflow T3.

5.6.7 Increasing Robustness

From the data obtained during LBA, we group attacks that are countered by the same counterattack. After evaluating the composition of Stage 2 and Stage 3 under the given attacker and counterattacker models, we observe that 12 non-trivial counterattacks counter 431 successful attacks. The number of attacks countered by each counterattack does not follow a uniform distribution; some counterattacks counter only a couple of attacks, while others counter hundreds.

Understanding the similarities among the attacks countered by the same counterattack should give us a good indication for why the counterattack is effective. For this evaluation, we manually analysed possible common causes, but a more systematic and efficient methodology is encouraged for future work.

A Correct Repair

In Section **??**, we briefly mentioned that, to soundly apply a counterattack, we must ensure to only apply it when an attack it counters exists in the system. This implicitly assumes the existence of a very powerful system monitor that can correctly identify which attack is being applied to the system. Once the is identified, the entity implementing the counterattacker applies the appropriate counterattack. Given that these requirements over the monitor are quite restrictive, we explore a different approach: we directly modify the program of a controller PLC based on the information we have on counterattacks, and we run LBA to measure the robustness of the new system to check if it improved.

We used a counterattacker model that acts on the coordinates $\vec{u}[MV3]$ and $\vec{u}[P3]$, which are normally controlled by controller PLC3. Our plan is to incorporate the "most useful

counterattack" as part of the program of PLC3. Two things are necessary: the conditions that would trigger the application of this counterattack, and the right effects over $\vec{u}[MV3]$ and $\vec{u}[P3]$.

From LBA, we observe that several attacks that empty tank T3 can be countered by means of the counterattack

$$\vec{w}(\pi) = \begin{bmatrix} \mathtt{id} \circ \Delta_{\mathtt{off}}^{\vec{u}[P3]} \\ \mathtt{id} \circ \mathtt{id} \\ \mathtt{id} \circ \mathtt{id} \end{bmatrix} \cdot \begin{bmatrix} \pi[\vec{y}[L3]] < L3min \\ L3min \leq \pi[\vec{y}[L3]] \leq L3max \\ \pi[\vec{y}[L3]] > L3max \end{bmatrix},$$

which is sensible: if the tank $T3$ is losing water, by shutting off the pump $P3$ when the level is low, we prevent water from leaving the tank. The counterattack $\vec{w}$ counters 122 attacks out of the 729 attacks generated by the given attacker model.

The counterattack $\vec{w}$ defines the following effects over $\vec{u}[MV3]$ and $\vec{u}[P3]$ to be implemented by the controller PLC3: do nothing with respect to $\vec{u}[MV3]$ and set $\vec{u}[P3]$ to $\mathtt{off}$ only if $\pi[\vec{y}[L3]] < L3min$. While this gives us the implementation of the counterattack, we still need to find a trigger condition for it. We look for common elements among the attacks countered by $\vec{w}$, and we observe among several of those attacks that they share the factor $(\Delta_{\mathtt{off}}^{\vec{u}[P2]})(\vec{x}[L3]] < L3min)$. This factor provides the trigger condition for the application of the effects given by $\vec{w}$; more precisely, to implement the counterattack $\vec{w}$ as part of the controller PLC3, we change its program such that $\vec{u}[P3]$ is $\mathtt{off}$ if $\vec{u}[P2]$ is $\mathtt{off}$ and $\vec{y}[L3]] < L3min$.

After implementing these changes, we run LBA again. We see that the changes to PLC3 increase the robustness of the system from $162/729$ to $333/729$, since the controller PLC3 now prevents the tank T3 from running out, countering the effects of 171 previously successful attacks. We recall that $\vec{w}$ countered 122 attacks according to the original LBA, so the change to the program of PLC3 is more profound than anticipated. Nevertheless, the increase in robustness indicates that the system is now more robust after the change, even if the latent robustness remains the same, $593/729$.

A Non-Repair

We now explore the effects of a modification to the program of PLC3 that does not improve the overall robustness of the system; i.e. a non-repair. The counterattack

$$\vec{w'}(\pi) = \begin{bmatrix} \mathtt{id} \circ \mathtt{id} \\ \Delta_{\mathtt{open}}^{\vec{u}[MV3]} \circ \mathtt{id} \\ \mathtt{id} \circ \mathtt{id} \end{bmatrix} \cdot \begin{bmatrix} \pi[\vec{y}[L3]] < L3min \\ L3min \leq \pi[\vec{y}[L3]] \leq L3max \\ \pi[\vec{y}[L3]] > L3max \end{bmatrix},$$

counters 58 attacks, many of which overflow T2 and have a common factor of $(\vec{x}[L3] > L3max)\Delta_{\mathtt{on}}^{\vec{u}[P2]}$, i.e., many attacks turn on the pump $P2$ then the level of tank $T3$ is beyond the safety limit $L3max$. Since the condition of the attacks $(\vec{x}[L3] > L3max)$ is mutually exclusive with the condition of the counterattack $L3min \leq \pi[\vec{y}[L3]] \leq L3max$, we prioritise the condition of the counterattack over those of attacks, and we implement it in the controller directly as a rule. To check if such modification improved robustness, we perform LBA, and we observe that the robustness factor remains the

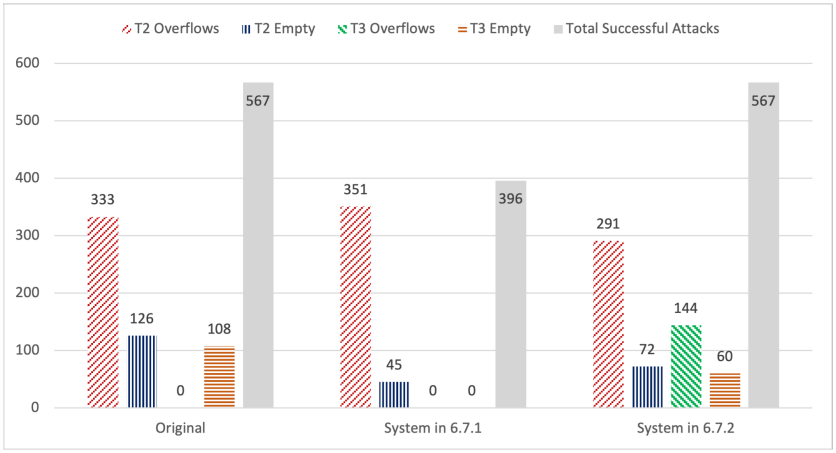


FIGURE 5.6: Attack distribution with respect to the first broken requirement for the original system, the repaired system from Section **??** and the non-repaired system from Section **??**

same at 162/729; i.e., this new system also resists the effect of 162 attacks. We observe two major differences: the latent robustness of the system changed from 593/729 to 606/729, and the distribution of successful attacks with respect to the first requirement they break also changes, as shown in Figure **??**. The robustness of the new system is equal to the original robustness, so we cannot consider this to be an improvement over the previous version of the system, which is why we affirm that this modification is an example of a non-repair.

This modification reduced the number of attacks that overflow tank T2, but it now enables attacks that overflow tank T3, which were previously inexistent (now there are 144 attacks whose first broken requirement is to overflow tank T3). An increase in latent robustness indicates that more attacks in this new system can be countered, but this vaguely indicates that the new system has more *security potential* than the original system with respect to the given attacker and counterattacker model.

5.6.8 Summary and Discussion of Results

We use LBA and a testing methodology based on simulation to check if an arbitrary modification enhances the robustness of a CPS by comparing the original and the resulting robustness factors. We propose two changes: one is a repair since it strictly improves the robustness of the system, while the other is not since it only changes the distribution of successful attacks. We believe that a more systematic analysis of the large amounts of data obtained via LBA can produce meaningful suggestions for correct model repairs, e.g., using machine learning or clustering techniques to find common causes of attacks that are countered by the same counterattack, or finding similarities among counterattacks for the same types of requirements might help the implementation of both system monitors and repairs.

Figure **??** shows a transition from the robustness that is innate to the systems presented in this section and their latent robustness. This transition assumes shows the potential

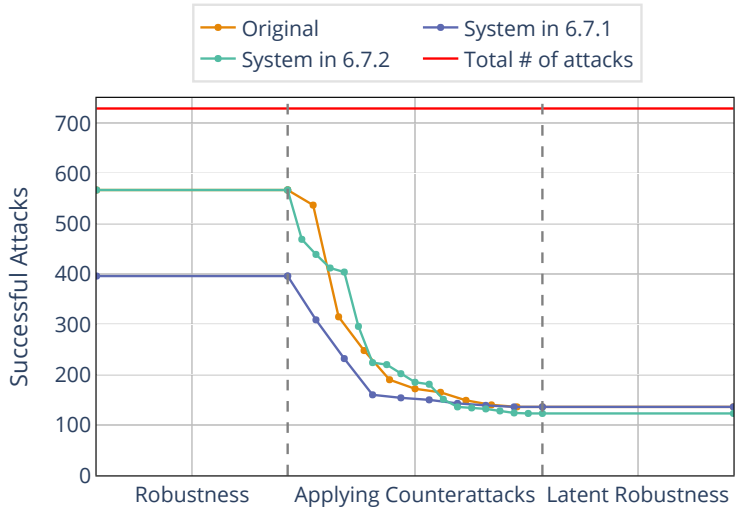


FIGURE 5.7: Robustness improvement. The implementation of counterattacks improves the robustness of the system, making it susceptible to fewer attacks.

for improvement if counterattacks were properly implemented, given the existence of an adequate attack identification monitor. When properly implemented, a counterattack reduces the number of successful attacks, which in turn increases the robustness of the system. Each dot in the "applying counterattacks" section corresponds to the cumulative reduction of successful attacks by the proper implementation of a counterattacks. The slight difference in latent robustness is due to the fact that the original system and the system in Section **??** are different, and the latter has more attacks that can be countered under the current attacker and counterattacker models.

For this experiment, we used a very basic counting unweighted metric for robustness that assumes all requirements are equally important. We can refine the definitions of robustness and latent robustness metrics from Definitions **??** and **??** to represent a larger lack of robustness if a particular security requirement is broken by many attacks or to reflect other measurement biases.

5.7 Related Work

Information Flow Analysis (IFA) In [**CPSSec**], Gollmann and Krotofil state that "Physical relationships between the variables in an industrial process, e.g. between volume, pressure, and temperature in a vessel, can be viewed as information flows from a modelling perspective," acknowledging that it is possible to model physical aspects of CPS using an information flow setting; however, they do not explicitly say how. Gamage *et al.* [**Gamage2010**] use IFA to illustrate how to prevent attacks on confidentiality of CPS though the notion of *compensating pair* (a, a^c), where a^c is an action that cancels the physical manifestation of the earlier occurring action a so that attackers do not infer

that a took place. In our setting, counterattack play the part of "compensating pairs" for integrity attacks, mitigating the action of attacks.

Clarkson and Schneider show in [**QuantitativeIntegrity**] how to adapt traditional measures of information leakage (which quantify of confidentiality) to provide a measure for information contamination/corruption (which quantifies integrity). In essence, Clarkson and Schneider determine the highest rate of contamination by modelling programs as channels and by using mutual information between untrusted inputs and trusted output, and trusted input. We believe that our framework fits their "program as a channel"-model as follows: attacks are untrusted inputs, and the security requirements are trusted outputs. We did not explicitly consider IFA techniques that could trivialise the quantification of the robustness of a system with respect to a given attacker model, but we believe they are compatible with our framework, e.g., if an attacker acts on a region of the system that is isolated, or has no attacks at all, then the robustness with respect to that attacker is one, since no information flows from the attacker to critical components.

On Attacker Models Several authors have proposed other attacker models for CPSs. For example, Howser and McMillin provide in [**StuxnetOnCPS**] an information flow-based attacker model that builds on top of nondeducibility [**Nondeducibility**], where attackers aim to hide information relevant to attacks or faults to the monitoring systems, preventing operators from realising that the behaviour of the system is anomalous. Our analysis is not aimed at deciding whether the attacker hides information from the operator, and instead focuses on preserving the integrity of the system. Rocchetto and Tippenhauer [**CPSDolevYao**] extend the Dolev-Yao attacker model [**DolevYao**] to the Cyber-Physical Dolev-Yao (CPDY) model, where attackers can interact with the physical domain through orthogonal channels. We believe that their attackers can be modelled in our framework with attackers that control the coordinates of the control vector $\vec{u}$, but a formal proof is still missing.

Control Theory Most control-theoretical approaches to CPS security assume the existence of a probabilistic behavioural model that is used as a reference of normal (attack-free) behaviour, and their goal is to monitor and protect the system with respect to this ideal model at runtime, when attacks might arise. There are some limitations with this approach. On the one hand, this approach is mainly reactive due to its reliance on monitors and observers, and as such it does not shed light on how to improve a given CPSs design to make it inherently more robust (although there are results on redesigning controllers and models to improve robustness of CPSs against hidden sensor attacks [**ReachableSets**; **Weerakkody**]). On the other hand, an inherent limitation of many behavioural models for CPS is that they usually are approximate due to linearisation, and might produce a high false-positive rate when used in practice; moreover, some works (e.g., [**Urbina2016**; **CPSDetectingIntegrityAttacksScada**; **IFCPSSec**]) rely on the assumption that it is possible to determine whether the system is operating normally or under attack, but there could be non-malicious deviations of a given behavioural model due to routine maintenance operations or other random benign factors. Finally, only few approaches (such as [**IFCPSSec**; **Gupta2**; **ReachableSets**]) quantify the severity and consequences of attacks on the system.

From the perspective of control theory, the work by Weerakkody *et al.* **[IFCPSSec]** proposes the *Kullback-Leibler divergence* between the distributions of the attacked and attack-free residuals as a measure for information flow to determine to which extent the actions of an attacker interfere with the system. However, their definition of security is ultimately tied to a maximum deviation from a prediction model, while ours is based on how many attacks given an attacker model are successful at breaking at least one security requirement. Thus, although we can imagine that both measures are somehow related, it is not evident what their exact relationship is.

We believe that it is possible to provide a reactive implementation of the counterattacks found via LBA by relying on monitors similar to those used by control theoretical approaches. However, we ultimately suggest to change the system based on the relations between attacks and counterattacks, and quantify the robustness of the resulting system to attest that the new system is inherently more robust than its original version.

Verification Several tools compute an over approximation of the set of reachable states, which we can use to verify safety properties (see, e.g., Flow* **[FlowStar]**). These tools are extremely valuable for the verification of safety properties, but in the presence of adversaries, we believe that a testing approach like the one proposed in this work is more suitable given the exponential number of scenarios to verify, and the tendency of attacks to break safety requirements. This reasoning also extends to other verification tools like McLaughlin *et al.*'s **[TSVPLC]** that use symbolic execution for model checking safety properties in PLCs.

