Monitoring and Enforcement of Safety Hyperproperties

by

Shreya Agrawal

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

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

Abstract

Certain important security policies such as *information flow* characterize system-wide behaviors and are not properties of individual executions. It is known that such security policies cannot be expressed in trace-based specification languages such as linear-time temporal logic (LTL). However, formalisms such as *hyperproperties* and the associated logic HyperLTL allow us to specify such policies. In this thesis, we concentrate on the static enforcement and runtime verification of safety hyperproperties expressed in HyperLTL.

For static enforcement of safety hyperproperties, we incorporate *program repair* techniques, where an input program is modified to satisfy certain properties while preserving its existing specifications. Assuming finite state space for the input program, we show that the complexity of program repair for safety hyperproperties is in general NP-hard. However, there are certain cases in which the problem can be solved in polynomial time. We identify such cases and give polynomial-time algorithms for them.

In the context of runtime verification, we make two contributions: we (1) analyze the complexity of decision procedures for verifying safety hyperproperties, (2) provide a syntactic fragment in HyperLTL to express certain k-safety hyperproperties, and (3) develop a general runtime verification technique for HyperLTL k-safety formulas, for cases where verification at run time can be done in polynomial time. Our technique is based on runtime formula progression as well as on-the-fly monitor synthesis across multiple executions.

Acknowledgements

I would like to thank my supervisor, Prof. Borzoo Bonakdarpour, for his guidance and support throughout my Master's. His insights and discussions about the problems were of immense help. I would like to especially thank my readers Prof. Ian Goldberg and Prof. Eric Blais for their invaluable insights and comments.

My life at Waterloo has been an amazing journey because of the people I met here and the friends I made. Everyone has been extremely caring and supportive. My love and regards to them all for always bringing a smile to my face. I would like to thank my climbing friends Kevin and Georg who got me introduced to climbing and made my Wednesday evenings great at Trivia Nights. My office mates Hella, Stephen and Marianna made me look forward to come to work every morning. They introduced me to their amazing friends Valerie, Dean, Oliver, Nika and Jack who were always so kind to me. Sharon, you have been a very caring roommate, thank you so much. Cecylia always manages to bring so much energy in me with her fun-loving nature.

I can never thank my dearest friends David, Hemant, Neeraj and Vijay enough. Always making me laugh, being there for me in my good and bad times, listening to my craziness and even being a part of it. Thank you for making my dream trip happen, it would not have been so fantastic without you guys there. I would like to thank my friends from DA-IICT who are a constant support. Nikhil, it would take up another thesis altogether if I start to write down everything I would like to thank you for. Aakash, Anurag, Jigar, Lalit, Naman, Namrata, Neel, Sagar, Sid, Sumeet, Tanu—you know I will love you guys always.

In all my endeavours and accomplishments, the love and support I received from my parents and brother can never be thanked enough. I am most grateful to them for being there for me always. Abhinav (Bhaiya) you are the sweetest brother anyone can ask for. You are truly the best man in my life.

Dedication

This is dedicated to my mother without whom this would have not been possible. I would have never been the person I am today without her love and brilliance. All my hard work, achievements and endeavors are only because of her constant motivation and encouragement. She is my biggest inspiration and I am always aspiring to be as loving, caring and amazing as she is.

Table of Contents

List of Tables ix

List of Figures x

Nomenclature xi

1 Introduction 1

1.1 Difficulties in Formal Treatment of Security Policies . . . 2

1.2 Thesis Statement . . . 4

1.3 Runtime Verification of Hyperproperties . . . 5

1.3.1 Challenges . . . 5

1.3.2 Contributions . . . 5

1.4 Automated Program Repair for Hyperproperties . . . 6

1.4.1 Challenges . . . 7

1.4.2 Contributions . . . 7

1.5 Organization . . . 9

2 Preliminaries 10

2.1 Programs . . . 10

2.2 Trace Properties . . . 12

2.3 Hyperproperties . . . 13

2.3.1 Safety Hyperproperties . . . 14
2.3.2 Co-safety Hyperproperties . . . 16
2.3.3 Liveness Hyperproperty . . . 16
2.4 HyperLTL . . . 17
2.4.1 Syntax . . . 17
2.4.2 Semantics . . . 17
2.4.3 Specifying Trace Relations . . . 18

3 Runtime Verification of k-Safety Hyperproperties in *HyperLTL* 20
3.1 k-Safety/Co-k-Safety Hyperproperties in HyperLTL . . . 20
3.1.1 Relation between Hypersafety and Co-hypersafety Properties . . . 21
3.1.2 Representing k-safety and Co-k-safety Hyperproperties in HyperLTL 22
3.2 Monitorability in HyperLTL . . . 24
3.3 Complexity of Verification of Safety Hyperproperties at Run Time . . . 28
3.3.1 Undecidability . . . 29
3.3.2 NP-hardness for a Subclass of k-safety Hyperproperties . . . 30
3.3.3 A Sufficient Condition for Polynomial-Time Runtime Verification . 31
3.4 Monitoring Algorithm . . . 32
3.4.1 Progression for Trace Relations . . . 34
3.4.2 Algorithm . . . 35
3.4.3 Monitoring beyond k-hypersafety . . . 39
3.5 Implementation and Results . . . 42
3.5.1 Experimental Settings . . . 42
3.5.2 Results and Analysis . . . 43

4 The Complexity of Program Repair for Safety Hyperproperties 47
4.1 Problem Statement . . . 48
4.2 Repair for k-Safety . . . 49

4.3 Repair for 1-Safety . . . 52
4.4 Polynomial-time Repair for Safety Hyperproperties . . . 57
4.4.1 Polynomial-time Repair for a Class of k_1-Safety Hyperproperty . . . 58
4.4.2 Polynomial-time Repair for a Class of 1_ℓ Safety Hyperproperty . . . 60

5 Related Work **63**
5.1 Runtime Verification . . . 63
5.1.1 Runtime Verification for Linear Temporal Logics . . . 63
5.1.2 Verification of Security Policies . . . 64
5.2 Program Repair . . . 65
5.2.1 Repair for Temporal Logics . . . 65
5.2.2 Repair for Various Systems . . . 66
5.2.3 Checking of Safety Hyperproperties . . . 66
5.2.4 Synthesis and Repair for Security Policies . . . 67
5.3 Runtime Enforcement of Security Policies . . . 68

6 Conclusion **69**
6.1 Summary . . . 69
6.1.1 Runtime Verification . . . 69
6.1.2 Program Repair . . . 70
6.2 Future Work . . . 70
6.2.1 Runtime Verification . . . 71
6.2.2 Program Repair . . . 71

References **72**

List of Tables

3.1 Monitorability of universally quantified HyperLTL$_1$ formulas. 26

3.2 Monitorability of existentially quantified HyperLTL$_1$ formulas. 27

3.3 Monitorability of HyperLTL formulas with conjunction or disjunction of formulas from Tables 3.1 and 3.2 28

3.4 Trace length of monitored formulas before first violation 45

List of Figures

1.1 EDAS conference management website's security violation 3

2.1 Secret sharing program p 11

3.1 Monitor of secret sharing policy. 33
3.2 LTL_3 monitor for formula $a \,\mathbf{U}\, b$. 35
3.3 Petri net for property termination-sensitive 38
3.4 Number of Petri net components generated before detection of first violation 44
3.5 Total components vs. violations 45
3.6 Length of formulas vs. trace length 46

4.1 Instance 2SS 51
4.2 Instance 1S2PP 53
4.3 The partial structure of the revised program for instance 1S2PP. 55
4.4 Secret sharing repaired program 62

Nomenclature

$\top$	evaluates to true
$\bot$	evaluates to false
?	evaluates to unknown
$\wedge$	logical and
$\vee$	logical or
$\forall$	'for all paths'
$\exists$	'there exists a path'
X	Next operator in Ltl
U	Until operator in Ltl
F	Finally operator in Ltl
G	Globally operator in Ltl
Ltl	Linear-time Temporal Logic
LTL	set of all Linear-time Temporal Logic formulas
Ctl	set of all Computation Tree Logic formulas
LTL_S	set of all LTL safety formulas
LTL_C	set of all LTL co-safety formulas
HyperLTL$_1$	set of HyperLTL formulas with no quantifier alternation

HyperLTL-3	3-valued semantics for HyperLTL
AP	set of atomic propositions
Σ	set of all letters
Σ^*	set of all finite traces
Σ^ω	set of all infinite traces
$\Sigma^{\leq \ell}$	set of all traces of length at most ℓ
$t[i]$	element of t at index i
$t[0, i]$	trace prefix upto and including ith element
$t[i, \infty]$	infinite suffix beginning with ith element
$\mathcal{P}$	powerset operator
$\mathcal{P}(\Sigma^\omega)$	set of all trace properties
$\mathcal{P}^*(X)$	set of all finite-size subsets of X
$\mathcal{P}(\mathcal{P}(\Sigma^\omega))$	set of all hyperproperties
$\models$	trace property (and hyperproperty) satisfaction
p	program
I_p	set of initial states
δ_p	set of transitions
$\leq$	trace (or trace set) prefix
ψ	set of all traces of a program
$\mathbb{E}$	existing specifications of a program
$\boldsymbol{H}$	A hyperproperty
$\boldsymbol{S}$	A safety hyperproperty
$\boldsymbol{OD}$	Observational Determinism

$\boldsymbol{GMNI}$	Goguen and Meseguer's non-interference
$\boldsymbol{SS_k}$	k-safety hyperproperty for secret sharing scheme for k shares
$\boldsymbol{SecS}$	safety hyperproperty for secret sharing scheme
$\boldsymbol{M_h}$	a set of finite sets of finite traces
$\approx_L$	low-indistinguishibility relation on traces
$=_L$	low-indistinguishibility relation on states
$=_{L,in}$	low-indistinguishibility relation on states for input variables only
$=_{L,out}$	low-indistinguishibility relation on states for output variables only
$=_H$	high-indistinguishibility relation on states
Γ	set of all trace variables
Π	mapping of trace variables to traces
π	trace variable
$\sim_{f,P}$	trace relation over function f and atomic propositions P
$\boldsymbol{Pg}$	progression function
$\uparrow m$	set of all infinite continuations of trace m
$\mathcal{M}$	a monitor for runtime verification
Q	set of states of the DFA
q_0	initial state of the DFA
Δ	set of transitions for the DFA or the Petri net
λ	function to evaluate a state of a DFA to a value in $\{\top, \bot, ?\}$
$\mathbb{B}_3$	$\{\top, \bot, ?\}$
$S = (L, \Sigma, \Delta)$	a petri net
L	set of places of the petri net

${}^\bullet\tau$	input place of transition τ
$\tau^\bullet$	output place of transition τ
$q_\perp$	state that evaluates to $\perp$
$\sup S$	supremum of set S
$\inf S$	infimum of set S

Chapter 1

Introduction

Due to the interdependence of scripts and data, security threats to the authorized access of data, and to its confidentiality and integrity, are becoming a major concern. For example in web browsers, web pages holding secure information may include untrusted third-party javascript code in the form of libraries or advertisements. This potential security loophole can be exploited to either compromise security/policies in the form of having access to unauthorized information or planting malware. With the wide adoption of Internet of Things, cyber attacks are likely to result in physical threats in addition to the virtual ones.

Given that even a short transient violation of security policies may result in leaking private or highly sensitive information, software security assurance is one of the crucial requirements in today's computer systems. In particular, according to the U.S. National Strategy to Secure Cyberspace:

> *"A...critical area of national exposure is due to the many flaws that exist in critical infrastructures due to software vulnerabilities. New vulnerabilities emerge daily as use of software reveals flaws that malicious actors can exploit. Currently, approximately 3,500 vulnerabilities are reported annually." [50]*

As a result, it is critical to ensure enforcement of the required policies and timely detection of security loopholes. While enforcement could be done either to fix existing flaws or to additionally enforce new properties, both without requiring the system to be rebuilt from the bottom up, detection involves checking for the satisfaction of the desired properties.

One way to deal with this problem is to utilize formal methods. We model a system along with its behaviors, and the environment and the intention is to mathematically analyze the system correctness in an automated fashion. In the context of system security,

numerous formal methods have been developed, most notably, different inference frameworks [24], model checking [11, 39, 43], and theorem proving techniques [49].

1.1 Difficulties in Formal Treatment of Security Policies

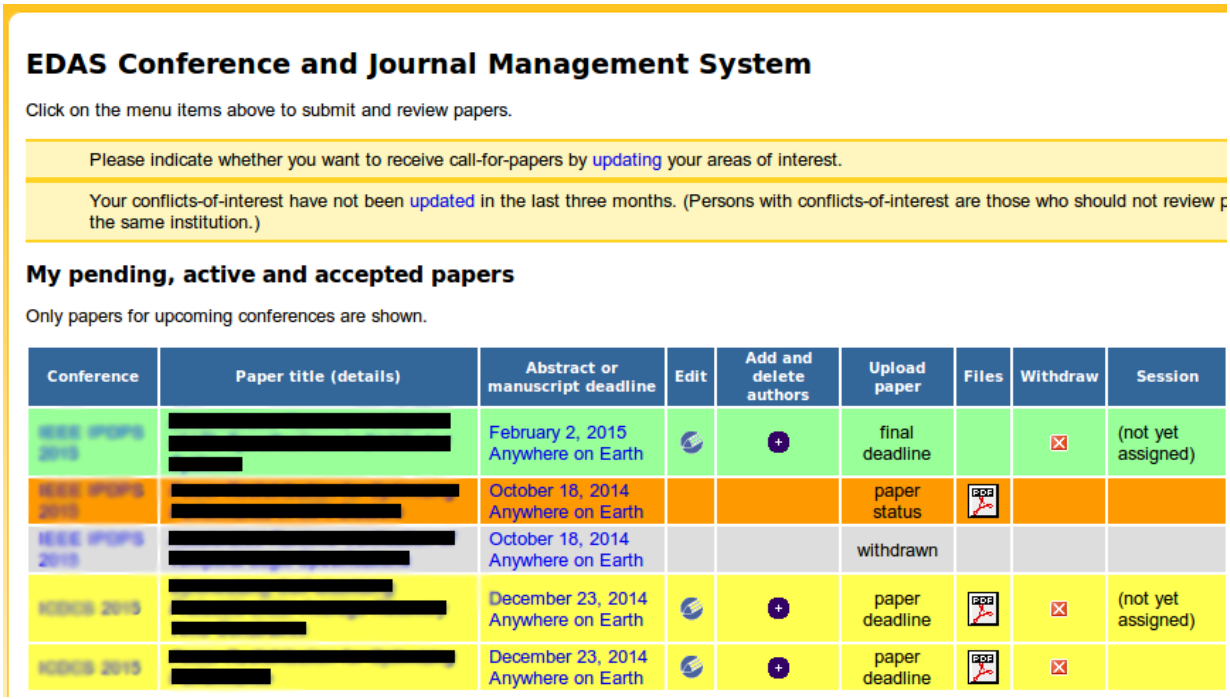
There are three major difficulties in designing automated formal methods to reason about the correctness of secure systems: (1) their scalability (2) human-in-the-loop when bugs are identified, and (3) expressiveness of specification languages. The first problem is due to the fact that the entire state space of the system may need to be explored. The second issue arises when a bug identified by a verification technique needs to be fixed. This process is now conducted manually. The third problem stems from the fact that many interesting security policies cannot be expressed by trace-based specification languages such as the propositional linear-time temporal logic (LTL) [76]. Examples of policies that cannot be expressed using trace-based specification languages include *information flow* requirements such as:

- Goguen and Meseguer's *non-interference* (***GMNI***) where the removal of private information in a system should not affect the information observed by public observers [45].
- *Observational determinism*, where any event observed by public observers must happen deterministically based on only the inputs of the public observers [71].
- *Information leakage*, where over every series of experiments in a system, the quantity of information leak is less than x bits [33], or users of channel a cannot send messages to the users of channel b [30].[1]

To demonstrate the subtlety of reasoning about these policies, consider the screenshot from the author's supervisor's EDAS Conference Management[2] web interface in Fig. 1.1, taken in early 2015. The table shows the status of submitted papers (accepted, rejected, withdrawn, and pending). This web interface exhibits the following blunt violation of non-interference. The first two rows show the status of two papers to a conference after its

[1] A channel can be, for example, some communication line in a physical or virtual network, a set of memory addresses, etc.

[2] http://www.edas.info



EDAS Conference and Journal Management System

Click on the menu items above to submit and review papers.

Please indicate whether you want to receive call-for-papers by updating your areas of interest.

Your conflicts-of-interest have not been updated in the last three months. (Persons with conflicts-of-interest are those who should not review p the same institution.)

My pending, active and accepted papers

Only papers for upcoming conferences are shown.

Conference	Paper title (details)	Abstract or manuscript deadline	Edit	Add and delete authors	Upload paper	Files	Withdraw	Session
IEEE IPDPS 2015	[redacted]	February 2, 2015 Anywhere on Earth			final deadline			(not yet assigned)
IEEE IPDPS 2015	[redacted]	October 18, 2014 Anywhere on Earth			paper status			
IEEE IPDPS 2015	[redacted]	October 18, 2014 Anywhere on Earth			withdrawn			
ICDCS 2015	[redacted]	December 23, 2014 Anywhere on Earth			paper deadline			(not yet assigned)
ICDCS 2015	[redacted]	December 23, 2014 Anywhere on Earth			paper deadline			

Figure 1.1: In the figure, the first paper is accepted while the second is rejected. The third and fourth papers are reviewed but the decision is not yet public. However, since the Session attributes of first and fourth are the same, one can infer that the fourth paper is also accepted.

notification: The first paper is accepted while the second is rejected. The last two rows show two papers submitted to a different conference with 'pending' status at the time the screenshot was taken. Although the authors should not be able to infer the internal decision making activities, this table does exactly that. By comparing the first and the fourth row, one can observe that their 'Session' column have the same value (i.e., 'not yet assigned'). Similarly, both the second and the last rows have an empty 'Session' column. This simply means that the paper on the fourth row is internally marked as accepted while the last paper is internally marked as rejected. This is clearly a security violation when two independent executions result in leaking sensitive information.

To formalize such security policies, Clarkson and Schneider introduced the notion of *hyperproperties* [34]. A hyperproperty is a set of sets of execution traces (i.e., a set of properties). Reasoning about systems using hyperproperties is especially challenging for

the following reasons. For trace-based languages, satisfaction of a property by a program is defined based on the principle of language inclusion, i.e., whether the set of traces (language) of the program is a subset of the set of traces (language) allowed by the property. On the contrary, in the case of hyperproperties, verification is equivalent to determining language equality, i.e., a program satisfies a hyperproperty if the set of traces of the program is identical to one of the set of traces in the hyperproperty.

Similar to the traditional concepts of safety and liveness for regular properties [2], hyperproperties are also classified as *safety hyperproperties* and *liveness hyperproperties* [34]. Violation of a safety hyperproperty is finitely observable and irremediable. Thus, if a set of traces T violates a given safety hyperproperty, then this can be observed by looking for undesirable patterns in T. More specifically, a set $M \subseteq T$ of finite traces (known as *"bad traces"*) will depict an undesirable behavior. Moreover, this is irremediable in that any extension of M also violates that safety hyperproperty. If the size of every such finite set M is at most k, it is called a k-safety hyperproperty. For example, Goguen and Meseguer's *non-interference* policy (***GMNI***) is a 2-safety hyperproperty as information could be leaked by observing 2 bad traces. Also note that the security violation in EDAS (Figure 1.1) is, in fact, a violation of non-interference.

Next, we describe our contributions to address the aforementioned problems in the context of hyperproperties. Specifically, we make contributions in static enforcement through program repair and verification through monitoring at run time. It is mostly believed that monitoring of security policies that cannot be expressed in a trace-based specification language cannot be done without using a static analyzer. In the context of program repair, it is currently unknown whether repair for such policies is computationally more complex.

1.2 Thesis Statement

In this thesis, we will validate the following statements:

- Monitoring certain security policies that cannot be expressed in a trace-based specification language at run time is possible without the assistance of a static analyzer.
- Automated program repair of safety security policies that cannot be expressed in a trace-based specification language has the same computational complexity as for safety security policies that can be expressed in such languages.

1.3 Runtime Verification of Hyperproperties

While exhaustive verification methods are extremely valuable, they often require developing an abstract model of the system and may suffer from the infamous state-explosion problem [31] (i.e., the first problem mentioned in Section 1.1). *Runtime verification* (RV) refers to a technique where a monitor checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. RV complements exhaustive verification techniques as well as under-approximating techniques such as testing and tracing. In the context of cybersecurity, RV is expected to be even more effective as new threats that were not considered during design time may exploit existing vulnerabilities, can now be easily detected at run time.

1.3.1 Challenges

RV of hyperproperties such as *information flow* policies is especially challenging because the monitor has to reason about the policy across different executions. For example, in Fig. 1.1 (and in fact in any HTML generation of this sort) the rows of the table are generated by independent executions of a function. Now, the monitor has to do two things. Firstly, the monitor has to deal with the fact that it only observes a finite execution at run time. Secondly, it needs to implement a mechanism to memorize and reason about its observations across multiple executions that occurred in the past and will happen in the future. Some existing methods such as the ones by Chudnov et al. [28] and Magazinius et al. [59] (for monitoring information flow) attempt to solve this problem using a hybrid of an online monitor with static analysis to annotate the conditional branches that were not taken in the current run. Chudnov et al. use hybrid monitoring techniques for specifications in relational logic [27]. Le Guernic et al. [46] provide a judgement on observing a single execution alone through dynamic analysis that uses the results of a previously run static analysis. This technique enforces security requirements by either concluding that the execution is harmless or by altering the violating execution.

1.3.2 Contributions

In this thesis, we introduce an RV technique without the assistance of a static analyzer for monitoring *safety hyperproperties*—which represent a rich class of security policies. The RV monitor produces verdicts on whether or not the currently observed executions is equal to one of the trace sets of the hyperproperty. In particular, we make the following contributions:

- First, we present a mapping from a subset of k-safety hyperproperties to HyperLTL—a temporal logic that allows quantification over execution traces [32]. We show that a subset of k-safety (respectively, co-k-safety) hyperproperties can be syntactically expressed as a disjunctive (respectively, conjunctive) HyperLTL formula with at most k universal quantifiers.

- We generalize the 3-valued semantics of Ltl (LTL_3) [14] to k-safety HyperLTL formulas. Through this generalization, we define the notion of *monitorability* and identify k-safety and co-k-safety hyperproperties that are monitorable based on their syntactic representation. We also identify other classes of HyperLTL formulas that are monitorable.