5.8 Discussion and Future Work

We highlight that the parametrisation of LBA (i.e., the security requirements, the attacker model, and the counterattacker model) clearly defines the boundaries and the meaning of the security guarantees concluded by the analysis. There is ample room for improvement, since a single analysis only covers one point in the combinatorial space of such parameters, which is exponential in size. To obtain more comprehensive security guarantees, we should systematically vary the parameters of the analysis in a way that we cover as much as we can of the parameter space in an organised and comprehensive way. We can take inspiration from other testing approaches, especially fuzzing and fault injection, which use heuristics to expand their coverage and guide the analysis.

We also highlight that the vast amount of data produced by LBA has clearly defined features. For example, attacks and counterattacks are defined in terms of state conditions and actions, and they may be related if one counters the other. This opens a potential collaboration between LBA and machine learning techniques, where LBA systematically generates training data for monitors (from the attack data) and reactive modules (from the counterattack data).

We now discuss other concrete points that could benefit from future work.

5.8.1 Modelling and Abstraction level

While many CPSs are probabilistic systems in nature, our formalism defines a deterministic transition system whose transition function is given by the single-cycle semantics function $[\![\cdot]\!]$ (see Definition **??**). Consequently, the notion of robustness and latent robustness that we give in Definitions **??** and **??** are not probabilistic. We believe that our approach is good enough to carry out an approximate analysis if the probabilistic nature of the process is due to the existence of zero-mean noises. Moreover, since we use a testing approach, we believe that it is possible to adapt the LBA to obtain statistically significant results (e.g., by repeating experiments), missing only rare events (which is an innate limitation of testing approaches).

Hybrid automata [**ALUR19953**] are state-based models for dynamical systems. These automata allow the description of invariants and side-effects on transitions, making them suitable for the description of many continuous physical processes. Being state-based systems, we believe they are compatible with LBA, and their attacker models could contain transformations that change the state of the continuous process while preserving the current discrete mode of the system (as long as the transformation is physically possible).

5.8.2 Beyond Robustness

There are many well-known properties in control theory that characterise important aspects of dynamical systems, including controllability, observability, and stability. Since we focus on the protection of the integrity of the process, we want to quantify the degree of *controllability* that the attacker has over the system. A focus on the protection of the confidentiality of data would aim to quantify the degree of *observability* that the attacker has over the system. Krotofil and Larsen acknowledge the importance of reasoning about controllability and observability, but they highlight that it is also important to reason about *operability*, which is "the ability to achieve acceptable operations" [**krotofil2015rocking**]. Some counterattacks could take the system to a safe state when implemented, but such a state may be inoperable, and requires a restart of the system. By adding operability and functionality requirements to the LBA when discovering counterattacks, we refine the search of useful counterattacks, and is an interesting avenue for future work.

5.8.3 More Automation

LBA offers several use cases for automation. Partitioning the state space could be automated by means of data-flow analysis of the controller, i.e., by a procedure that determines the path conditions of the program of controller, as illustrated in [**castellanos2021AttkFinder**]. Moreover, the transformations we use to define attacker models in Section **??** can be systematically obtained by a fuzzing framework (e.g. AFL [**AFL**] or [**chen2019learning**] for CPSs) or by a symbolic execution engine (see [**castellanos2021AttkFinder**]).

5.8.4 Coverage

Each LBA offers only a partial quantification of robustness since it is parametrised by an attacker and a counterattacker model. For a more comprehensive result, we must systematically explore different attacker and counterattacker models covering several coordinates of the system. This systematic exploration of attacker models benefits from the monotonicity properties presented in Corollaries **??** and **??**. We leave this systematic exploration of attacker and counterattacker models in CPSs for future work.

Chapter 6

Transforming Programs for Timing Side-Channel Repair

6.1 Introduction

Until now, we have considered the effect of spatial transformations from an integrity perspective. The attacker has used spatial transformations to perform integrity attacks to alter the behaviour of systems, and we have quantified how robust a system is with respect to a set of attacks derived from a given attacker model. Figuratively, the attacker has used spatial transformations for destructive purposes. In this chapter, we explore the possibility of using spatial transformations for constructive purposes only, i.e., only for the defender while the attacker is using weaknesses of the system that leak confidential information. More precisely, we are interested in using spatial transformations for enforcing protection against side-channel attacks that use the timing channel to leak secrets.

What are timing side-channel attacks? Timing attacks are among the best known side-channel attacks [**timing-channel-survey**] to ex-filtrate secret information from a program. Basic timing side-channel attacks aim to establish a relationship between inputs and execution time, which can be done by attackers who have a copy of the program. After running the program with different inputs, the attacker has a model of this input to execution-time relationship. Attackers then observe the total execution time of the same program run by the victim, and can infer the value of the secret input.

More advanced timing attacks exploit micro-architectural features, e.g. caches, and they require the attacker be able to interact with these micro-architectural aspects in the machine where the victim executes the vulnerable program. Spectre style attacks [**Spectre**], as discovered in 2018, are sophisticated attacks where the attacker exploits timing covert channels to ex-filtrate secret information loaded by speculative execution in the cache by reverse engineering the value of a secret input after observing cache hit/miss timings [**timing-cache**] or by computing the cache lines being accessed [**prime-probe**]. While Spectre attacks rely on speculative execution to load secrets into the cache, poorly implemented cryptographic software could directly leak secrets via memory access patterns, even when speculative execution is disabled.

Repairing Leakage due to Memory Access Patterns (MAP) If we assume that attackers can only infer information from the behaviour of the program counter, then the automatic repair of programs with timing side-channel vulnerabilities is a well understood problem; moreover, existing solutions [**SCEliminator**; **MSESC**; **Racoon**] do an acceptable job at closing timing side-channels while preserving functionality and preventing the appearance of undesired side-effects (e.g. unsafe memory accesses). However, once we empower the attacker to manipulate micro-architectural aspects, especially those that breach memory isolation like the ones used for Spectre [**Spectre**] and Meltdown [**Meltdown**], the repairing of vulnerable programs remains an open problem [**timing-channel-survey**].

In this work, we are particularly interested in repairing programs that leak information when a data structures is accessed using a secret value. These programs are vulnerable to an attacker that cannot read the contents of the cache directly but can manipulate and observe the state of the cache using attacks like Flush+Reload [**Flush+Reload**] or Prime+Probe[**prime-probe**] to infer secrets.

Consider the example program **P**, defined by

$$\text{if } A[s] \text{ then } x := 1 \text{ else } y := 0,$$

which reveals $A[s]$ under the *baseline leakage model*, because the attacker can infer the value of $A[s]$ by following the program counter. The baseline leakage model assumes that the valuations of branch conditions are exposed to attackers (see [**usenix_ctp_verification**]), so the program **P** is unsafe if `A[s]` is a secret, but it is safe if `A[s]` is public (e.g. if `A[s]` is declassified data). Existing repair tools [**SCEliminator**; **MSESC**; **Racoon**] offer strong security guarantees against the baseline leakage model, and can repair the program **P** with respect to this leakage model, yielding the linear-code program **T(P)**, defined by

$$x := CTSel(A[s], 1, x); y := CTSel(A[s], y, 0),$$

where $CTSel(c, a, b)$ is a constant-time selector which returns a if c is true, or b otherwise. The program **T(P)** is arguably safer with respect to the baseline leakage model, since the behaviour of the program counter no longer reveals the value of $A[s]$.

Now, if we consider a leakage model which considers Memory Access Patterns (MAP), then the program **T(P)** leaks s, because the attacker can still infer s by probing the cache. Existing compiler-based repair tools [**SCEliminator**; **MSESC**; **Racoon**] offer weak, inefficient, or no guarantees at all against this leakage model: Racoon [**Racoon**] implements ORAM, which is quite taxing in terms of performance (it induces an overhead with geometric mean 16x), SC-ELIMINATOR uses data structure preloading, but it is an unsound repair if the attacker can manipulate the cache, and the methodology presented in [**MSESC**] does not repair against this leakage model. This lack of effective and efficient repair guarantees is the main motivation for our work.

Our Contributions The authors of [**WhatYouCisWhatYouGet**] describe a philosophy which aims to delegate the compiler the enforcement of timing side-channel freedom. We follow this philosophy, and we propose a set of repair rules to close the timing side-channel created by accessing fixed-size data structures indexed by secret information.

Our repair rules offer strong security guarantees with respect to the leakage model that accounts for memory access patterns. In a nutshell, we replace each read access $y := A[x]$ and each write access $A[x] := y$ by a linear program that explores A, systematically loading it in memory, guaranteeing that $y = A[x]$ in the case of a read, and that $A[x] = y$ in the case of a write. These operations are similar to *fold* high-order functions, which is why we name our repair rules ORIGAMI.

We implement ORIGAMI as an LLVM opt pass to make it compatible with other target-independent compiler optimizations. The tool lets the compiler first perform optimisations, and then the tool applies the ORIGAMI repair rules to the resulting intermediate representation code. Our pass ensures that compiled programs have a constant memory footprint when indexing fixed-size array-like data structures using secrets. Our proposed solution only requires minimal annotation to the source code (i.e., to inform the compiler which variables and function arguments are secrets, and for loops whose bounds cannot be automatically derived by the compiler) and provides theoretical guarantees that the transformed code is secure under the memory access pattern leakage model.

There are a couple of clear limitations when repairing programs with ORIGAMI, (e.g. ORIGAMI can obfuscate a read access to a data structure whose values are secret pointers, but fails to protect the program if such resulting pointer is later used to load or store a value) which we discuss in Section **??**. These limitations illustrate the impossibility of repairing every program program with respect to the the memory access patterns leakage model while keeping the program functional, efficient, and secure.

This paper is structured as follows: we first provide a brief background in Section **??**. In Section **??**, we formally define timing side-channel freedom (TSCF) under the MAP leakage model, which is the property that we want to enforce. We present the ORIGAMI repair rules in Section **??**, and we prove that they enforce TSCF under MAP for a language of while programs; we also discuss the limitations of this enforcement. In section **??**, we evaluate an implementation of the ORIGAMI rules as LLVM optimization passes; we use these passes to enforce TSCF in LLVM-IR after all other compiler optimizations have taken place. We apply ORIGAMI to a small toy example and to real cryptographic ciphers from OpenSSL [**OpenSSL**] and to GDK library routines [**gdklib**; **gdklib**], and we evaluate all the repaired programs using GEM5 – a cycle accurate simulator for x86 processor – to empirically show that all repaired programs satisfy TSCF with respect to MAP leakage. We then compare ORIGAMI against related work in Section **??**, and we conclude in Section **??**.

6.2 Preliminaries

In this section, we provide the definitions and notation that we use through this work.

6.2.1 Timing Side Channel Freedom Enforcement

Why are timing side-channel vulnerabilities so hard to fix? Both the programming languages and the computer security communities understand fairly well where timing differences could be introduced during the compilation and execution processes of

software, and they tackle the problem of enforcing *timing side-channel freedom (TSCF)* using a layered approach. Unfortunately, timing differences can be (unintentionally) introduced at every step of the compilation process, and they propagate to the following stages.

For illustration purposes, let us consider a simplified version of the compilation and execution process. A program starts out as *source code*, which is then given to a compiler. The compiler often creates an *intermediate representation* (IR) of the program (e.g. a control flow graph (CFG)), which the compiler then optimises by using transformation rules that can be applied to *any* IR (e.g. dead-code elimination). We call the entity in charge of creation and optimisation of the IR the *front-end* of the compiler. The *back-end* of the compiler then compiles the optimised IR into a microarchitecture-dependent *low-level representation* (LLR), performs microarchitecture-dependent optimisations, and then creates the executable. Finally, the executable runs on the microarchitecture by following the sequence of instructions in the executable.

At the *source code* level, a developer who does not follow constant-time programming guidelines, e.g., CryptoCoding [**CryptoCoding**], can introduce timing differences by, e.g., using loops with input-dependent bounds, or by terminating early if a branch condition is satisfied, e.g. in a base case of a recursive functions. At the *IR* level, since compilers often optimise for performance, they may introduce timing differences via optimisations at the IR level, just like a programmer would at source level. To make matters more complicated, the compiler may even remove TSCF countermeasures introduced at the source code level if it deems them non-optimal, which is why developers of crypto algorithms disable compiler optimisations, or even choose to avoid compilers altogether and instead directly implement crypto routines in assembly [**timing-channel-survey**].

Then, the back-end repeats the story of the front-end: it creates the LLR, optimises it and creates the executable, but its own optimisations may remove any TSCF enforcement introduced in the IR, and it may itself introduce timing differences. Finally, even if the back-end does not itself introduce timing differences or removes countermeasures added at the previous stages, the microarchitecture may manifest timing differences during program execution; this may be because a micro-architectural instruction can vary its execution time depending on its parameters (e.g. multiplication), or due to out-of-order execution and speculative execution. In that sense, any TSCF enforcement introduced at early stages can be made irrelevant at later stages.

6.2.2 Leakage Models

We find the notion of leakage models used in `ct-verif` [**usenix_ctp_verification**] particularly enlightening. Although their leakage models are defined based on LLVM rather than machine code, they argue in [**usenix_ctp_verification**] that "LLVM assembly code produced just before code generation [is] sufficiently similar to *any* target-machine's assembly code to provide a high level of confidence."

In the following, we provide the intuition behind three useful leakage models, what it means for a program to leak secrets with respect to them, and insights on what repairing a program with respect to each model entails.

Baseline Leakage Model This leakage model reveals to the attacker the valuations of branch conditions. More precisely, the program

$$\text{if } c \text{ then } p_1 \text{ else } p_2$$

reveals the valuation of c, and the program

$$\text{while } c \text{ do } p$$

reveals the valuation of c. This is the *baseline* leakage model because it is implied by all other leakage models.

A program leaks secrets with respect to this model if secrets influence the behaviour of the *program counter*, which is why it is also known as the *program counter security model* [**Molnar**]. To repair a program with respect to this model, secret-dependent branches are linearised by replacing conditionals with constant-time selectors and loops are fully unrolled. This causes the behaviour of the program counter to be independent of value of secrets.

Memory Access Patterns Leakage Model (MAP) in addition to revealing the valuation of branch conditions, the MAP model reveals the indices used to access data structures. More precisely, the programs

$$A[x] := y, \quad \text{and} \quad y := A[x]$$

each reveals x because different indices may have different memory access patterns (e.g. when the cache lines for $A[x]$ and $A[x']$ are different), and the attacker can infer this information. Thus, a program leaks secrets under the MAP model if they are used to access data structures [**usenix_ctp_verification**]. This leakage model is related to memory trace obliviousness [**MemoryTraceOblivious**], which requires constant behaviour of the memory for all public-equivalent traces. To avoid leakage under the MAP model, we must not use secrets when accessing data structures, and we must enforce a secret-independent behaviour on the program counter (to avoid leakage following the baseline model).