- We propose a monitoring algorithm for k-safety and co-k-safety HyperLTL formulas. Our algorithm employs three techniques: (1) a progression logic (which enables us to reason about trace inter-dependencies), (2) on-the-fly LTL_3 monitor generation, and (3) a procedure that aggregates the progressed formulas and computes run time verdicts using the generated LTL_3 monitors. We emphasize that the algorithm only evaluates formulas and it does not react to violations detected at run time.

- We show that runtime verification of certain k-safety hyperproperties is NP-hard. Further, we show that for any safety hyperproperty of unbounded length the problem is in general undecidable. In addition, we give a sufficiency condition for the problem to be solved in polynomial time.

- We present rigorous experimental results on monitoring three different security policies on a dataset from Microsoft Research [53] as well as sets of synthetically generated traces. We analyze different metrics such as the length of progressed formulas, the number of generated LTL_3 monitors, and the overhead of runtime monitoring.

1.4 Automated Program Repair for Hyperproperties

Our first contribution is in static enforcement of security policies through *program repair* techniques. Program repair is an automated technique that revises a program with respect to a logical property that the program currently does not satisfy, while preserving its existing semantics. That is, it intends to automated the procedure of fixing a bug (i.e., the second problem mentioned in Section 1.1). Formally, let p be a program, $\mathbb{E}$ be the program's existing specification, and $\boldsymbol{S}$ be a property, where p satisfies $\mathbb{E}$ and does not satisfy $\boldsymbol{S}$. In this context, $\boldsymbol{S}$ is the newly identified requirement for which p will be repaired and $\mathbb{E}$ is

the existing specification of p that should be preserved during repair. A repair algorithm takes as input p, $\boldsymbol{S}$ and $\mathbb{E}$, and generates as output a program p', such that p' satisfies $\mathbb{E}$ and $\boldsymbol{S}$ simultaneously.

A model repair algorithm enhances the robustness and security requirements of the resulting code, while minimizing penalties to code performance and overhead [17]. Ideally, a repair algorithm *fixes* errors with minimal intervention in the behavior of the input program. It is also crucial to ensure that in the repaired program new errors are not introduced such as deadlocks or those that do not allow programs to terminate safely.

1.4.1 Challenges

Program repair is a challenging task as it requires removing transitions from an existing program while preserving existing specifications. Moreover, all the possible sets of executions of a program need to be analyzed to enforce a policy, which increases exponentially with the size of the program.

1.4.2 Contributions

This thesis addresses the complexity of program repair with respect to safety hyperproperties as complexity analysis is often the first step to developing sound and complete automated formal techniques. We focus on a simple but arguably effective type of program repair, where repair is performed only to identify and exclude program traces that violate a given safety hyperproperty. This formulation of program repair has been shown to be quite effective in synthesizing highly complex distributed fault-tolerant protocols [20] and UNITY programs [17].

To analyze the complexity of repair, we classify safety hyperproperties based on two parameters: (1) the (maximum) cardinality of the sets M involved (recall from the previous section that if the maximum cardinality of the set of sequences is at most k, it constitutes a k-safety hyperproperty [34]), and (2) the (maximum) length of sequences that are used to describe the undesirable patterns. Thus, we consider k_ℓ-safety hyperproperties—a subclass of k-safety hyperproperties—where the length of undesirable sequences is at most ℓ. The significance of analyzing the complexity of repair based on k and ℓ is due to expressiveness of k_ℓ-safety hyperproperties. For instance, an example of Goguen and Meseguer's noninterference for one public and one private channel is a 2_1-safety hyperproperty, where

$k = 2$ is the number of channels and $\ell = 1$ stipulates the fact that violation of non-interference can be detected at each state (i.e., the length of the undesirable finite trace is 1). We make the following contributions:

- We show that repairing a program for k-safety hyperproperties is in general NP-hard assuming finite state space of the input program and it is NP-complete if the k-safety hyperproperty can be expressed in a HyperLTL$_1$ formula.
- We show that the problem remains NP-hard even if we consider 2_1-safety hyperproperties.
- We show that the problem remains NP-hard even if we consider 1_3-safety hyperproperties.
- We note that the above NP-hardness results are tight in that the problem can be solved in polynomial time in the state space of the input program if we consider 1_2-safety hyperproperties. The polynomial-time algorithm for 1_2-safety hyperproperties is also tight in that the problem becomes NP-hard if we consider another version, namely, *elongated* safety hyperproperties. It also becomes NP-hard if we desire to preserve the maximum number of traces during the repair.
- We identify a class of *k-generated* safety hyperproperties that are a subclass of k_1-safety hyperproperties for which the problem can be solved in polynomial time in the state space of the input program. This class includes 1-safety hyperproperties, where undesirable prefixes encode unreachability of one pair of predicates. This class includes several important problems such as secret sharing [77].

The significance of these results is that it allows us to understand the complexity of an algorithm when repairing with respect to a certain class of safety hyperproperties. From the given results we can determine whether it is possible to design sound and complete polynomial-time algorithms. If not, one may consider focusing on developing efficient heuristics for repairing for any k_ℓ-safety hyperproperty when $k > 1$ or $k = 1$ and $\ell \geq 3$, among other techniques such as using SMT solvers or program repair for particular classes of programs.

1.5 Organization

The organization of this thesis is as follows: Chapter 2 introduces the necessary preliminaries and definitions required for the understanding of hyperproperties. It formally gives the syntax and semantics of HyperLTL along with the formal definition of a program and trace properties. In Chapter 3, we establish a connection between k-hypersafety and HyperLTL. Following this we present a runtime verification technique of these properties and also discuss the complexity of verification of safety hyperproperties. In Chapter 4, we analyze the complexity of program repair with respect to safety hyperproperties. In Chapter 5, we present the related work on the problem of runtime verification and program repair with respect to safety hyperproperties. Finally, we make concluding remarks and discuss future work in Chapter 6.

Chapter 2

Preliminaries

Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{AP}$ be the finite *alphabet*. We call each element of Σ a *letter* (or an *event*). Throughout the paper, Σ^ω denotes the set of all infinite sequences (called *traces*) over Σ, and Σ^* denotes the set of all finite traces over Σ. $\Sigma^{\leq l}$ denotes the set of all traces over Σ of length at most l. A *state predicate* is any subset of Σ. For a trace $t \in \Sigma^\omega$, $t[i]$ denotes the i^{th} element of t, where $i \in \mathbb{N}$. Also, $t[0, i]$ denotes the prefix of t up to and including i, and $t[i, \infty]$ is written to denote the infinite suffix of t beginning with element i.

Now, let u be a finite trace and v be a finite or infinite trace. We denote the concatenation of u and v by $\sigma = uv$. Also, $u \leq \sigma$ denotes the fact that u is a prefix of σ. Finally, if U is a set of finite traces and V is a finite or infinite set of traces, then the prefix relation $\leq$ on sets of traces is defined as:

$$U \leq V \equiv (\forall u \in U.\ (\exists v \in V.\ u \leq v))$$

Note that V may contain new traces that have no prefix in U.

2.1 Programs

Definition 1. *A* program *is a tuple* $p = \langle I_p, \delta_p \rangle$*, where* $I_p \subseteq \Sigma$ *is the set of* initial states *of* p *and* $\delta_p \subseteq \Sigma \times \Sigma$ *is the set of* transitions *of* p*.* ■

Definition 2. *A sequence of states* $\sigma = \langle s_0 s_1 \dots \rangle$ *in* Σ^ω *is a* trace *of* $p = \langle I_p, \delta_p \rangle$ *iff the following two conditions are satisfied:*

- $s_0 \in I_p$
- $\forall i \geq 0 : (s_i, s_{i+1}) \in \delta_p$ ■

The set of all traces of a program p is denoted by $\psi(p)$.

From the above definition, a program should not contain *deadlocked* traces. That is, if there exists a state s_d from which there is no outgoing transition (including a self-loop), then s_d is a *deadlocked state* and a trace of p that reaches s_d is a *deadlocked trace*. A trace σ is called a *terminating* trace when it *terminates* in state s_t. In such a case, we include the transition (s_t, s_t) in the set of transitions of the program δ_p; i.e., σ can be extended to an infinite trace by stuttering at s_t.

```
int main( ){
    x1 = 1, x2 = 2, . . . , xn = n;
    while(1) {
        current = (rand()%n) + 1;
        switch(current) {
            case x1 : output = Input(P1);
                break;
            case x2 : output = Input(P2);
                break;
                ⋮
            case xn : output = Input(Pn);
        }
        print (output, current);
    }
}
```

Figure 2.1: Secret sharing program p

Running example. A secret Γ has been divided into n shares using a secret sharing scheme [77] and given to n agents, $P_1, P_2, \ldots, P_n$. All n of these shares are required to reconstruct the original secret Γ. The program in Figure 2.1 constantly queries randomly

one out of the n agents for an input; that agent replies with a value for the variable 'output' that is printed. Let us say the observer reading the print statements is an adversary colluding with the malicious agents. The adversary repeatedly tries to construct the secret from the values read. If all the shares are revealed by all of the agents then the secret can be reconstructed by the adversary. It is straightforward to transform this program into its set of states and transitions.

2.2 Trace Properties

A *trace property* is a set of infinite traces (i.e., a subset of Σ^ω). The set of all trace properties is $\mathcal{P}(\Sigma^\omega)$, where $\mathcal{P}$ denotes the powerset. By $\mathcal{P}^*(X)$, we mean the set of all finite-size subsets of X. We assume that for a *program* p, $\psi(p)$ is a set of infinite traces; i.e., $\psi(p) \subseteq \Sigma^\omega$. We say that a program p *satisfies* a property S (denoted $p \models S$) iff $\psi(p) \subseteq S$.

Following Alpern and Schneider, a trace property is an intersection of a *safety* and a *liveness* property [2]. A safety property is characterized by a set of "bad things". A bad thing must be finitely observable and its occurrence can never be remediated by future events. Precisely, a trace property S is a *safety property* [2] iff

$$\forall t \in \Sigma^\omega.(t \notin S) \implies \exists m \in \Sigma^*.(m \leq t) \ \wedge \ (\forall t' \in \Sigma^\omega.(m \leq t') \implies (t' \notin S))$$

Examples.

- The following policy can be expressed as a safety property; "No two processes have permission to write to the same file simultaneously." Once this policy is violated, i.e., two processes acquire permission to write to the same file simultaneously, it can never be remediated in the future.
- *Access control* security policy requires every action of a particular user to be consistent with the rights or privileges granted to that user. This policy is a safety property, where the bad thing is a finite trace with an action taken by an unauthorized user for that action.

A *co-safety* property is defined as follows [14]:

$$\forall t \in \Sigma^\omega.(t \in S) \implies \exists m \in \Sigma^*.(m \leq t) \ \wedge \ (\forall t' \in \Sigma^\omega.(m \leq t') \implies (t' \in S))$$

For every trace in a co-safety property there exists a good prefix such that any continuation of it is in the property.

Example. "Every trace must eventually reach an accepting state" is a co-safety property.

A liveness property is characterized by a "good thing" always possible no matter what has occurred so far, and possibly infinite, so it need not be a discrete event. Precisely, a trace property L is a *liveness property* [2] iff

$$\forall t \in \Sigma^*.\exists t' \in L.(t \leq t')$$

Example. An example of a liveness property is *guaranteed service*, which requires that a request for a service is eventually satisfied. The good thing is the eventual satisfaction of the request.

2.3 Hyperproperties

It is well known that a large number of interesting security policies, such as non-interference and observational determinism, cannot be expressed by trace properties [76]. To overcome this shortcoming, Clarkson and Schneider introduced the notion of *hyperproperties* to incorporate an additional level of sets to the notion of trace properties [34].

Definition 3 (hyperproperty [34])**.** *A* hyperproperty *is a set of sets of infinite traces, or equivalently a set of trace properties.* ■

The set of all hyperproperties is $\mathcal{P}(\mathcal{P}(\Sigma^\omega))$. The interpretation of a hyperproperty as a security policy is that the hyperproperty is the set of programs allowed by that policy. That is, each trace property in a hyperproperty is an allowed system, specifying exactly which executions must be possible for that system. Thus, unlike trace properties, where the notion of *satisfaction* is based on language inclusion, the definition of satisfaction for hyperproperties is based on language equality. More formally,

Definition 4. *A program p satisfies a hyperproperty $\boldsymbol{H}$ (denoted, $p \models \boldsymbol{H}$) iff $\psi(p) \in \boldsymbol{H}$.*

That is, a program satisfies a security policy if and only if its set of traces adheres with one of the entire sets (and not just a subset) of traces of the prescribed policy.

2.3.1 Safety Hyperproperties

Safety hyperproperty (or hypersafety) is a generalization of *safety* [2], where the bad thing occurs in a finite set of finite traces. The definition of hypersafety is essentially the same as the definition of safety, except for an additional level of sets.

Definition 5. *A hyperproperty* $\boldsymbol{S}$ *is a* safety hyperproperty *(or* hypersafety*) iff*

$$\forall T \in \mathcal{P}(\Sigma^\omega).(T \notin \boldsymbol{S}) \implies \exists M \in \mathcal{P}^*(\Sigma^*).(M \leq T) \wedge (\forall T' \in \mathcal{P}(\Sigma^\omega).(M \leq T') \implies (T' \notin \boldsymbol{S}))$$

Example. An example of a safety hyperproperty is *Observational Determinism* as described previously in Chapter 1 [34]. Its formal definition is as follows:

$$\boldsymbol{OD} = \{T \in \mathcal{P}(\Sigma^\omega) \mid \forall t, t' \in T.t[0] =_L t'[0] \implies t \approx_L t'\}$$

where $t \approx_L t'$ is a trace equivalence relation that holds whenever traces t and t' are indistinguishable to a low user. State $t[0]$ is the first state in trace t and $s =_L s'$ is a state equivalence relation that holds whenever states s and s' are indistinguishable to a low user.

In Definition 5, set M represents the *bad thing* that should never happen. If we put a bound on the cardinality of M, it becomes a k-safety hyperproperty defined as follows:

Definition 6 (k-safety hyperproperty [34])**.** *A hyperproperty* $\boldsymbol{S}$ *is a* k-safety hyperproperty *(is* k-hypersafety*) iff*

$$\forall T \in \mathcal{P}(\Sigma^\omega).(T \notin \boldsymbol{S}) \Rightarrow \exists M \in \mathcal{P}^*(\Sigma^*).(M \leq T) \wedge (|M| \leq k) \wedge ((\forall T' \in \mathcal{P}(\Sigma^\omega).(M \leq T') \Rightarrow (T' \notin \boldsymbol{S})) \blacksquare$$

Notice that a traditional safety property [2] is synonymous to a 1-safety hyperproperty [34].

Examples

- A policy that requires 'whenever there is a fail event, then there must not be a login event for at least four time units' is a 1-safety hyperproperty. If one models the passage of every time unit by the event tick, then the bad thing here is a finite trace that contains a fail followed by three or fewer tick events before a login event.