SC-ELIMINATOR proposes the use of preloading and must-hit analysis to repair programs so that they satisfy TSCF under the MAP leakage model. Unfortunately, this repair implicitly assumes that the state of the cache during must-hit analysis is the same as when the program executes, which is a problematic assumption if we consider that the attacker can also manipulate the cache using Prime+Probe and Flush+Reload attacks. Instead of preloading, we propose a new repair rule where instructions accessing data structures using secrets, i.e.$A[x] := y$ and $y := A[x]$, have the constant memory access patterns, thus preventing leaks under the MAP. The details of this solution are explained in detail in Section **??**.

Operand Sensitive Leakage Model (OS) For completeness, we include the leakage model that distinguishes operations whose execution time are sensitive to inputs. The program $y := f(x)$ leaks its parameter x if its total execution time depends on x. This

leakage model is the most general of the three models presented, as it implies the MAP and baseline models. Repairing programs with respect to the OS model at the source or compiler level is extremely challenging, because some operations offered by the micro-architecture leak their parameters (e.g. division and multiplication); thus, solutions for the OS model may need to be target dependent. Enforcing TSCF with respect to this leakage model is outside the scope of this work.

6.2.3 TSCF Under MAP Leakage - Informally

The MAP leakage model represents an attacker that is able to use the timing of hits and misses in the cache to indirectly obtain values from the cache, similar to what attackers relying on Spectre attacks do to recover the secrets loaded in memory. More precisely, if an array-like structure A is large enough to require several cache lines for it to be fully loaded in the cache, then caching $A[s]$ only fills the cache line that corresponds to the index s and its neighbouring values. An attacker can gain information about s by probing the cache, testing which parts of A result in a cache hits and which ones do not.

For example, under the MAP leakage model, the program

$$\text{if } s < size_A \text{ then } x := B[A[s]]$$

reveals both s and $A[s]$, because the state of the cache is different for different values of s. This program does not reveal $B[A[s]]$, only the indices used to access it.

6.2.4 Guarded Kleene Algebra with Tests (GKAT)

Guarded Kleene Algebra with Tests (GKAT) is a modern formalism that offers a propositional abstraction of imperative while programs with uninterpreted actions [**GKAT**]. The specialty of GKAT is to enable reasoning about properties of programs *by merely looking at their structure and not at their (functional) semantics*. This makes GKAT interesting for modelling transformations at the compiler level [**KATForCompilers**], because a general-purpose compiler should not need know the exact semantics of a program to optimise it; in general, the compiler should look for structural patterns which enable optimisations, just as GKAT does for reasoning.

We use GKAT to provide a formal foundation in reasoning about TSCF for arbitrary programs. Specifically, the axioms and rules for GKAT expressions help us reason about the equivalence of programs. In this setting, the notion of semantic equivalence is defined over uninterpreted actions using languages of *guarded strings*, defined using *actions* and *tests*.

Actions, Tests and Expressions Every GKAT is parametrised by a set of abstract *actions* Σ and a finite set of abstract *primitive tests* T. We assume T and Σ are disjoint and non-empty. A test $t \in T$ is an atomic proposition about the state of the program, and the execution of an action $p \in \Sigma$ can affect the state. We form *GKAT expressions* with the grammar presented in Figure **??**.

Atoms An *atom* is a truth assignment of all the tests in T. We denote atoms by α, β, and γ, and the set of atoms by A. For example, if $T = \{t_1, t_2\}$, the boolean expressions $\alpha = \overline{t_1} \cdot \overline{t_2}$, $\beta = \overline{t_1} \cdot t_2$, $\gamma = t_1 \cdot \overline{t_2}$, and $\delta = t_1 \cdot t_2$ are all the atoms, where $\overline{t_i}$ is the complement of t_i, for $i \in \{1, 2\}$.

Guarded strings are an intercalation of a logical atom and an action, which can be seen as the concatenation of $\{\texttt{pre}\}\{\texttt{action}\}\{\texttt{pos}\}$ elements, where `pre` and `pos` represent the precondition and postcondition of the `action` (any `pre` and any `pos` as we work with uninterpreted actions, but a `pos` and a `pre` need to be compatible to be concatenated).

Guarded Strings A *guarded string* g is an element of the set $\mathrm{GS} := A \cdot (\Sigma \cdot A)^*$, and it models a trace of an abstract program. To compose guarded strings, we use *fusion product* $\diamond$: $\mathrm{GS} \times \mathrm{GS} \to \mathrm{GS}$, a partial function defined by

$$w\alpha \diamond \beta v \triangleq \begin{cases} w\alpha v, & \text{if } \alpha = \beta; \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{6.1}$$

The fusion product of sets $L_1, L_2 \subseteq \mathrm{GS}$ is defined by $L_1 \diamond L_2$, where

$$L_1 \diamond L_2 \triangleq \{g_1 \diamond g_2 \mid g_1 \in L_1, g_2 \in L_2, \text{ and } g_1 \diamond g_2 \text{ is defined}\}.$$

Language-based Semantics The *language-based semantics* of a GKAT expression is a set of guarded strings (i.e., a language) defined by

$$\begin{aligned} [\![p]\!] &\triangleq \{\alpha p \beta \mid \alpha, \beta \in A\}, \\ [\![b]\!] &\triangleq \{\alpha \mid \alpha \in A \text{ and } \alpha \Rightarrow b\}, \\ [\![f \cdot g]\!] &\triangleq [\![f]\!] \diamond [\![g]\!], \\ [\![f +_b g]\!] &\triangleq ([\![b]\!] \diamond [\![f]\!]) \cup ((A - [\![b]\!]) \diamond [\![g]\!]), \\ [\![e^{(b)}]\!] &\triangleq \bigcup_{n \geq 0} ([\![b]\!] \diamond [\![e]\!])^n \diamond (A - [\![b]\!]), \end{aligned}$$

where $L^0 \triangleq A$ and $L^{n+1} \triangleq L^n \diamond L$, for $L \subseteq \mathrm{GS}$. Since actions in GKAT expressions are uninterpreted, language-based semantics are an over-approximation of functional semantics.

Rules A *GKAT rule* is an equivalence that lets us transform an arbitrary GKAT expression into a GKAT expression that is syntactically different, yet semantically equivalent; e.g., $b \cdot (e +_b f) \equiv b \cdot e$.

6.3 Formalising TSCF with MAP

Informally, TSCF means that all secret-dependent program traces have very similar *execution time*. If we model program traces using guarded strings, then the execution

$$
\begin{array}{lll}
b, c, d, \in \text{BExp} ::= & & \\
\quad | \quad 0 & & \textbf{False} \\
\quad | \quad 1 & & \textbf{True} \\
\quad | \quad t \in T & & t \\
\quad | \quad b \cdot c & & b \textbf{ and } c \\
\quad | \quad b + c & & b \textbf{ or } c \\
\quad | \quad \bar{b} & & \textbf{not } b
\end{array}
\qquad
\begin{array}{lll}
e, f, g, \in \text{Exp} ::= & & \\
\quad | \quad p \in \Sigma & & \textbf{do } p \\
\quad | \quad b \in \text{BExp} & & \textbf{assert } b \\
\quad | \quad e \cdot f & & e; f \\
\quad | \quad f +_b g & & \textbf{if } b \textbf{ then } f \textbf{ else } g \\
\quad | \quad e^{(b)} & & \textbf{while } b \textbf{ do } e
\end{array}
$$

FIGURE 6.1: **Left:** boolean expressions. **Right:** GKAT expressions (from **[GKAT]**).

time of a program trace is the sum of the execution time required by each of its actions. To quantify the execution time of actions, we use a *time metric*.

A *time metric* associates each actions in a program trace with a time consumption. Under the MAP leakage model, an action p may consume different amounts of time depending on the state of the cache when p is executed: if a results in several cache hits then it consumes less time than it would if a resulted in several cache misses. Thus, the memory access pattern of a program trace directly impacts its execution time. We formalise this notion with the metric $\texttt{mem}$.

The `mem` Time Metric Let $\mathscr{V}(p)$ be the variables used in the action p, let $\omega \colon \text{GS} \times \mathscr{V} \to 2$ be a function where $\omega(g, v)$ states if the variable v is in the cache after executing g (or after executing nothing if $g \in A$), let $\texttt{hit}_g(p) \triangleq \{v \in \mathscr{V}(p) | \omega(g, v)\}$ be the set of variables in p that are present in the cache, and let $\texttt{miss}_g(p) \triangleq \{v \notin \mathscr{V}(p) | \omega(g, v)\}$; we define $\texttt{mem}(g)(p)$, the *MAP of p (with respect to g)*, by

$$\texttt{mem}(g)(p) = \eta \times \texttt{miss}_g(p) + \mu \times \texttt{hit}_g(p)$$

where $\eta, \mu \in \mathbb{R}^+$ are constants with η being much greater than μ, respectively modelling the time to load a variable from memory and the time to load a variable from the cache.

Given a GS g, the *MAP of* g, denoted $\text{RC}_{\texttt{mem}}(g)$, is the accumulated MAP of its actions; formally,

$$\text{RC}_{\texttt{mem}}(g) \triangleq \begin{cases} 0, & \text{if } g \in A; \\ \text{RC}_{\texttt{mem}}(h) + \texttt{mem}(h)(p), & \text{if } g = h \cdot p \cdot \alpha; \end{cases}$$

The metric $\texttt{mem}$ is *causal*, since the resource consumption of actions depends on the history of the execution.

Now, consider two values for a secret s, say $s = 0$ or $s = n$; the $\texttt{mem}$ values of $x := A[s]$ when executing with an empty cache is

$$\texttt{mem}(1)(x := A[s]) = \begin{cases} \eta \times \{x, A[0], s\} + \mu \times \emptyset, & \text{if } s = 0 \\ \eta \times \{x, A[n], s\} + \mu \times \emptyset, & \text{if } s = n. \end{cases}$$

The secret s leaks because the MAP of the expression $x := A[s]$ depends on the value of s. When loading $A[0]$ or $A[n]$ into the cache, if they touch different regions of the cache, the attacker can infer the value of s by looking at which regions corresponding to A are loaded.

A preloading strategy, as the one shown in **[SCEliminator]**, places the contents of the structure A in the cache so that $\texttt{mem}(1)(x := A[s]) = \eta \times \emptyset + \mu \times \{x, A[s], s\}$ for all s, implying that the attacker cannot infer information from checking which regions are loaded when we use a secret as an index in A, since all relevant regions are loaded if A fits in the cache. However, recall that the attacker can flush the contents of the cache after preloading, so preloading is not a viable strategy: as long as there is an instruction $x := A[s]$ or $A[s] := x$, a preloading strategy is not sound with respect to TSCFs.

Constant MAPs Given a language of guarded strings $L \subseteq \text{GS}$, we say that L has *constant resource consumption with respect to* $\texttt{mem}$ if and only if $\text{RC}_{\texttt{mem}}(g_1) = \text{RC}_{\texttt{mem}}(g_2)$ for all $g_1, g_2 \in L$.

Constant resource consumption in its current form requires *all* GS in the language L to have the same resource consumption. However, a program that has no secret values trivially satisfy this property, since there are no secrets that could be leaked. Thus, we need to enrich the current definition of constant resource consumption so that we only require traces that exclusively differ on secret values have the same resource consumption. As it is standard (see e.g., **[Molnar; usenix_ctp_verification]**), we introduce a notion of *public equality*.

Let $\mathscr{V}$ be the set of public variables, and let $[\![\mathscr{V}]\!]$ be the set of valuation functions which map variables to values; we extend the notion of guarded strings so that they are now generated by the grammar

$$\text{GS} := ([\![\mathscr{V}]\!] \times A) \cdot (\Sigma \cdot ([\![\mathscr{V}]\!] \times A))^* .$$

Now, guarded strings satisfy either the pattern (Γ, α) or the pattern $(\Gamma, \alpha) \cdot p \cdot g$, where Γ is a valuation of the public variables, α is an atom and g is a guarded string.

Public Equality Two guarded strings g_1 and g_2 are equal on their public variables, denoted $g_1 =_p g_2$, if and only if their *initial* valuations are equal on public variables.

We now define a formal notion of constant resource consumption, which we apply to the $\texttt{mem}$ metric.

Definition 6.3.1 (TSCF with MAP)**.** Given a language of guarded strings $L \subseteq \text{GS}$, we say that L has *secure constant time consumption guarantees with respect to* $\texttt{mem}$ if and only if, for all $g_1, g_2 \in L$:

$$g_1 =_p g_2 \Rightarrow \text{RC}_{\texttt{mem}}(g_1) = \text{RC}_{\texttt{mem}}(g_2).$$

Equivalently, we say that L satisfies TSCF.

This definition naturally extends to GKAT expressions. A GKAT expression e satisfies TSCF if and only if $[\![e]\!]$ satisfies TSCF.

Attacker model Definition **??** characterises an attacker model where the attacker that can choose all the values of public variable before execution of every GKAT expression and, at the end of the execution of the GKAT expression, can measure the total number of hits and misses to the cache. Additionally, the attacker can execute Prime+Probe [**prime-probe**] or Flush+Reload [**Flush+Reload**] attacks, either remotely or locally. More precisely, this attacker can see which addresses are loaded in the cache but cannot see their contents, and can flush the cache at will. This attacker is similar to the attacker which exploits a Spectre V1 vulnerability to extract secrets via the cache, but our attacker does not rely on speculative execution (speculative execution is beyond the scope of this work).

In the following sections, we provide a concrete GKAT for an enriched language of while programs. Using this GKAT, we describe existing repair rules used by other repair tools to enforce TSCF under the baseline leakage model. We discuss why these rules are not enough to enforce TSCF under the MAP leakage model, and we propose our own set of rules to fill this gap.

6.4 Repairing TSCF under MAP with ORIGAMI

Existing program repair solutions for TSCF [**Racoon**; **SCEliminator**; **MSESC**] offer high-overhead, unsound guarantees, or no guarantees at all with respect to the MAP leakage model. Racoon [**Racoon**] implements ORAM to enforce TSCF in the MAP leakage model. While secure, the use of ORAM is quite taxing in terms of performance overhead. This overhead is unfortunate, which is why we look for other software/compiler-based alternatives to repair TSCF for the MAP model. SC-ELIMINATOR [**SCEliminator**] uses must-hit analysis and data structure preloading to enforce TSCF in the MAP leakage model. However, the solution presented is unsound when we consider that the attacker can manipulate the cache. More precisely, SC-ELIMINATOR implicitly assumes that the state of the cache during the must-hit analysis is maintained through execution, which is not true since the attacker can flush the cache after data structures have been preloaded, rendering the must-hit assumptions invalid. Finally, the methodology presented in [**MSESC**] only offers security guarantees with respect to the baseline leakage model, and not with respect to the MAP leakage model.

In the following, we present an enriched language of while programs, and we study the possible causes for different MAPs. First, we define a concrete GKAT and give more precise definitions about what their time consumption means. Then, we formally present the ORIGAMI program transformation rules, and we justify their soundness formally; i.e., we prove that they enforce TSCF for the MAP leakage model. Then, given that the MAP model implies the baseline model, we include for completeness the well-known repair rules used to repair branches and loops. We proceed to discuss limitations of enforcement, and finally we show a natural extension of ORIGAMI from arrays to multidimensional fixed-size data structures.

6.4.1 A Concrete GKAT

We want to keep the GKAT as abstract as possible, but we do need to define when a variable hits or misses the cache.

We start small by considering uni-dimensional arrays. Let $\mathscr{V}$ be the set of variables names; let $\mathscr{S}$ be the set of names of arrays. We define the set of *atomic expressions* $\mathscr{A}$ by the grammar

$$\begin{aligned} a \in \mathscr{A} ::= {} & x \in \mathscr{V} \mid n \in \mathbb{N} \mid \\ & \vec{S}[x] \text{ with } \vec{S} \in \mathscr{S}, x \in \mathscr{V} \mid \\ & \vec{S}[n] \text{ with } \vec{S} \in \mathscr{S}, n \in \mathbb{N}. \end{aligned}$$

We define the set of *tests* T by the grammar

$$t \in T ::= \quad a_1 = a_2, \text{ with } a_1, a_2 \in \mathscr{A}.$$

We define the set of *actions* Σ by the grammar

$$\begin{aligned} p \in \Sigma ::= {} & a_1 := a_2 && \text{where } a_1, a_2 \in \mathscr{A} \mid \\ & a := \mathbf{sel}(b, a_1, a_2) && \text{where } a_1, a_2, a_3 \in \mathscr{A}, b \in \mathscr{B} \end{aligned}$$

The semantics of the language is standard, and we omit the details since they are largely irrelevant for the purposes of enforcing TSCF under the MAP leakage model. More precisely, we care when and where we load elements in the cache, independently of the functional semantics of an action. Nevertheless, we assume that the compiler discards expressions that are not well-formed (e.g. $1 := A[x]$). We consider nested expressions to be using syntactic sugar; e.g. $A[x] := B[C[x]]$ is equivalent to

$$x_1 := C[x] \cdot x_2 := B[x_1] \cdot A[x] := x_2.$$

6.4.2 ORIGAMI Rules

It is impossible to repair programs that are not TSCF with traditional GKAT rules. Since GKAT rules preserve semantics, non-TSCF expressions are never transformed into TSCF expressions via these rules. Thus, for the purposes of enforcing TSCF, we require two functional equivalences between actions and GKAT expressions: the *read equivalence* $(y := A[x]) \equiv \mathscr{O}(y := A[x])$ and the *write equivalence* $(A[x] := y) \equiv \mathscr{O}(A[x] := y)$, where

$$\begin{aligned} \mathscr{O}(y := A[x]) \triangleq {} & y := \mathbf{sel}(x = 0, A[0], y) \cdot \\ & y := \mathbf{sel}(x = 1, A[1], y) \cdot \\ & \dots \\ & y := \mathbf{sel}(x = n, A[n], y) \end{aligned}$$

and

$$\begin{aligned}\mathscr{O}(A[x] := y) \triangleq\ & A[0] := \mathbf{sel}(x = 0, y, A[0])\cdot \\ & A[1] := \mathbf{sel}(x = 1, y, A[1])\cdot \\ & \dots \\ & A[n] := \mathbf{sel}(x = n, y, A[n]),\end{aligned}$$

such that A has a fixed size of $n + 1$ and x and y are different variables. The *read ORIGAMI rule* is transforming $(y := A[x])$ into $\mathscr{O}(y := A[x])$, denoted

$$(y := A[x]) \rightsquigarrow\ \mathscr{O}(y := A[x]), \tag{6.2}$$

and the *write ORIGAMI rule* is

$$(A[x] := y) \rightsquigarrow\ \mathscr{O}(A[x] := y). \tag{6.3}$$

Intuitively, the read and write rules are sound: $\mathscr{O}(y := A[x])$ implements an iteration over A, loading every value $A[i]$ but only storing it in y if $i = x$; the program $\mathscr{O}(A[x] := y)$ also implements an iteration, which loads $A[i]$ and stores it back at $A[i]$ if $i \neq x$, or loads $A[i]$ but stores y instead when $i = x$. We provide stronger arguments for the soundness of these rules in Section **??**.

We now provide an argument for why these rules are sound with respect to TSCF. We assume that $0 \leq x \leq n$, which we consider a reasonable assumption since in many languages the semantics of the expression $A[x]$ where $x > n$ is undefined or triggers an exception. To show why $\mathscr{O}(y := A[x])$ is TSCF with respect to the MAP leakage, we do a proof by induction on the size of A.

Theorem 6.1. Let A be an array-like data structure of size $n + 1$ with $n \in \mathbb{N}$, then $[\![\mathscr{O}(y := A[x])]\!]$ satisfies TSCF with MAP.

Proof. The intuition behind the proof is the following: this ORIGAMI rule expands a single access to the data structure A into a constant sequence of accesses to A with fixed parameters, folding a **sel** instruction to accumulate the value that would be computed by the access $A[x]$; since the new sequence of accesses to A is independent of x, the MAP of $\mathscr{O}(y := A[x])$ does not leak x.

To prove that $[\![\mathscr{O}(y := A[x])]\!]$ satisfies TSCF, we do a proof by induction. For $n = 0$, $\mathscr{O}(y := A[x])$ is equal to $y := \mathbf{sel}(x = 0, A[x], y)$. Now, since $\mathtt{mem}$ only depends on the actions of GSs and not on its atoms, we can take the symbolic GS

$$g = \alpha \cdot (y := \mathbf{sel}(x = 0, A[x], y)) \cdot \beta,$$

where α and β are symbolic variables for atoms, and compute $\mathrm{RC}_{\mathtt{mem}}(g)$. The value of $\mathrm{RC}_{\mathtt{mem}}(g)$ in the base case where $n = 0$ is

$$\begin{aligned}\mathrm{RC}_{\mathtt{mem}}(g) &= \mathtt{mem}(\alpha)(y := \mathbf{sel}(x = 0, A[x], y)) \\ &= [\eta \times \{y, x, A[0]\} + \mu \times \emptyset].\end{aligned}$$

In the inductive case where $n > 0$, we have

$$\begin{aligned}\mathrm{RC}_{\mathtt{mem}}(g) = [&\eta \times \{y, x, A[0]\} + \mu \times \emptyset,\\ &\eta \times \{A[1]\} + \mu \times \{y, x\}\,,\\ &\eta \times \{A[2]\} + \mu \times \{y, x\}\,,\\ &\dots\\ &\eta \times \{A[n]\} + \mu \times \{y, x\}].\end{aligned}$$

We remark that if the attacker clears the cache before the i-th instruction $y := \mathbf{sel}(x = i+1, A[x], y)$, then the MAP changes from $\eta \times \{A[2]\} + \mu \times \{y, x\}$ to $\eta \times \{y, x, A[i]\} + \mu \times \emptyset$, as if it were a base case; the secret x does not leak because we are loading $A[i]$ and not $A[x]$, and the attacker cannot decide whether $i = x$ (the attacker can only observe whether i and x are loaded, but not their values). We conclude that the MAP is indistinguishable for all values of x, and only depends on the size of A (i.e., $n + 1$).

□

A similar argument to Theorem **??** can be made for rules of the form $\mathscr{O}(A[x] := y)$.

Theorem 6.2. Let A be an array-like data structure of size $n + 1$ with $n \in \mathbb{N}$, then $[\![\mathscr{O}(A[x] := y)]\!]$ satisfies TSCF with MAP.

Proof. This ORIGAMI rule also expands the single access $A[x]$ into a constant sequence of accesses to A with fixed parameters, but this time it loads from every position of A, and it stores a value chosen via a **sel**; again, since this new sequence of accesses to A is independent of x, the MAP of $\mathscr{O}(A[x] := y)$ does not leak x.

To prove that $[\![\mathscr{O}(A[x] := y)]\!]$ satisfies TSCF, we do a proof by induction. For $n = 0$, $\mathscr{O}(A[x] := y)$ is equal to $A[0] := \mathbf{sel}(x = 0, y, A[x])$. We take symbolic GS $g = \alpha \cdot A[0] := \mathbf{sel}(x = 0, y, A[x]) \cdot \beta$, where α and β are symbolic variables for atoms, and we compute $\mathrm{RC}_{\mathtt{mem}}(g)$ in the base case where $n = 0$ as follows:

$$\mathrm{RC}_{\mathtt{mem}}(g) = \mathtt{mem}(\alpha)(A[0] := \mathbf{sel}(x = 0, y, A[0]))$$

which is given by the sequence

$$\mathrm{RC}_{\mathtt{mem}}(g) = [\eta \times \{x, y, A[0]\} + \mu \times \{A[0]\}].$$

In the inductive case where $n > 0$, we have

$$\begin{aligned}\mathrm{RC}_{\mathtt{mem}}(g) = [&\eta \times \{x, y, A[0]\} + \mu \times \{A[0]\}\,,\\ &\eta \times \{A[1]\} + \mu \times \{x, y, A[1]\}\,,\\ &\eta \times \{A[2]\} + \mu \times \{x, y, A[2]\}\,,\\ &\dots\\ &\eta \times \{A[n]\} + \mu \times \{x, y, A[n]\}]\end{aligned}$$

If the attacker clears the cache before the i-th instruction $A[i] := \mathbf{sel}(x = i, y, A[i])$, then the MAP changes from $\eta \times \{A[i]\} + \mu \times \{x, y, A[i]\}$ to $\eta \times \{x, y, A[i]\} + \mu \times \{A[i]\}$, as if

it were a base case. We again conclude that the MAP is indistinguishable for all values of x, and only depends on the size of A

□

6.5 Repair Rules for the Baseline Model

Since the MAP implies the baseline model, we must repair programs also with respect to the baseline model. The baseline repair rules are well known, but we briefly mention them here for completeness. Existing repair tools, including [**Racoon**; **SCEliminator**; **MSESC**], are in general capable of repairing programs with respect to the baseline leakage model using the following repair rules, whose basic strategy consists of removing branches via predication and by unrolling loops once they have been fixed to have a constant number of iterations.

Branch predication Given a program of the form

$$\text{if } b \text{ then } x := e_1 \text{ else } x := e_2$$

whose corresponding GKAT expression is $(x := e_1) +_b (x := e_2)$, the *predication repair rule* replaces the branch by the select instruction $x := \mathbf{sel}(b, e_1, e_2)$. The tools [**Racoon**; **SCEliminator**; **MSESC**] implement this rule to remove leaks due to branching. There are a couple of minor complications; repairing the GKAT expression $(a := b/c) +_{c \neq 0} 1$ could introduce runtime exceptions that were previously prevented by the conditional (e.g. division by zero or accessing an array out of bounds). However, it suffices to take a couple extra precautions before repair, as shown in [**Racoon**] and [**MSESC**].

Loop Unrolling Given a program of the form

$$\text{while } b \text{ do } e$$

which corresponds to the GKAT expression $e^{(b)}$, the *loop unrolling rule* replaces the program with a sequence of linearisable branches

$$\underbrace{(e +_b 1) \cdot (e +_b 1) \cdot \ldots \cdot (e +_b 1)}_{k \text{ times}}.$$

where k is the maximum number of iterations that the loop could execute for all secrets.

The major complication to apply the loop unrolling rule is determining k. Racoon does not protect information leaks from loop trip counts [**Racoon**], so they cannot repair programs with loops whose trip count is originally secret dependent. `SC-eliminator` arbitrarily chooses a value from a list of fixed sizes (e.g. 64, 128, 256) depending on static analysis, [**MSESC**] assumes that loops are already unrolled, and [**Racoon**] do not protect against information leaks from loop trip counts.

Limitations of Predication and Unrolling Neither predication nor unrolling repair programs with respect to the MAP leakage model. The loopless and branchless program $x := A[s]$ where s is a secret and A is a data structure has different memory access patterns for the different values of s; this limitation is the main motivation for ORIGAMI.

Vulnerable to OS Leakage Since the MAP leakage model is weaker than the OS leakage model, ORIGAMI does not repair timing side-channel vulnerabilities that arise by the use of operations whose execution time varies depending on their parameters. We only repair leakage that occurs due to accesses to data structures using secret indices.

6.6 Limitations of ORIGAMI

There are a couple of limitations and side effects to applying the ORIGAMI rules.

Secret Pointers While we can fold a data structure whose data are secret pointers without revealing the value of the resulting pointer or the value used to access it, once this resulting pointer is loaded, it is leaked to the attacker via the cache. Thus, ORIGAMI should not be used to repair programs which contain secret pointers that are naively accessed.

Secret Data Structure Sizes The MAP of $\mathscr{O}(y := A[x])$ and of $\mathscr{O}(A[x] := y)$ depends on the size of A. This adds a caveat for using ORIGAMI to enforce TSCF under MAP: the size of data structures indexed by secrets must be public, since the size of A can be inferred via timing. We believe this caveat is acceptable, since several cryptographic algorithms fix the size of the data structures that hold secrets (e.g. AES has key sizes 128, 192 or 256 bits). We also believe this case is dual to trying to repair a loop by unrolling it if the loop bound depends on an unbounded secret: it is impossible to repair without compromising its functionality[**SCEliminator**; **MSESC**].

Out-of-bounds Accesses As a side-effect of applying the ORIGAMI rules, accessing a data structure A of size n using a sensitive index x such that $x > n$ no longer results in a runtime error. The repaired version of $y := A[x]$ simply casues y to keep its current value if $x > n$. Similarly, a repaired version of $A[x] := y$ where $x > n$ never stores y; it simply loads all $A[i]$ and writes them back. Consequently, these new behaviours leak to the attacker the information $x > n$, assuming that n is public. This limitation is arguably artificial, since the original program possibly also leaks this information through a runtime error.

Vulnerabilities Beyond MAP As we previously stated, ORIGAMI does not offer security guarantees for programs that are vulnerable due to speculative execution or that are vulnerable in the OS leakage model, since these have leakage and attacker models that are beyond the scope of this work.

6.7 Multidimensional ORIGAMI

So far, we presented and illustrated ORIGAMI using array-like data structures A of fixed size n. However, ORIGAMI can be applied to repair programs that use any iterable structure of fixed size. These data structures include, e.g., C structures and multidimensional arrays.