- Goguen and Meseguer's *non-interference* (***GMNI***) [45] and ***OD*** security policy is a 2-safety hyperproperty.
- The *information leakage* policy is an example of a safety hyperproperty. The bad thing is some series of experiments, where the information leaked is more than x bits. Notice that in this example there is no bound on k.

Observe that if a hypersafety is violated, we can identify a finite set of finite traces, such that any extension of that set violates the hypersafety. Hence, *all* the bad things for a safety hyperproperty are specified by a set of finite sets of finite traces, say $\boldsymbol{M_h} \in \mathcal{P}(\mathcal{P}^*(\Sigma^*))$. And, a hypersafety is violated iff the given set of traces extends some element in $\boldsymbol{M_h}$. Thus, a hypersafety $\boldsymbol{S}$ can be characterized by a set $\boldsymbol{M_h}$ such that:

$$\forall T \in \mathcal{P}(\Sigma^\omega).(\exists M \in \boldsymbol{M_h}.(M \leq T) \iff (T \notin \boldsymbol{S})) \tag{2.1}$$

Now, an $\boldsymbol{M_h}'$ that characterizes a k-safety hyperproperty $\boldsymbol{S_k}$, where $\forall M, M' \in \boldsymbol{M_h}'$ it is not the case that $M' \leq M$ and $|M'| < |M|$, is called a *minimal* $\boldsymbol{M_h}$ that characterizes this $\boldsymbol{S_k}$.

Lemma 1. *For every k-safety hyperproperty $\boldsymbol{S_k}$ with a bounded cardinality of $\boldsymbol{M_h}$, there exists an $\boldsymbol{M_h}$ that is a minimal $\boldsymbol{M_h}$ that satisfies Equation 2.1 such that $|M| \leq k$ for all sets M in the set $\boldsymbol{M_h}$.*

Proof. Given an $\boldsymbol{M_h}'$ and $M, M' \in \boldsymbol{M_h}'$ such that $M' \leq M$, removing M from $\boldsymbol{M_h}'$, i.e., $\boldsymbol{M_h}' \backslash \{M\}$ still characterizes $\boldsymbol{S_k}$ using Equation 2.1. This is because any extension of M is also an extension of M'. Further, $\forall M \in \boldsymbol{M_h}$ and $\forall m, m' \in M$ such that $m' \leq m$, removing m' from M still characterizes $\boldsymbol{S_k}$ since m includes the trace m'. Therefore, $\forall M, M' \in \boldsymbol{M_h}'$ if $M' \leq M$, then remove M from $\boldsymbol{M_h}'$, and $\forall m, m' \in M'$ if $m' \leq m$, remove m' from M'. Through this process, we can obtain a minimal $\boldsymbol{M_h}$ satisfying Equation 2.1 such that $\forall M \in \boldsymbol{M_h}$, $|M| \leq k$. ■

Example. If we take the example of observational determinism, $\boldsymbol{M_h}$ is a set of sets of finite traces such that each element of this set contains pairs of finite traces with their initial states containing the same input from a public observer but, some other corresponding states containing differing outputs observable by the public observer. More formally,

$$\boldsymbol{M_{OD}} = \{M \in \mathcal{P}^*(\Sigma^*) \mid M = \{t, t'\}, t[0] =_L t'[0] \ \wedge \ t \not\approx_L t'\}$$

2.3.2 Co-safety Hyperproperties

Intuitively, a *co-safety hyperproperty* (or *co-hypersafety*) stipulates a policy which describes the occurrence of a *good thing* and is a generalization of traditional *co-safety* [54]. Note that, co-safety hyperproperty is an observable hyperproperty as given below [34]:

Definition 7. *Hyperproperty* $\boldsymbol{C}$ *is a co-safety hyperproperty (or co-hypersafety) iff*

$$\forall T \in \mathcal{P}(\Sigma^{\omega}).(T \in \boldsymbol{C}) \Longrightarrow \exists M \in \mathcal{P}^{*}(\Sigma^{*}).(M \leq T) \wedge (\forall T' \in \mathcal{P}(\Sigma^{\omega}).(M \leq T') \Longrightarrow (T' \in \boldsymbol{C}))$$

In Definition 7, set M represents the *good thing* that can happen.

Definition 8 (co-k-safety hyperproperty)**.** *Hyperproperty* $\boldsymbol{C}$ *is a co-*k*-safety hyperproperty (or co-*k*-hypersafety) iff*

$$\forall T \in \mathcal{P}(\Sigma^{\omega}).(T \in \boldsymbol{C}) \Longrightarrow \exists M \in \mathcal{P}^{*}(\Sigma^{*}).(M \leq T) \wedge (|M| \leq k) \ \wedge \ ((\forall T' \in \mathcal{P}(\Sigma^{\omega}).(M \leq T') \Longrightarrow (T' \in \boldsymbol{C})) \ \blacksquare$$

Notice that a co-safety property is synonymous to a co-1-safety hyperproperty [34].

Example The hyperproperty 'for every initial state, there is some terminating trace, but not all traces must terminate' is a co-safety hyperproperty. The good thing here is a set of traces such that for all initial states, a trace in this set terminates. If the number of initial states is restricted to k, then this is a co-k-safety hyperproperty.

2.3.3 Liveness Hyperproperty

Clarkson et al. extended the notion of liveness to sets of traces and defined *liveness hyperproperties* as follows [34]:

Definition 9. *Hyperproperty* $\boldsymbol{L}$ *is a liveness hyperproperty (equivalently, is hyperliveness) iff*

$$(\forall T \in \mathcal{P}^{*}(\Sigma^{*}).\ (\exists T' \in \mathcal{P}(\Sigma^{\omega}).\ T \leq T' \ \wedge \ T' \in \boldsymbol{L}))$$

Example A policy that places a low threshold on the mean response time on a set of traces is a hyperliveness property. A set of traces can always be extended to a set with a lower mean response time.

Policies which bound a resource to a certain value over all traces of a program is hyperliveness. 'The channel capacity is k bits' is an example of this.

2.4 HyperLTL

HyperLTL extends Ltl, and allows explicit quantification over multiple execution traces simultaneously [32]. It eases the expression of hyperproperties syntactically.

2.4.1 Syntax

The set of HyperLTL formulas is inductively defined by the grammar as follows:

$$\varphi ::= \exists \pi.\varphi \mid \forall \pi.\varphi \mid \phi$$
$$\phi ::= a(\pi) \mid \neg\phi \mid \phi \vee \phi \mid \phi \,\mathbf{U}\, \phi \mid \mathbf{X}\phi \mid \phi(\pi)$$

where $a \in AP$ and π is a trace variable from an infinite supply of variables Γ.

We note that we have extended the syntax of HyperLTL by allowing annotation of a *formula* ϕ with trace variable π. This way, HyperLTL allows us to clearly specify to which trace ϕ refers in a formula which includes multiple traces.

Temporal connectives hold on every quantified trace. The intuitive description for $\psi \mathbf{U} \phi$ is that eventually ϕ holds and 'until' then ψ holds, and for $\mathbf{X}\phi$ is in the 'next' state ϕ holds. The other standard temporal connectives are defined as syntactic sugar as follows; $\mathbf{F}\phi$ (eventually ϕ) $\equiv \texttt{true}\,\mathbf{U}\,\phi$, and $\mathbf{G}\phi$ (globally ϕ) $\equiv \neg\mathbf{F}\neg\phi$. Quantified formulas $\exists\pi$ and $\forall\pi$ are read as 'along some trace π' and 'along all traces π', respectively.