For example, we can apply ORIGAMI to a fixed-size multidimensional structure A of size $(n+1) \times (m+1)$ by systematically folding all of its $(n+1) \times (m+1)$ elements. More precisely, to repair $y := A[i][j]$, we compute

$$\begin{aligned}\mathscr{O}(y := A[i][j]) \triangleq\ & y := \mathbf{sel}(i = 0 \wedge j = 0, A[0][0], y)\cdot \\ & y := \mathbf{sel}(i = 0 \wedge j = 1, A[0][1], y)\cdot \\ & \dots \\ & y := \mathbf{sel}(i = 0 \wedge j = m, A[0][m], y)\cdot \\ & y := \mathbf{sel}(i = 1 \wedge j = 0, A[1][0], y)\cdot \\ & \dots \\ & y := \mathbf{sel}(i = 1 \wedge j = m, A[1][m], y)\cdot \\ & \dots \\ & y := \mathbf{sel}(i = n \wedge j = m, A[n][m], y).\end{aligned}$$

Similarly, to repair $A[i][j] := y$, we compute

$$\begin{aligned}\mathscr{O}(A[i][j] := y) \triangleq\ & A[0][0] := \mathbf{sel}(i = 0 \wedge j = 0, y, A[0][0])\cdot \\ & A[0][1] := \mathbf{sel}(i = 0 \wedge j = 1, y, A[0][1])\cdot \\ & \dots \\ & A[0][m] := \mathbf{sel}(i = 0 \wedge j = m, y, A[0][m])\cdot \\ & A[1][0] := \mathbf{sel}(i = 1 \wedge j = 0, y, A[1][0])\cdot \\ & \dots \\ & A[1][m] := \mathbf{sel}(i = 1 \wedge j = m, y, A[1][m)\cdot \\ & \dots \\ & A[n][m] := \mathbf{sel}(i = n \wedge j = m, y, A[n][m]).\end{aligned}$$

For higher dimensions the ORIGAMI rules extend similarly, and they can be customised if some indices are secrets and some are not. Informally, we only need to fold those indices that hold secrets, e.g., if a read access $y := A[i][k]$ uses a secret i and a public value k, it suffices to fold as follows:

$$\begin{aligned}\mathscr{O}(y := A[i][k]) \triangleq\ & y := \mathbf{sel}(i = 0, A[0][k], y)\cdot \\ & y := \mathbf{sel}(i = 1, A[1][k], y)\cdot \\ & \dots \\ & y := \mathbf{sel}(i = n, A[n][k], y)\end{aligned}$$

We can use the index k directly because it is public, i.e., if it leaks via the cache, the attacker does not learn any new information about the secret i.

6.8 ORIGAMI Rules as Spatial Transformations

In Section **??**, we provide formal evidence that the ORIGAMI rules enforce TSCF with respect to the MAP leakage model. Now, in this section, we provide formal evidence that the ORIGAMI rules preserve the functionality of the programs that they transform by modelling the ORIGAMI rules as spatial transformations in an F-coalgebra. While the preservation of functionality via ORIGAMI rules is not unexpected, our analysis shows how functionality may differ in corner cases, and, in particular, which assumptions we must make to preserve the functionality of programs with expressions $x := A[s]$ and $A[s] := x$ when s is an index out of bounds.

We start by summarising the MAP leakage model with the following set of leakage assumptions:

- for any expression of the form $e_1 +_b e_2$, we assume that it leaks information if b depends on a secret,
- for any expression of the form $e^{(b)}$, we assume that it leaks information if b depends on a secret,
- for any expression of the form $x := A[s]$ or $A[s] := x$, we assume that it leaks information if s depends on a secret,
- if an expression e leaks information, then the expressions $e \cdot e'$, $e +_b e'$, $e^{(b)}$, and their equivalent expressions leak information for any other expression e' if $b \neq 0$.

The assumptions of the MAP model correspond to attacks that are practical. An expression of the form $e_1 +_b e_2$ leaks information about b via the program counter PC; an attacker observing PC can deduce whether b is true or false based on the instruction that follows the evaluation of b. The reasoning for e^b is similar: the program counter leaks b, and the attacker may even obtain more information based on how long it took for the loop to execute. The expressions $x := A[s]$ and $A[s] := x$ leak s via the cache because an attacker can flush the cache before their execution, and then probe the cache after their execution; the regions that result in a hit correspond to the neighbourhood of $A[s]$, meaning that s can be inferred with high probability from this analysis. The last assumption concerns the safety nature of confidentiality leaks: once a secret has been leaked, it cannot be recovered.

Given the assumptions of the MAP leakage model, we approach the problem of repairing TSCF by treating it as a safety enforcement problem, which we solve by replacing all dangerous actions in programs without altering their functional semantics. In the following, we show how to enforce TSCF by using spatial transformations in a coalgebraic specification language of concrete GKAT expressions.

6.8.1 A GKAT Coalgebraic Specification Language

In the following, we summarise the method used by the authors of **[GKAT]** to give coalgebraic semantics to GKAT expressions. Consider the functor $G(X) = (2 + \Sigma \times X)^A$. A G-coalgebra (X, c) has the following operational semantics: for $x \in X$, x receives an atom $\alpha \in A$, which represents a precondition, and either terminates well, terminates badly, or continues executing the next action. Formally, if $c(x)(\alpha) = \iota_1(0)$, then x terminates badly if α holds at x; if $c(x)(\alpha) = \iota_1(1)$, then x terminates well if α holds at x; finally, if $c(x)(\alpha) = \iota_2(p, y)$, then x transitions to state y, during the execution of action p.

Let A^+ be the set of non-empty finite sequences of atoms; we define the set σG, a subset of $(2 + \Sigma)^{A^+}$ as follows: for $\phi \in (2 + \Sigma)^{A^+}$, $\phi \in \sigma G$ iff $\phi(\alpha) = \iota_1(b)$ implies that $\phi(\alpha \cdot \omega) = \iota_1(b)$ for all $\alpha \in A$, $\omega \in A^*$, and $b \in 2$. The expression $\phi(\alpha) = \iota_1(b)$ models good or bad termination depending on whether $b = 1$ or $b = 0$ respectively, assuming that the last state satisfies the atom α.

The final G-coalgebra is $(\sigma G, 1)$, whose final map $1 \colon \sigma G \to (A \to (2 + \Sigma \times \sigma G))$ is defined for $\phi \in \sigma G$, $\alpha \in A$ and ω^+ by

$$1(\phi)(\alpha) \triangleq \begin{cases} \iota_1(b), & \text{if } \phi(\alpha) = \iota_1(b), \\ \iota_2(p, \phi^\alpha), & \text{if } \phi(\alpha) = \iota_2(p) \end{cases} \tag{6.4}$$

$$\text{where} \quad \phi^\alpha(\omega) \triangleq \phi(\alpha \cdot \omega). \tag{6.5}$$

A coalgebraic specification language for the functor G that uses the set of GKAT expressions as its carrier would, given an expression $e \in \mathscr{E}$ and a precondition modelled by an atom $\alpha \in A$, state via $\delta(e)(\alpha)$ whether we finish well (i.e.,$\delta(e)(\alpha) = \iota_1(1)$), we finish badly (i.e.,$\delta(e)(\alpha) = \iota_1(0)$), or we continue execution with an action $p \in \Sigma$, yielding a remainder program e' (i.e.,$\delta(e)(\alpha) = \iota_2(p, e')$)). The rules shown in **[GKATCoequations]** define a G-coalgebra $(\mathscr{E}, \delta)$ that implements a coalgebraic specification language. We compile them here for completeness. First, let us introduce some notation; for $e \in \mathscr{E}$, $\alpha \in A$,

- the expression $e \uparrow \alpha$ denotes $\delta(e)(\alpha) = \iota_1(1)$; i.e. if the atom α holds as a precondition for e, then e terminates well as its next step;
- the expression $e \downarrow \alpha$ denotes $\delta(e)(\alpha) = \iota_1(0)$; i.e. if the atom α holds as a precondition for e, then e terminates badly as its next step;
- the expression $e \xrightarrow{\alpha|p} e'$ represents $\delta(e)(\alpha) = \iota_2(p, e')$; i.e., if α holds before the execution of e, then the next step is to execute the action p and continue execution from the expression e'.

We explain each of the inference rules: the rule

$$\frac{b \in \text{BExp} \qquad \alpha \in A \qquad \alpha \Rightarrow b}{b \uparrow \alpha} \tag{6.6}$$

means that the program **assert** b ends well if its precondition α implies the assertion.

The rule

$$\frac{p \in \Sigma \qquad \alpha \in A}{p \xrightarrow{\alpha|p} 1} \tag{6.7}$$

states that the program p can always continue to the program **assert true** after executing the action p.

The next four rules describe the dynamics of branch expressions. The rule

$$\frac{b \in \text{BExp} \qquad \alpha \in A \qquad \alpha \Rightarrow b \qquad f \uparrow \alpha}{f +_b g \uparrow \alpha} \tag{6.8}$$

means that the program $f +_b g$ ends well if its precondition α implies the branch condition b and the expression f ends well. It is dual to the rule

$$\frac{b \in \text{BExp} \qquad \alpha \in A \qquad \alpha \Rightarrow \bar{b} \qquad g \uparrow \alpha}{f +_b g \uparrow \alpha}. \tag{6.9}$$

The following rule

$$\frac{b \in \text{BExp} \qquad \alpha \in A \qquad p \in \Sigma \qquad \alpha \Rightarrow b \qquad f \xrightarrow{\alpha|p} f'}{f +_b g \xrightarrow{\alpha|p} f'} \tag{6.10}$$

states that the program $f +_b g$ continues by executing the action p if the precondition α satisfies the branch guard b, and the program f continues by executing the action p. The dual of this rule is

$$\frac{b \in \text{BExp} \qquad \alpha \in A \qquad p \in \Sigma \qquad \alpha \Rightarrow \bar{b} \qquad g \xrightarrow{\alpha|p} g'}{f +_b g \xrightarrow{\alpha|p} f'}. \tag{6.11}$$

The following four rules describe the behaviour of composite expressions. The rule

$$\frac{f \Rightarrow \alpha \qquad g \Rightarrow \alpha}{f \cdot g \uparrow \alpha} \tag{6.12}$$

states that if f would terminate well given α and g would terminate well given α, then $f \cdot g$ terminates well given α. The next rule is similar,

$$\frac{f \Rightarrow \alpha \qquad g \xrightarrow{\alpha|p} g'}{f \cdot g \xrightarrow{\alpha|p} g'} \tag{6.13}$$

means that f would terminate well given α and g continues by executing p given α, then $f \cdot g$ continues by executing p given α, and then continues with the execution of g'.

The rule

$$\frac{f \xrightarrow{\alpha|p} f'}{f \cdot g \xrightarrow{\alpha|p} f' \cdot g} \tag{6.14}$$

means that if f continues by executing p given α, then $f \cdot g$ continues by executing p given α and then continue with the execution of f', delaying g.

The rule

$$\frac{f \xrightarrow{\alpha|p} f'}{f \cdot g \xrightarrow{\alpha|p} f' \cdot g} \tag{6.15}$$

means that if f continues by executing p given α, then $f \cdot g$ continues by executing p given α and then continue with the execution of f', delaying g.

The last two rules address the evolution of loops. The rule

$$\frac{b \in \text{BExp} \qquad \alpha \Rightarrow b \qquad f \xrightarrow{\alpha|p} f'}{f^{(b)} \xrightarrow{\alpha|p} f' \cdot bf} \tag{6.16}$$

states that if f continues by executing p given α, and α satisfies the loop condition b, then $f^{(b)}$ continues by executing p given α to then continue with the execution of f', delaying a new iteration $f^{(b)}$.

Finally, the rule

$$\frac{b \in \text{BExp} \qquad \alpha \Rightarrow \bar{b}}{f^{(b)} \uparrow \alpha} \tag{6.17}$$

means that the program $f^{(b)}$ finishes correctly given α if α implies the negation of the loop condition b.

We conclude this exposition by stating that if an expression $e \in$ does not satisfy any of the inference rules presented above, then e terminates badly for all $\alpha \in A$; i.e., $\delta(e)(\alpha) = \iota_1(0)$.

6.8.2 Transforming GKAT expressions to enforce TSCF

Now that we have a dynamics function that defines how GKAT expressions evolve during computations in the form of the G-coalgebra $(\mathscr{E}, \delta)$, we enforce TSCF by applying a spatial transformations which implement the branch predication, loop unrolling and the read and write ORIGAMI rules. Through this section, we assume that the equivalence $(v := v) \equiv 1$ is valid for all $v \in \mathscr{V}$.

Branch Predication. The branch predication rule $\mathscr{B}\colon \mathscr{E} \to \mathscr{E}$ maps $(x := e_1) +_b (x := e_2)$ to $x := \mathbf{sel}(b, e_1, e_2)$; this causes a change of dynamics from

$$\delta((x := e_1) +_b (x := e_2))(\alpha) = \begin{cases} \iota_2(x := e_1, 1), & \text{if } \alpha \Rightarrow b \\ \iota_2(x := e_2, 1), & \text{if } \alpha \Rightarrow \bar{b} \end{cases}$$

to

$$\delta(x := \mathbf{sel}(b, e_1, e_2))(\alpha) = \iota_2(x := \mathbf{sel}(b, e_1, e_2), 1).$$

The functional semantics are preserved if $\iota_2(x := e_1, 1)$ is equal to $\iota_2(x := \mathbf{sel}(b, e_1, e_2), 1)$ when α implies b, and if $\iota_2(x := e_2, 1)$ is equal to $\iota_2(x := \mathbf{sel}(b, e_1, e_2), 1)$ when α implies $\bar{b}$. Fortunately, the concrete semantics of the action $x := \mathbf{sel}(b, e_1, e_2)$ is precisely given by those conditions. We conclude that the functionality of the program $(x := e_1) +_b (x := e_2)$ is not affected by the transformation rule $\mathscr{B}$. By applying $\mathscr{B}$, we remove the problematic branch while preserving the functional semantics.

If a branch is of the form $(x := e_1) +_b (y := e_2)$, given the equivalence $v := v \equiv 1$, we can rewrite it as

$$(x := e_1 \cdot y := y) +_b (x := x \cdot y := e_2)$$

and the rule $\mathscr{B}$ maps $(x := e_1) +_b (y := e_2)$ to

$$x := \mathbf{sel}(b^\star, e_1, x) \cdot y := \mathbf{sel}(b^\star, y, e_2). \tag{6.18}$$

where $b^\star$ is the value of b before any **sel** instructions are executed.

Loop Unrolling. The loop unrolling rule $\mathscr{L}\colon \mathscr{E} \to \mathscr{E}$ transforms $e^{(b)}$ into the GKAT expression

$$\underbrace{(e +_b 1) \cdot (e +_b 1) \cdot \ldots \cdot (e +_b 1)}_{k \text{ times}},$$

where k is the maximum number of iterations of the loop. The transformation $\mathscr{L}$ preserves functional semantics under an assumption: by combining the guarded loop axiom $(e +_c 1)^{(b)} \equiv (ce)^{(b)}$ (see **[GKAT]**) and the guarded iteration fact $(be)^{(b)} \equiv e^{(b)}$ (see **[GKAT]**), we obtain $(e +_b 1)^{(b)} \equiv e^{(b)}$, which we unfold k times into

$$\underbrace{(e +_b 1) \cdot (e +_b 1) \cdot \ldots \cdot (e +_b 1)}_{k \text{ times}},$$

under the assumption that the loop iterates at most k times. Only if the assumption that the loop iterates at most k times is satisfied, then $\delta(e^{(b)}) = \delta(\mathscr{L}(e^{(b)}))$. More precisely, for $0 \leq j \leq k$, if $\bar{b}$ is true after j iterations, the program $e +_b 1$ is equivalent to 1 (by the inference rule in Equation **??**), and it does not alter the result of the computation; however, if k is poorly computed, and b holds after k iterations, then the semantics of $e^{(b)}$ and $\mathscr{L}(e^{(b)})$ are no longer equal in general.

The ORIGAMI read and write rules $\mathscr{O}\colon \mathscr{E} \to \mathscr{E}$ map $x := A[s]$ and $A[s] := x$ respectively to $\mathscr{O}(x := A[s])$ and $\mathscr{O}(A[s] := x)$ following the definitions in Section **??**. To prove that $\mathscr{O}$ preserves the functional semantics, we need to show both that $\delta(A[s] := x) = \delta(\mathscr{O}(A[s] := x))$ and that $\delta(x := A[s]) = \delta(\mathscr{O}(x := A[s]))$.

ORIGAMI read rule. We define the GKAT expression $\mathbf{rFold}(x, A, s, i..n)$, for $0 \leq i \leq n$, by

$$\begin{aligned}\mathbf{rFold}(x, A, s, i..n) \triangleq & x := \mathbf{sel}(s = i, A[i], x)\cdot \\ & x := \mathbf{sel}(s = i+1, A[i+1], x)\cdot \\ & \cdots \\ & x := \mathbf{sel}(s = n, A[n], x),\end{aligned}$$

where s is a variable different from x.

Our goal now is to justify the equivalence

$$\begin{aligned}x := A[s] &\equiv \mathscr{O}(x := A[s]) \\ &\equiv \mathbf{rFold}(x, A, s, 0..n).\end{aligned}$$

by means of coinduction; i.e., by showing for $\alpha \in A$ that $\delta(\mathbf{rFold}(x, A, s, i..n))(\alpha) = \delta(x := A[s])(\alpha)$. The equality $\delta(x := A[s]) = \delta(\mathbf{rFold}(x, A, s, 0..n))$ is not obvious at first, since we are introducing more actions when we apply $\mathscr{O}$. We must consider the concrete semantics of actions again, just as we did when defining the equivalence $x := \mathbf{sel}(b, e_1, e_2) \equiv (x := e_1) +_b (x := e_2)$ for the branch predication rule.

For $\alpha \in A$ and $0 \leq i \leq n$, we have two cases: when $\alpha \Rightarrow s = i$ and when $\alpha \not\Rightarrow s = i$. If $\alpha \Rightarrow s = i$ then the next action of $\mathbf{rFold}(x, A, s, i..n)$ is $x := \mathbf{sel}(s = i, A[i], x)$, which is equivalent to $x := A[s]$ since $s = i$ given α. If $\alpha \not\Rightarrow s = i$, then the next action of $\mathbf{rFold}(x, A, s, i..n)$ is $x := \mathbf{sel}(s = i, A[i], x)$, which is equivalent to $x := x$ and to 1. Now, by the inference rule in Equation **??**, we know that $\delta(1\cdot e)(\alpha) = \delta(e)(\alpha)$; i.e., $\delta(e)(\alpha)$ skips trivial actions. Now, at most one of the conditions $\alpha \Rightarrow s = i$ is satisfied; for such an index i, the next action of $\mathbf{rFold}(x, A, s, i..n)$ is $x := A[i]$. In the case when no condition $\alpha \Rightarrow s = i$ is satisfied, i.e. when $s < 0$ or $s > n$, the program $\mathbf{rFold}(x, A, s, i..n)$ simply terminates well. The semantics of $x := A[s]$ is normally undefined when $s < 0$ or $s > n$, unlike the semantics of $\mathbf{rFold}(x, A, s, 0..n)$ to terminate well. Therefore, we must assume the equivalence $x := A[s] \equiv 1$ when $s < 0$ or $s > n$ for the equivalence $x := A[s] \equiv \mathbf{rFold}(x, A, s, 0..n)$ to be sound with respect to functionality; otherwise, we face the *out-of-bounds accesses* limitation from Section **??** (we cannot assume $x := A[s] \equiv 1$ for C/C++ programs during out-of-bounds accesses, which is why the ORIGAMI tool has this limitation).

In summary, we have shown that for $\alpha \in A$,

$$\delta(\mathbf{rFold}(x, A, s, 0..n))(\alpha) = \begin{cases}\iota_2(x := A[0], \mathbf{rFold}(x, A, s, 1..n)), & \text{if } \alpha \Rightarrow s = 0, \\ \iota_2(1, \mathbf{rFold}(x, A, s, 1..n)), & \text{otherwise.}\end{cases}$$

and

$$\delta(x := A[s])(\alpha) = \iota_2(x := A[s], 1)$$

describe the same dynamics, which is why we believe the equivalence $x := A[s] \equiv$ **rFold**$(x, A, s, 0..n)$ is justified. If $x := A[s]$ is equivalent to **rFold**$(x, A, s, 0..n)$, and $\mathscr{O}(x := A[s]) =$ **rFold**$(x, A, s, 0..n)$, we conclude that $\mathscr{O}$ does not affect the functionality of the action $x := A[s]$.

ORIGAMI write rule. Consider the GKAT expression **wFold**$(x, A, s, i..n))$, defined for $0 \leq i \leq n$ by

$$\begin{aligned}\textbf{wFold}(x, A, s, i..n) \triangleq & A[i] := \textbf{sel}(s = i, x, A[i]) \cdot \\ & A[i+1] := \textbf{sel}(s = i+1, x, A[i+1]) \cdot \\ & \dots \\ & A[n] := \textbf{sel}(s = n, x, A[n]),\end{aligned}$$

where s is a variable different from x.

We want to show that the equivalence

$$\begin{aligned}A[s] := x &\equiv \mathscr{O}(x := A[s])) \\ &\equiv \textbf{wFold}(x, A, s, 0..n),\end{aligned}$$

is justified. We follow a similar approach than the one we used for the read rule: by showing that $\delta(A[s] := x) = \delta(\textbf{wFold}(x, A, s, 0..n))$, we use coinduction to justify $A[s] := x \equiv \mathscr{O}(x := A[s]))$.

For $\alpha \in A$ and $0 \leq i \leq n$, we have two cases: when $\alpha \Rightarrow s = i$ and when $\alpha \not\Rightarrow s = i$. If $\alpha \Rightarrow s = i$ then the next action of **wFold**$(x, A, s, i..n)$ is $A[i] :=$ **sel**$(s = i, x, A[i])$, which is equivalent to $A[s] := x$. If $\alpha \not\Rightarrow s = i$, then the next action of **wFold**$(x, A, s, i..n)$ is $A[i] :=$ **sel**$(s = i, x, A[i])$, which is equivalent to $A[i] := A[i]$, which in turn is equivalent to 1. We use again the inference rule in Equation **??** to derive $\delta(1 \cdot e)(\alpha) = \delta(e)(\alpha)$. Similarly to the read rule, at most one of the conditions $\alpha \Rightarrow s = i$ is satisfied, and for such index i, the next action of **wFold**$(x, A, s, i..n)$ is $A[i] := x$. Dually, in the case when no condition $\alpha \Rightarrow s = i$ is satisfied, i.e. when $s < 0$ or $s > n$, the program **wFold**$(x, A, s, i..n)$ simply terminates well. In the out-of-bounds case when $s < 0$ or $s > n$, the semantics of $A[s] := x$ is again undefined while the semantics of **wFold**$(x, A, s, 0..n)$ to terminate well. Therefore, we must assume the equivalence $A[s] := x \equiv 1$ when $s < 0$ or $s > n$ for the equivalence $A[s] := x \equiv$ **wFold**$(x, A, s, 0..n)$ to be sound with respect to functionality.

To summarise, we have shown that for $\alpha \in A$,

$$\delta(\textbf{wFold}(x, A, s, 0..n))(\alpha) = \begin{cases} \iota_2(A[0] := x, \textbf{wFold}(x, A, s, 1..n)), & \text{if } \alpha \Rightarrow s = 0, \\ \iota_2(1, \textbf{wFold}(x, A, s, 1..n)), & \text{otherwise.} \end{cases}$$

and

$$\delta(A[s] := x)(\alpha) = \iota_2(A[s] := x, 1)$$

describe the same dynamics, which is why we believe the equivalence $A[s] := x \equiv$ **wFold**$(x, A, s, 0..n)$ is justified, implying that the ORIGAMI write rule preserves the functionality of the program.

We conclude this section by stating that the final spatial transformation that implements the repair for TSCF under the MAP model is the composition $\mathscr{O} \circ \mathscr{B} \circ \mathscr{L}$; we first unfold loops, then we predicate branches, and finally we replace dangerous accesses to data structures by switching over the indices of the data structure or by folding the data structure using ORIGAMI.

6.9 Evaluation

Three research questions motivate an empirical evaluation:

RQ??.1 How effectively can we enforce TSCF? In other words, which metrics can we use to show that programs are actually repaired by ORIGAMI?

RQ??.2 How efficiently can we enforce TSCF? More precisely, what is the overhead incurred by the application of the ORIGAMI repair rules.

RQ??.3 How do our enforcement results vary when compiler optimizations are considered?

We divide the evaluation of ORIGAMI in two sections: 1), a simple litmus tests written in C which illustrates what to expect when repairing programs with ORIGAMI, and 2) a set of five test programs from representative libraries (also written in C): we test two cryptographic libraries (AES from basic crypto [**AESBasic**], DES and RC4 from OpenSSL [**OpenSSL**]), the functions `gdk-keyname` and `gdk-keyuni` from the GDK Linux library.

6.9.1 Implementation

We implement the ORIGAMI repair rules presented in Section **??** using LLVM (version 13) as optimisation passes that we can apply with the LLVM `opt` tool. With this, we can let Clang perform code optimisations to the C programs, and then we apply the ORIGAMI rules, so that they are not undone by further code optimisation.

We rely on annotations to inform ORIGAMI which variables are secret. We provide the details of the tainting procedure in Section **??**.

The compilation chain is as follows: using `clang` with either the `-O0` or the `-O3` optimisation flag, we create an intermediate representation in LLVM-IR. This IR representation has been already optimised by `clang`, so we can apply the ORIGAMI rules without fearing them being optimised out. Finally, to obtain an executable, we use `llc`

with disabled optimizations (i.e., using the `-O0` flag) to compile and link the repaired LLVM-IR into the repaired executable. This is arguably the weakest link in our compilation chain, but our experiments evidence that ORIGAMI rules are preserved by this final compilation step.

6.9.2 Experimental Setup

To evaluate the ORIGAMI repair rules, we use Gem5 [**gem5**]. Gem5 is a highly-configurable simulator which encompasses system-level architecture and processor microarchitecture. The setup for the simulated environment in Gem5 consists of a single simple timing CPU with a 1Gz processor, possessing only a level one 2-set associative cache with a cache-line size of 64 bytes, the data cache has size 8kB and the instruction cache has size 16kB and is 2-set associative.

Gem5 provides statistical data of the execution, including the number of CPU cycles, the number of committed instructions, and the number of hits and misses on both data and instruction caches (among other metrics). We measure the variance of these metrics to evaluate RQ**??**.1. We use the growth factor $m_{repaired}/m_{original}$ of each metric m to evaluate RQ**??**.2. Finally, to address RQ**??**.3, we repair executables compiled with `O0` and with `O3` to see if there are any significant differences.

For each benchmark, we generate a set of five random secret inputs. We use these five instances of the secret to obtain the following metrics: the number of CPU cycles, the number of committed instructions, and the number of hits and misses on both data and instruction caches. An implementation is *vulnerable with respect to the baseline leakage model* if it shows variance in the numbers of committed instructions. An implementation is *vulnerable with respect to the MAP leakage model* if it shows variance in the numbers of cache hits or misses. Finally, an implementation is *vulnerable with respect to the OS leakage model* if it shows variance in the numbers of CPU cycles.

We say that the ORIGAMI rules are effective at repairing programs with respect to the MAP leakage model if for all original programs that are vulnerable with respect to MAP, they are no longer vulnerable with respect to MAP after repair. We remark that ORIGAMI does not repair with respect to the OS leakage model, so we do not expect the variance of CPU cycles in repaired programs to be zero.

6.9.3 About taint analysis

We follow the philosophy presented by the authors of [**WhatYouCisWhatYouGet**], where the programmer collaborates with the compiler to enforce TSCF. Thus, the programmer must provide the information that the compiler cannot derive on its own. Unfortunately, we did not find any reliable tool to perform taint propagation and analysis from the source language C to LLVM-IR, which is why we implement one.

The programmer needs to do the following annotations to the source code: 1) annotate secret variables at their declaration, 2) annotate sensitive loops by giving them a maximum *constant integer* loop bound, and 3) annotate functions that manipulate secrets such that the compiler does not inline. For the latter, the programmer uses the annotation

`__attribute__((noinline))`.

To implement taint propagation from C to LLVM-IR, we use the annotation

`__attribute__(annotate("secret"))`

to mark variables that contain secrets. We refer to these annotated variables as *taint sources*. At the IR level, we identify the taint sources, and we propagate the taint using both data and control flow dependencies, which we obtain from analysis passes offered directly by LLVM (which we assume are reliable). Our taint analysis is not interprocedural, so programmers must annotate variables and parameters in each function.

Because ORIGAMI not only applies the ORIGAMI rules but also branch linearisation and loop unrolling, whenever ORIGAMI finds a loop whose bound depends on a secret, ORIGAMI rejects the program and asks the programmer to provide the annotation

`__attribute__(annotate("bound=N"))`

on the respective taint source, where N is the maximum number of times the loop could iterate given any secret.

6.9.4 Litmus Test

We consider a small program to show how ORIGAMI repairs programs with iterated array reads and writes. In the following, let $A[10][10]$ be a two-dimensional data structure of size 10×10 with randomly generated integer values; now, consider the program

```
int secret1 = N % 10, secret2 = M % 10;
for (int i=0; i<secret1; i++)
    for (int j=0; j<secret2; j++)
        A[j][i]=A[i][j];
```

where `N` and `M` are randomly generated secret integers. This program accesses A to read and write different portions of it, which is what ORIGAMI aims to repair.