2.4.2 Semantics

A formula φ in HyperLTL satisfied by a set of traces T is written as $\Pi \models_T \varphi$, where trace assignment $\Pi : \Gamma \to \Sigma^\omega$ is a partial function mapping trace variables to traces. $\Pi[\pi \to t]$

denotes the same function as Π, except that π is mapped to trace t. The validity judgement is defined as follows:

$$
\begin{array}{lll}
\Pi \models_T \exists \pi.\varphi & \text{iff} & \exists t \in T.\, \Pi[\pi \to t] \models_T \varphi \\
\Pi \models_T \forall \pi.\varphi & \text{iff} & \forall t \in T.\, \Pi[\pi \to t] \models_T \varphi \\
\Pi \models_T a & \text{iff} & a \in \Pi[0] \\
\Pi \models_T \neg\phi & \text{iff} & \Pi \not\models_T \phi \\
\Pi \models_T \phi_1 \vee \phi_2 & \text{iff} & (\Pi \models_T \phi_1) \vee (\Pi \models_T \phi_2) \\
\Pi \models_T \mathbf{X}\phi & \text{iff} & \Pi[1,\infty] \models_T \phi \\
\Pi \models_T \phi_1 \,\mathbf{U}\, \phi_2 & \text{iff} & \exists i \geq 0.\, (\Pi[i,\infty] \models_T \phi_2 \wedge \\
 & & \forall j.\, 0 \leq j < i.\, \Pi[j,\infty] \models_T \phi_1) \\
\Pi \models_T \phi(\pi) & \text{iff} & \Pi(\pi) \models \phi
\end{array}
$$

Here, the trace assignment suffix $\Pi[i,\infty]$ denotes the trace assignment $\Pi' = \Pi(\pi)[i,\infty]$ for all π. If $\Pi \models_T \phi$ holds for the empty assignment Π, then T satisfies ϕ. Observe that when there is exactly one universal trace quantifier, then LTL and HyperLTL coincide.

Notation By $\phi(\pi_1, \ldots, \pi_k)$, we mean the formula $\phi(\pi_1) \vee \cdots \vee \phi(\pi_k)$. Also, let LTL, LTL_S, and LTL_C be the set of all, safety, and co-safety LTL formulas, respectively.

2.4.3 Specifying Trace Relations

Clarkson et al. [32] introduce the trace relation $=_P$ to ease the representation of equivalence between traces. Let π and π' be two trace variables. For a set $P \subseteq AP$ of atomic propositions, $\pi[0] =_P \pi'[0]$ denotes that the first letter in both π and π' agree on all propositions in P. Further

$$\pi =_P \pi' \;\equiv\; \mathbf{G}(\pi[0] =_P \pi'[0])$$

compares the two traces letter by letter for all letters.

Example

- ***GMNI*** can be specified as a HyperLTL formula as follows:

  $$\forall \pi. \forall \pi'. (\mathbf{G}\lambda_{\pi'} \;\wedge\; \pi \neq_H \pi') \;\Rightarrow \pi =_L \; \pi'$$

  where $\mathbf{G}\lambda_{\pi'}$ denotes that all high variables in π' hold the value λ for all letters, and H and L are the 'high' and 'low' atomic propositions, respectively.

- *Observational Determinism* (***OD***) requires a system to appear deterministic to a low user (users who only have access to low variables). It is specified as follows:

$$\forall\pi.\forall\pi'.\,(\pi[0] =_{L,in} \pi'[0]) \;\Rightarrow\; (\pi =_{L,out} \pi')$$

  where $=_{L,in}$ checks for agreement on propositions in L with input values issued by the low user.

Addressing Limitations While the operator $=_P$ for trace relations allows one to specify properties over a pair of traces that check for equivalence letter by letter, it does not capture comparison of some letter in one trace with one or more letters in another trace, or comparison of letters over temporal formulas specified over P.

To address this limitation, we define a function $f : 2^{AP} \to LTL$ and extend the trace relation to $\pi \sim_{f,P} \pi'$, for two trace variables π and π', and a set P of atomic propositions. We require that

$$\forall i.\, \pi'[i..\infty] \models f(\pi[i] \cap P)$$

Obviously, when function f is the identity function $\pi =_P \pi' \equiv \pi \sim_{f,P} \pi'$. In case of ***GMNI***, if we look at the policy which requires an event occurring in one trace to occur somewhere in the other trace, then function f is a mapping of $x \in 2^P$ to $\mathbf{F}x$.

Chapter 3

Runtime Verification of k-Safety Hyperproperties in $HyperLTL$

In this chapter, we present the complexity results along with a polynomial-time algorithm for the runtime verification of safety hyperproperties. We start by syntactically representing safety hyperproperties in HyperLTL and showing the monitorability of formulas in HyperLTL. The complete set of safety hyperproperties falls in two categories based on the $\boldsymbol{M_h}$ that characterizes a safety hyperproperty (see Equation 2.1). For the first category—when $\boldsymbol{M_h}$ is bounded—we present a polynomial-time algorithm for a class of k-safety hyperproperties and show that the problem is NP-hard for another class. For the second category—when $\boldsymbol{M_h}$ is unbounded—we show that the runtime verification problem is in general undecidable. We also present a sufficiency condition for the problem to be decided in polynomial time.

3.1 k-Safety/Co-k-Safety Hyperproperties in HyperLTL

In this section, we establish the connection between the set representation of k-safety and co-k-safety hyperproperties with HyperLTL.

As mentioned earlier, our goal in this paper is to monitor k-safety hyperproperties at run time. To monitor a k-safety hyperproperty, we essentially need to detect the bad things (i.e., $\boldsymbol{M_h}$ in Equation 2.1) in the program under inspection across different executions. Since the bad things are characteristic of the sets of traces that are present in the

complement of a safety hyperproperty, in effect, we need to monitor the satisfaction of these complement sets.

To clarify the above statement, consider the secret sharing scheme example given earlier in Section 2.1:

Example. Suppose, program P introduced in Figure 2.1 is malicious and wants to know the secret. Assuming that all the agents can show malicious behavior and eventually reveal their share when queried for an input. The security policy that '*the system cannot, across all of its executions, output all n shares*' ($\boldsymbol{SS_k}$) can be formulated as a k-safety where $k = n$. This example shows that this k-safety is violated when in each of the M finite trace prefixes, a state is reached which reveals the real share such that each of these states is different, i.e. $|M|$ different processes reveal the real share, and $|M| = k$.

Let a_i $(1 \leq i \leq k)$ be the proposition that share i is revealed. Then the following formula in HyperLTL captures the above policy:

$$\varphi_{\boldsymbol{SS_k}} = \forall \pi_1 \cdots \forall \pi_k. \, (\mathbf{G}\ \neg a_1(\pi_1, \ldots, \pi_k)\ \vee\ \ldots\ \vee\ \mathbf{G}\ \neg a_k(\pi_1, \ldots, \pi_k)) \tag{3.1}$$

where $\pi_1, \ldots, \pi_k$ are trace variables.

As said above, one way to monitor the violation of $\varphi_{\boldsymbol{SS_k}}$ is to check for the satisfaction of the bad things, that is, propositions $a_1 \cdots a_k$ at run time. If some a_i becomes true, then the monitor needs to track the satisfaction of a_j propositions, where $i \neq j$. Notice that here the satisfaction of some a_i resembles the partial occurrence of the bad thing in $\varphi_{\boldsymbol{SS_k}}$. Thus, monitoring, $\varphi_{\boldsymbol{SS_k}}$ boils down to evaluation of its negation HyperLTL formula:

$$\varphi_{\boldsymbol{C_{SS_k}}} = \exists \pi_1 \cdots \exists \pi_k. \, (\mathbf{F}\ a_1(\pi_1, \ldots, \pi_k)\ \wedge\ \ldots\ \wedge\ \mathbf{F}\ a_k(\pi_1, \ldots, \pi_k))$$

at run time. The rest of this section explores the relation between hypersafety and co-hypersafety properties along with their syntactic representations in HyperLTL.