This program is vulnerable with respect to both the baseline leakage model and the MAP leakage model. The loop bound of the outer loop depends on `N` and loop bound of the inner loop depends on `M`. Moreover, the first access we perform in the data structure is `A[N][M]`, which leaks both `N` and `M` according to the MAP leakage model. ORIGAMI needs to fix both the dependence of loop bounds on secrets and the MAPs for this program so that it is constant.

We remark that Racoon does not repair programs with sensitive loop trip counts like this litmus test, so we believe that their mean 16x performance overhead only applies when repairing programs which do not have sensitive loops. Thus, we cannot conclusively compare ORIGAMI to Racoon in terms of performance overhead for this litmus test; we postpone the comparison with Racoon for the following benchmarks, which require little or no loop unrolling to be repaired. We present the results of the repair in Table **??**.

Results Analysis: From the metrics of the original programs, both in `O0` and `O3`, the program displays high variance in all metrics except the number of cache misses. After

TABLE 6.1: Simulation Results for the litmus test

Metric	O0					O3				
	Average			Variance		Average			Variance	
	Original	Repair	Factor	Original	Repair	Original	Repair	Factor	Original	Repair
Size	16504	884856	53.615	0	0	16504	6082680	368.558	0	0
NumCycles	606469.2	3098976	5.11	1908459.2	0	604672.8	18269050	30.213	169281.2	0
Commited Insts	84967	299321	3.523	75442.5	0	84677	1283111	15.153	9292	0
Icache hits	114734.2	397059	3.461	197395.7	0	114214.2	1829849	16.021	13854.2	0
Icache misses	931	14535	15.612	0	0	930.6	95739	102.879	0.8	0
Icache accesses	115665.2	411594	3.558	197395.7	0	115144.8	1925588	16.723	14011.2	0
Dcache hits	114734.2	397059	3.461	197395.7	0	114214.2	1829849	16.021	13854.2	0
Dcache misses	931	14535	15.612	0	0	930.6	95739	102.879	0.8	0
Dcache accesses	115665.2	411594	3.558	197395.7	0	115144.8	1925588	16.723	14011.2	0

applying ORIGAMI, the variance is zero in all metrics, which confirms that the repair tool is homogenising accesses to A, and it is ensuring that no information leaks due to MAPs.

We remark that the number of CPU cycles (column `NumCycles`) has a variance of zero because the repaired program does not use operations whose execution time depends on parameters. In the following benchmarks, we see that ORIGAMI cannot always induce a variance of zero for this metric, since ORIGAMI does not repair the cause of this leakage (i.e., use of functions whose execution time varies with their parameters).

We also remark that the overhead factor of repairing the litmus test program (column `Factor`) is extraordinary: 5.11x for `O0` and 30.2x for `O3`. This large overhead factor is understandable; to repair this program, ORIGAMI must force both loops to have the same number of iterations independently of the secret given, so the program transforms from one with an average of 25 assignments per execution to a program which always performs 100 assignments per execution. By applying loop unrolling, ORIGAMI prevents secrets from leaking via the program counter. Then, ORIGAMI folds each of those 100 assignments over A, otherwise the secrets leak via the cache under the MAP leakage model.

We remark that the excessive overhead incurred by repairing the program is mainly due to loop unrolling and not due to the ORIGAMI rules. To confirm this claim, we apply ORIGAMI to the following benchmarks, which require little to no loop unrolling to be repaired. Without loop unrolling, we can better observe the effective impact of only applying the ORIGAMI rules to repair a program.

6.9.5 AES, OpenSSL and GDK

We now describe the other examples that we used for benchmarking, and briefly explain the results of the experiments for each of them.

Advanced Encryption Standard (AES)

AES is a specification for data encryption to establish secure communication. AES receives a plaintext message and as an array of size 16 bytes as secret inputs. These settings are enough to produce variations in the number of CPU instructions and hits or

misses in the cache **[Chalice]**. We repair the AES implementation available in **[AESBasic]**. We present the results of the repair in Table **??**.

Results analysis: since we are repairing a cryptographic encryption function, it is expected that it was developed using CTP guidelines; thus, the original version satisfies TSCF with respect to the MAP leakage model. However, the repair procedure slightly affects the program positively in terms of performance for the `O0` case, where compiler optimisations are disabled. The reason behind it is that our implementation of ORIGAMI requires the control flow graph (CFG) to match the structure of GKAT expressions, and during this transformation, the LLVM pass manager might optimise some parts of the code, and this optimisation compensates the overhead.

In the `O3` case, when compiler optimisations are enabled, we see that ORIGAMI does impact the performance of optimised code, although only by a factor of 3.2% if we consider the number of CPU committed instructions to quantify time (following the baseline leakage model). At the least, ORIGAMI does not add significant overhead to MAP TSCF compliant code.

OpenSSL Data Encryption Standard (DES)

DES is a symmetric-key algorithm for data encryption which receives as secret inputs a plaintext and an array of 8 bytes. We repair the OpenSSL implementation of DES available at **[OpenSSL]** and present the results of the repair in Table **??**.

Results analysis: unlike the AES `O0` case, the modifications introduced by ORIGAMI are not compensated by the slight optimisations used when transforming the CFG. We see a performance impact of 37% in the `O0` case and an impact of 1.7% in the `O3` case. This benchmark illustrates the benefits of having the compiler be the one in charge of enforcing TSCF, and how repair rules benefit from compiler optimisations. In particular, `O3` combines pointer computation operations (`GetElementPtr` instructions); e.g., to compute the pointer of $A[i][j]$, `O0` would first try to compute the pointer of $A[i]$ and then compute $A[i][j]$ relative to the pointer of $A[i][0]$, while `O3` would directly compute $A[i][j]$ relative to the pointer of $A[0][0]$. Due to implementation details, ORIGAMI in `O0` aggregates the partial pointer computations, heavily impacting performance. However, in the `O3` case, ORIGAMI need not aggregate partial computations, and the repair has a smaller performance impact.

OpenSSL RC4

This benchmark is a stream cipher that can be used for data encryption. Although simple, several vulnerabilities have been reported, and the use of RC4 been prohibited in standard implementations of TLS **[rfc7465]**. RC4 receives as secret inputs an array of 10 bytes and a plaintext.We repair the OpenSSL implementation of RC4 available at **[OpenSSL]** and present the results of the repair in Table **??**.

Results analysis: the original program of RC4 satisfies the OS leakage model, so it needs no repair. This case illustrates how applying ORIGAMI to safe code preserves

TSCF with respect to the original program, and it does not add significant performance penalties, since the only transformation the ORIGAMI tool is doing is some CFG restructuring so that it matches a GKAT. In this case, the penalty is 3.6% for `O0` and 5.1% for `O3`. Arguably, these type of programs do not benefit from ORIGAMI, and are better off being validated by a TSCF verifier like [**usenix_ctp_verification**] and compiled using a certified compiler like CompCert [**CompCert**].

GDK Library - `gdk_unicode_to_keyval`

We study the function `gdk_unicode_to_keyval` of the Linux GDK library. This function converts a secret ISO10646 character to a key symbol [**gdklib**]. The secret input is a single unsigned integer value. This benchmark uses a binary search that depends on a secret input, and it loads from a structure using a secret index during that binary search. The results of the repair of this benchmark appear in Table **??**.

Results analysis: the original program is not secure with respect to the baseline model, but even if we fix with respect to the baseline leakage model, we still have to deal with the MAP difference for each secret. ORIGAMI repairs both vulnerabilities. Interestingly, in the `O0` case, ORIGAMI shows that the number of CPU cycles is constant, while in the `O3` case, there are variations. However, this is just a coincidence, since ORIGAMI does not repair programs with respect to the OS leakage model.

GDK Library - `gdk_keyval_name`

Lastly, we simulate the function `gdk_keyval_name` of the Linux GDK library. This function converts a key value into a symbolic name [**gdklib**]. The secret input k is a single unsigned integer value. Again, for this test case, we generate 5 random secret k inputs. The results of repairing appear in Table **??**.

Results analysis: this benchmark behaves like `gdk_unicode_to_keyval`. The original program uses a binary search which depends on a secret, causing it to have timing side-channel vulnerabilities. ORIGAMI unfolds the binary search, and ensures that accesses to the structure holding the value-to-symbolic name pairs is loaded adequately.

6.9.6 Evaluation Conclusion

The code used for cryptographic libraries usually follows CryptoCoding guidelines [**CryptoCoding**]. This is reflected in some original binaries satisfying TSCF for at least one optimization level. This experiment lets us observe more precisely the impact of repairing secure code. We see that the impact factor remains relatively close to one for all the metrics in the benchmarks, which is reassuring: secure code should not need repair, and if it gets repaired, the impact should be minimal. We conclude that it is then viable for programmers to follow CryptoCoding guidelines, and then use ORIGAMI as a safety net without harshly impacting performance. By taking the number of CPU Cycles as a measure of time, the overhead of ORIGAMI in the benchmarks has a geometric mean of 1.24x when applied to programs with a fixed loop count, which is significantly better than the 16x overhead caused by Racoon.

TABLE 6.2: Simulation Results for AES

Metric	O0					O3				
	Average			Variance		Average			Variance	
	Original	Repair	Factor	Original	Repair	Original	Repair	Factor	Original	Repair
Size	26648	22608	0.848	0	0	22320	30744	1.377	0	0
NumCycles	660317.6	619675.2	0.938	172858.8	6919.2	617122.8	632704	1.025	9447.2	128468
Commited Insts	95836	88221	0.921	0	0	87370	90187	1.032	0	0
Icache hits	129047	118874	0.921	0	0	117596	122196	1.039	0	0
Icache misses	997	942	0.945	0	0	942	965	1.024	0	0
Icache accesses	130044	119816	0.921	0	0	118538	123161	1.039	0	0
Dcache hits	129047	118874	0.921	0	0	117596	122196	1.039	0	0
Dcache misses	997	942	0.945	0	0	942	965	1.024	0	0
Dcache accesses	130044	119816	0.921	0	0	118538	123161	1.039	0	0

TABLE 6.3: Simulation Results for OpenSSL DES

Metric	O0					O3				
	Average			Variance		Average			Variance	
	Original	Repair	Factor	Original	Repair	Original	Repair	Factor	Original	Repair
Size	37720	361264	9.578	0	0	28888	32984	1.142	0	0
NumCycles	631279.6	1085718.8	1.72	15402.8	270293.2	610879.6	620995.2	1.017	10270.8	277777.2
Commited Insts	88047	120661	1.37	0	0	85776	87256	1.017	0	0
Icache hits	118456	164702	1.39	0	0	115513	117432	1.017	0	0
Icache misses	1005	3539	3.521	0	0	952	968	1.017	0	0
Icache accesses	119461	168241	1.408	0	0	116465	118400	1.017	0	0
Dcache hits	118456	164702	1.39	0	0	115513	117432	1.017	0	0
Dcache misses	1005	3539	3.521	0	0	952	968	1.017	0	0
Dcache accesses	119461	168241	1.408	0	0	116465	118400	1.017	0	0

ORIGAMI can also repair vulnerable programs like the litmus test and the benchmarks `gdk_unicode_to_keyval` and `gdk_keyval_name`. However, repairing programs which contain loops whose trip counts depend on secrets can largely increase the size and execution time of programs. For example, the litmus test program uses secrets which range from 0 to 9, and loop iterate a different number of times depending on the value of the secrets. ORIGAMI enforces TSCF without sacrificing functionality by forcing the program to run the outer loop 10 times and the inner loop 10 times no matter which secrets are provided, obfuscating no-ops to preserve functionality with *sel* instructions. If we were to have more nested loops, this effect would compound exponentially; however, this is unavoidable, otherwise the secrets would leaks via the timing channel.

More than the applicability and usefulness of our tool, with this evaluation, we attest the viability of the philosophy of [**WhatYouCisWhatYouGet**]: compilers can be empowered to close timing-side channels automatically, and programmers need only worry about providing the right information to the compiler; independently of whether the programmer follows CryptoCoding guidelines or not.

6.10 Related Work

The existing solutions for closing timing side-channels proposed by researchers in both computer security and programming languages can be classified into two complementary main categories: *verification and testing* solutions and *enforcement* solutions. Verification and testing solutions address the problem of checking whether a system has any

TABLE 6.4: Simulation Results for OpenSSL RC4

Metric	O0					O3				
	Average			Variance		Average			Variance	
	Original	Repair	Factor	Original	Repair	Original	Repair	Factor	Original	Repair
Size	20856	20856	1	0	0	16760	20856	1.244	0	0
NumCycles	649490	660708	1.017	0	0	616062	635596	1.032	0	0
Commited Insts	96169	99648	1.036	0	0	87699	92180	1.051	0	0
Icache hits	128847	133321	1.035	0	0	118333	124891	1.055	0	0
Icache misses	960	966	1.006	0	0	942	976	1.036	0	0
Icache accesses	129807	134287	1.035	0	0	119275	125867	1.055	0	0
Dcache hits	128847	133321	1.035	0	0	118333	124891	1.055	0	0
Dcache misses	960	966	1.006	0	0	942	976	1.036	0	0
Dcache accesses	129807	134287	1.035	0	0	119275	125867	1.055	0	0

TABLE 6.5: Simulation Results for `gdk_unicode_to_keyval`

Metric	O0					O3				
	Average			Variance		Average			Variance	
	Original	Repair	Factor	Original	Repair	Original	Repair	Factor	Original	Repair
Size	22520	47096	2.091	0	0	20520	192552	9.384	0	0
NumCycles	594394.4	672250	1.131	24780.8	0	594569.6	1044676.4	1.757	229836.8	126774.8
Commited Insts	83921	91832	1.094	125	0	83820.4	119655	1.428	3678.8	0
Icache hits	113138	122786	1.085	245	0	112984.4	162846	1.441	5671.8	0
Icache misses	905	1318	1.456	0	0	901	3612	4.009	0	0
Icache accesses	114043	124104	1.088	245	0	113885.4	166458	1.462	5671.8	0
Dcache hits	113138	122786	1.085	245	0	112984.4	162846	1.441	5671.8	0
Dcache misses	905	1318	1.456	0	0	901	3612	4.009	0	0
Dcache accesses	114043	124104	1.088	245	0	113885.4	166458	1.462	5671.8	0

TSC vulnerabilities. Enforcement solutions ensure that the resulting systems have no TSC vulnerabilities.

Verification Solutions Verification solutions help us determine whether a system has timing side-channel vulnerabilities, but they do not offer automated repair solutions. There are verification solutions for source (e.g., ABPV13 [**ABPV13**],Low*[**LowStar**] and [**7958606**]), intermediate (e.g., [**usenix_ctp_verification**; **FlowTracker**]), assembly (e.g., [**Jasmin**; **Vale**]) and binary (e.g., [**CacheAudit**; **KMO12**]) levels. We refer the interested reader to Barbosa *et al.* [**timing-channel-survey**], which contains an overview of tools for side-channel resistance.

The verification procedure presented in [**SecureCompilation**] does not determine whether programs satisfy TSCF or not; instead, they verify whether compilers preserve the cryptographic constant-time properties of programs during compilation. This is a far better guarantee that just trusting the compilers to do so. They do not address the problem of program repair.

ORIGAMI is a static enforcement repair solution, so it does not fit the verification category. However, we believe that ORIGAMI could be used in conjunction with verification solutions to check which programs effectively need repair before applying ORIGAMI unnecessarily.

Testing Solutions Testing solutions like ct-fuzz [**ct-fuzz**] and [**stvr.1718**] aim to provide counterexamples that violate TSCF. By a counterexample, we mean two sensitive

TABLE 6.6: Simulation Results for `gdk_keyval_name`

Metric	O0					O3				
	Average			Variance		Average			Variance	
	Original	Repair	Factor	Original	Repair	Original	Repair	Factor	Original	Repair
Size	49456	53552	1.083	0	0	49080	368568	7.51	0	0
NumCycles	595192.48	606580	1.019	16235.09	0	593816.56	1429166.72	2.407	1635.51	1023769.29
Commited Insts	83977.32	85130	1.014	51.14	0	83848.12	149118	1.778	11.86	0
Icache hits	113206.12	114574	1.012	94.61	0	113021.44	205123	1.815	17.17	0
Icache misses	901	970	1.077	0	0	899	5913	6.577	0	0
Icache accesses	114107.12	115544	1.013	94.61	0	113920.44	211036	1.852	17.17	0
Dcache hits	113206.12	114574	1.012	94.61	0	113021.44	205123	1.815	17.17	0
Dcache misses	901	970	1.077	0	0	899	5913	6.577	0	0
Dcache accesses	114107.12	115544	1.013	94.61	0	113920.44	211036	1.852	17.17	0

inputs that could cause a program to display different execution times. However, these testing solutions do not repair programs in case that they find a counterexample. Similarly to verification solutions, we believe that ORIGAMI can be used in conjunction with these testing solutions.

Enforcement solutions Enforcement solutions are quite diverse, and can be further subdivided into two categories: *runtime enforcement* and *static enforcement*.

Runtime enforcement solutions, like SCHMIT [**Schmit**], alter the *execution* of programs to dynamically reduce the leakage of existing side channels. Static enforcement solutions modify the programs before execution, and do not monitor their execution.

Static enforcement solutions can be further subdivided into two categories: *synthesis* and *repair*. *Synthesis* solutions apply a security-by-design principle, so systems generated by this tools satisfy TSCF. Among synthesis tools we find FaCT [**FaCT**] and CompCert [**CompCert**]. FaCT is a C-like DSL that always generates constant-time LLVM bitcode, and CompCert is a certified compiler that ensures that properties satisfied by the original program are preserved in the compiled program. We appreciate the security-by-design aspect offered by FaCT: if only secure programs can be written, then TSCF is automatically enforced. However, this guarantee only holds if optimization passes are disabled, since they might introduce TSC vulnerabilities. Moreover, FaCT simply reject programs that access data structures with secrets. Unlike FaCT, we let the compiler apply any optimizations it deems necessary, and then we use the ORIGAMI rules to ensure the resulting code satisfies TSCF under the MAP leakage model.

Program repair solutions like Racoon[**Racoon**], SC-ELIMINATOR [**SCEliminator**] and [**MSESC**] often modify programs at the IR level (i.e., they modify the front-end of the compiler). All static enforcement solutions assume that actors further ahead in the compilation and execution processes do not undo their modifications to the program. Other repair solutions, like oo7 [**oo7**] work at lower levels, but they protect programs against Spectre V1 attacks, and not against leakage caused directly by the program.

Our tool is mostly similar to Racoon, SC-ELIMINATOR and the solution proposed in [**MSESC**] because we do static enforcement; however, ORIGAMI offers an advantage over each of these three solutions. The tool in [**MSESC**] does not repair programs with respect to the MAP leakage model (they repair with respect to the baseline model), the rules proposed by SC-ELIMINATOR are unsound in the context of the MAP leakage

model, and while Racoon does offer some protection against the MAP leakage model, the performance overhead has a geometric mean of 16x, while ORIGAMI has a performance overhead of 1.24x when repairing programs that do not require loop unrolling.

Shared limitations similarly to Racoon and SC-ELIMINATOR, ORIGAMI does not repair recursive calls, does not offer security guarantees over side-effects (e.g. I/O system calls), and does not repair programs whose timing side-channel vulnerability stems from the usage of functions whose execution time depends on their parameters (i.e., programs vulnerable in the OS leakage model).

Spectre V1 Spectre V1 attacks **[Spectre]** rely on the speculative execution of a gadget of the form $x := B[A[s]]$ to leak secrets into the cache. Under the MAP leakage model, $x := B[A[s]]$ reveals $A[s]$ when the program **if** $s < size_A$ **then** $x := B[A[s]]$ executes, so if some secret value k is speculatively accessed via the $A[s']$, and then loaded in memory by $B[A[s']]$, the attacker can reverse engineer the value of $A[s']$, which is k. While Spectre attacks load secrets in the cache through speculative execution, we focus on repairing programs that load secrets directly via memory access patterns.

ORIGAMI does not defend against Spectre type attacks. Our security guarantees only apply for non-speculative behaviours of the program. To repair programs that exhibit speculative leaks, we recommend to use a methodology that finds speculative leaks (e.g. **[pitchfork]**) or a methodology that prevents speculative leaks by cutting dataflows from sensitive sources to speculative sinks (e.g. BLADE **[Blade]**).

6.11 Discussion

Although ORIGAMI offers a sound method to enforce TSCF with respect to the MAP leakage model, there is still a long way to go until true TSCF is achieved. Most notably, ORIGAMI does not repair programs that are vulnerable with respect to the operand sensitive leakage model, but adds another layer of security by repairing programs that are safe with respect to the baseline leakage model, yet unsafe with respect to the MAP model.

While we share the philosophy of **[WhatYouCisWhatYouGet]**, and we expect compilers to take over the task of enforcement of TSCF instead of the programmer, we believe that as long as hardware does not offer strong TSCF guarantees which can be depended on and rightfully used by compilers, any software based solution for TSCF enforcement, including ORIGAMI, will remain conditional, and thus not entirely reliable. Nevertheless, while this type of hardware is available, software-based solutions can mitigate timing side-channel vulnerabilities.

Chapter 7

Conclusion

Thorough this thesis, we used the Research Questions **??**, **??**, **??** and **??** to motivate our research, which provides an intuition of the usefulness of LBA. We studied these questions in three practical scenarios: the classification of attacker models in Chapter **??**, the quantification of robustness in CPS in Chapter **??**, and timing side-channel repair in Chapter **??**. Although these three applications are quite different in terms of the domain of their problems, we successfully approached them using LBA, attesting the applicability and generality of the method.

With respect to Research Question **??** – the modelling of attacks and attacker models– we conclude that LBA is expressive enough to model practical attackers, including man-in-the-middle attackers and attackers that can override the values of state variables while leaving the program of the transition system intact. Nevertheless, due to its focus on spatial transformations, LBA has limitations in its applicability. For example, we cannot capture the attackers of a transition system that can directly change the transition relation. To model this type of attackers, we would need to consider *dynamics transformations*, the dual of spatial transformations, which we briefly discussed in Section **??**. LBA also cannot capture attackers that can add or remove states from the carrier set. However, there may be sets of spatial transformations in the resulting carrier set that emulate these attackers.

To address Research Questions **??** and **??** –the quantification of harmful effects of attackers and how to compare them– we conclude that LBA can provide an adequate measure for these harmful effects, which we use to compare attackers. To compute this measure, we need a list of security requirements to determine when an effect is considered harmful, and a set of capabilities for each attacker model. LBA describes a framework to systematically generate systems that we can check using either verification or testing techniques. We used a verification approach in Chapter **??** and a testing approach in Chapter **??**.

In a verification approach, we can use a SAT solver to find a spatial transformation that is capable of breaking at least one requirement. This method has the advantage that we need not need verify each spatial transformation generated by the attacker model, and a negative answer confirms the inability of the attacker to affect the system in a harmful way. However, this method is at a disadvantage if we are studying several attackers, since requires a heavy setup: to take advantage of the optimisations available in SAT solvers, we should avoid changing the model for each instance of model checking. This

requirement implies that we should create a single model that contains all attackers, where we adjust the set of assumptions accordingly to define the attacker model we want to study.

In a testing approach, we systematically generate spatial transformations corresponding to the attacker model, and we monitor the execution of the system after it has been affected by the spatial transformation. If a requirement is broken during the testing of a spatial transformation, then we conclude that the attacker can harm the system. The advantage of testing is that we obtain results relatively fast if either the attacker model generates a small amount of spatial transformations, or the system is particularly vulnerable to the attacker. The main disadvantage of a testing approach is that if the attacker model generates a large number of spatial transformations, it may be impractical to test them all, which can lead to inconclusive results.

Given these observations, we believe that the verification and testing approaches are complementary. We can do testing for the smaller attacker models, and verification for the larger attacker (or counterattacker) models. Smaller attacker models generate a treatable number of spatial transformations, which can be systematically tested. Moreover, thanks to monotonicity properties (see Theorem **??** and Corollaries **??** and **??**), finding successful attackers that have minimal capabilities should be a priority.

Finally, with respect to the Research Question **??** –how to repair a system vulnerable to a given attacker model– we propose two approaches: an approach based on finding counterattacks, shown in Chapter **??**, and an enforcement mechanism via spatial transformations, shown in Chapter **??**. In Chapter **??**, we quantify the robustness of a system with respect to the attacker model to obtain a relative measure of security. In addition, we consider a counterattacker model, and we use LBA to identify the attacks that have at least one counterattack. Based on the similarities between the attacks countered by a single counterattack, we can propose a repair (i.e. a change to the program of the system). To check if the repair is sound, we quantify the robustness of the new version to verify that it is greater than the robustness of the original system. In Chapter **??**, we use a spatial transformation to enforce a security property (i.e. timing side-channel freedom with respect to memory access patterns). This illustrates the dual purpose of spatial transformations: attackers can use them to break security requirements, and designers can use them to enforce security properties.

To conclude this thesis, we briefly describe four directions for future work that, in our opinion, expand the theory and applicability of LBA.

7.1 Beyond Deterministic Transition Systems

LBA is a framework rooted in universal coalgebra. Thus, LBA directly benefits from the major results in the theory of coalgebras. Most notably, LBA is extensible to systems tht incorporate monadic computation by means of the generalised determinisation procedure presented by Silva *et al.* in [**GeneralisingDetermination**]. Our intuition tells us that, given a monad T with unit η and multiplication μ that models a computational side-effect (e.g., non-determinism), and a functor F, the procedure in [**GeneralisingDetermination**] lifts an FT-coalgebra (X, c) where $c\colon X \to F(T(X))$

to a semantically equivalent F-coalgebra $(T(X), c^\sharp)$ with $c^\sharp \colon T(X) \to F(T(X))$. We believe that a latent FT-coalgebra $(X, c \circ m)$ with $m \colon X \to X$ lifts to a semantically equivalent latent F-coalgebra $(T(X), c^\sharp \circ T(m))$, since $T(m) \colon T(X) \to T(X)$. The functor T aggregates the effects of m in X to obtain $T(m)$; e.g., in the case of non-determinism, if $X = \{a, b\}$ and $m \colon X \to X$ maps a to b, and b to itself, the transformation $\mathscr{P}(m) \colon \mathscr{P}(X) \to \mathscr{P}(X)$ maps $\emptyset$ to itself, $\{a\}$ to $\{b\}$, $\{b\}$ to itself, and $\{a, b\}$ to $\{b\}$. We leave a definitive proof semantic equivalence between the latent FT-coalgebra $(X, c \circ m)$ and the latent F-coalgebra $(T(X), c^\sharp \circ T(m))$ as future work.

7.2 Predictability

By only using spatial transformations $m \colon X \to X$ and not dynamics transformations $b \colon F(X) \to F(X)$, we force every latent coalgebra $(X, c \circ m)$ to somehow factor through their original coalgebra (X, c). While these latent coalgebras are clearly and undoubtedly related to their original coalgebra, the relations between the original behaviours and the revealed latent behaviours are, in general, hard to predict from the spatial transformations. We attribute this unpredictability to the lack of restrictions over the choice of spatial transformations.

In geometry, we can predict the effect of some transformations on objects before we apply them, e.g., symmetries. We believe that there should be a non-trivial family of spatial transformations whose impact on the behaviour of systems is predictable before they are used (the trivial families of spatial transformations are constants and identity transformations). Our intuition points towards spatial transformations that naturally lift to dynamics transformation; more precisely, the transformations where the latent coalgebra $(X, F(m) \circ c)$ and $(X, c \circ m)$ have the same behaviours are predictable. The systems that satisfy this property are known as *bialgebras* [**JacobsBook**]. We leave formal proof of the predictability of the effects of spatial transformation in these systems as future work.

7.3 A Proof Principle

In this thesis, we studied the effects that a single spatial transformation $m \colon X \to X$ has on an F-coalgebra (X, c); however, since m is an element of the monoid of endofunctions in X, it can surely be non-uniquely decomposed into a finite sequence of transformations $m_1, m_2, \ldots m_n$ where $m = m_n \circ \ldots \circ m_2 \circ m_1$ each with $m_i \colon X \to X$, where all m_i preserve a behavioural property Q. If that is the case, then $(X, c \circ m)$ should satisfy the property Q. This proof principle follows the line of certified compilers (e.g., CompCert [**CompCert**]) that can transform one system into another while preserving their behavioural properties.

This proof principle seems particularly useful for a system (X, d) that can be described as a latent coalgebra of a system (X, c) and a spatial transformation m (i.e., $d = c \circ m$), where (X, c) is a system where proving the property Q is relatively simple, but proving it in (X, d) is daunting. For example, consider the system in Figure **??**, which is a latent system of the one in Figure **??**; perhaps it is easier to prove a property of the system in Figure **??** by showing that it holds in the system in Figure **??** and showing that

the spatial transformation in Figure **??** preserves it, instead of proving such property directly.

7.4 Automated Repair

Informally, the system repair problem is the following: given a list of requirements R and a system (X, c), is there a way to make (X, c) satisfy all the requirements in R? We can approach a simpler yet related problem with LBA as follows: is there an efficient procedure to determine whether there exists a spatial transformation $m\colon X \to X$ such that $(X, c \circ m)$ satisfies all the requirements in R? If we find such a transformation m and we reveal the latent system $(X, c \circ m)$, we have found a solution for the repair problem.

Defining the search space for the spatial transformation is not a trivial problem, as we have discussed in Chapters **??** and **??**; moreover, we have not studied important testing or verification considerations if the system (X, c) has computational side-effects (e.g., non-determinism). We believe that this scenario offers yet another practical and interesting use case for LBA, which we would like to study in the future.