3.1.1 Relation between Hypersafety and Co-hypersafety Properties

We now present a lemma that shows that the complement of a safety (respectively, co-safety) hyperproperty $\boldsymbol{S}$, denoted as $\bar{\boldsymbol{S}}$, is a co-safety (respectively, safety) hyperproperty.

Lemma 2. *The complement of a safety hyperproperty is a co-safety hyperproperty and vice versa. Also, the complement of a k-safety hyperproperty is a co-k-safety hyperproperty and vice versa.*

Proof. Let $\boldsymbol{S}$ be a safety hyperproperty and $\bar{\boldsymbol{S}}$ be its complement set. Let $\boldsymbol{M_h}$ be the bad set of finite sets of finite traces, such that for each $M \in \boldsymbol{M_h}$ and any $T \in \mathcal{P}(\Sigma^\omega)$, where $M \leq T$, we have $T \notin \boldsymbol{S}$. This means that $T \in \bar{\boldsymbol{S}}$ and, hence, every infinite extension of M is in $\bar{\boldsymbol{S}}$. Since any $T \in \bar{\boldsymbol{S}}$ can be associated with such an M, hyperproperty $\bar{\boldsymbol{S}}$ is indeed a co-safety hyperproperty. In fact, the bad thing in $\boldsymbol{S}$ (i.e., set M) becomes the good thing in $\bar{\boldsymbol{S}}$. Finally, if $\boldsymbol{S}$ is a k-safety hyperproperty, since $|M| \leq k$, then $\bar{\boldsymbol{S}}$ trivially becomes a co-k-safety hyperproperty. The other direction from co-hypersafety to hypersafety trivially follows similar proof structure. ■

The above lemma entails that the set $\boldsymbol{M_h}$ that characterizes a k-safety hyperproperty (see Formula 2.1), also characterizes a co-k-safety hyperproperty $\boldsymbol{C}$ as follows:

$$\forall T \in \mathcal{P}(\Sigma^\omega). \forall M \in \boldsymbol{M_h}. (M \leq T) \;\Rightarrow\; (T \in \boldsymbol{C}) \tag{3.2}$$

The next lemma shows a characteristic that allows us to utilize $\boldsymbol{M_h}$ in terms of *co-safety* properties.

Lemma 3. *Let $\boldsymbol{M_h}$ be a set of finite sets of finite traces. Let $M \in \boldsymbol{M_h}$ and $m \in M$. It is the case that the set:*

$$\uparrow m = \{mv \mid v \in \Sigma^\omega\}$$

is a co-safety property.

Proof. First, $\uparrow m$ is a trace property, as it only includes infinite traces. Observe that any trace in $\uparrow m$ is an infinite extension of m. This means that for any infinite trace $t \in \uparrow m$, there exists a finite trace m, such that any infinite continuation of m is an element of $\uparrow m$. Hence, $\uparrow m$ is a co-safety property. ■

3.1.2 Representing k-safety and Co-k-safety Hyperproperties in HyperLTL

Clarkson et al. [32] identified HyperLTL$_n$ as the class of HyperLTL formulas in which the sequence of quantifiers at the beginning of the formula involves at most $n-1$ alternations. We now show that a subset of k-safety and co-k-safety hyperproperties can be expressed as a HyperLTL$_1$ formula.

Lemma 4. *Consider a* HyperLTL$_1$ *formula of the following form:*

$$\varphi_{\boldsymbol{C_k}} = \exists \pi_1 \cdots \exists \pi_k. (\phi_1(\pi_1, \ldots, \pi_k) \;\wedge\; \cdots \;\wedge\; \phi_k(\pi_1, \ldots, \pi_k))$$

where $\pi_1 \cdots \pi_k$ are trace variables and $\phi_1, \ldots, \phi_k \in LTL_C$. Such a formula represents a co-k-safety hyperproperty $\boldsymbol{C_k}$.

Proof. Let $\boldsymbol{C_k}$ be the hyperproperty (i.e., the set of sets of traces) that represents $\varphi_{\boldsymbol{C_k}}$. We will show that $\boldsymbol{C_k}$ is a co-k-safety hyperproperty. We need to show that for any $T \in \boldsymbol{C_k}$, there is a finite set of finite traces M, such that any infinite continuation of M is in $\boldsymbol{C_k}$. From the semantics of HyperLTL, we know that $\forall T \in \boldsymbol{C_k}$, there is a Π such that $\Pi \models_T \varphi_{\boldsymbol{C_k}}$. Therefore, there exist infinite traces $t_1, \ldots, t_k \in T$ that satisfy $\phi_1, \ldots, \phi_k$. Since, $\phi_1, \ldots, \phi_k$ are co-safety properties, there exists an m_i $(1 \leq i \leq k)$ for each $t_i \models \phi_i$, such that

$$\forall t \in \Sigma^\omega. (m_i \leq t \;\Rightarrow\; t \models \phi_i)$$

Now observe that for the set M of all such m_i, any infinite continuation T' of M will satisfy $\varphi_{\boldsymbol{C_k}}$ and hence $T' \in \boldsymbol{C_k}$. Hence, $\boldsymbol{C_k}$ is a co-safety hyperproperty. Finally, since $\varphi_{\boldsymbol{C_k}}$ involves only k trace variables, $\boldsymbol{C_k}$ is a co-k-safety hyperproperty. ■

An immediate corollary of Lemma 4 is that some k-safety hyperproperties can also be represented by a HyperLTL$_1$ formula.

Corollary 1. *Consider a* HyperLTL$_1$ *formula of the following form:*

$$\varphi_{\boldsymbol{S_k}} = \forall \pi_1 \cdots \forall \pi_k. (\phi_1(\pi_1, \ldots, \pi_k) \;\vee\; \cdots \;\vee\; \phi_k(\pi_1, \ldots, \pi_k))$$

where $\pi_1 \cdots \pi_k$ are trace variables and $\phi_1, \ldots, \phi_k \in LTL_S$. Such a formula represents a k-safety hyperproperty $\boldsymbol{S_k}$.

Proof. First, notice that $\neg\varphi_{\boldsymbol{C_k}}$ in Lemma 4 will exactly have the syntax of $\varphi_{\boldsymbol{S_k}}$, where each $\neg\phi_i$ is a safety property. Now, observe that in Lemma 4, $\varphi_{\boldsymbol{C_k}}$ gives the syntactic representation of the co-k-safety hyperproperty $\boldsymbol{C_k}$. It follows that $\neg\varphi_{\boldsymbol{C_k}}$ will be the syntactic representation of a k-safety hyperproperty $\boldsymbol{S_k}$ (from Lemma 2). ■

Theorem 1. *Conjunction (respectively, disjunction) of* HyperLTL$_1$ *formulas, with at most k quantifiers, given by $\varphi_{\boldsymbol{S_k}}$ in Corollary 1 (respectively, $\varphi_{\boldsymbol{C_k}}$ in Lemma 4), is a k-hypersafety property (respectively, co-k-hypersafety property).*

Proof. Let's consider a disjunction of HyperLTL$_1$ formulas

$$\varphi_{\boldsymbol{C}} = \varphi_1 \;\vee\; \ldots \;\vee\; \varphi_n$$

where φ_i $(1 \leq i \leq n)$ is a closed HyperLTL$_1$ formula representing a co-k-hypersafety property $\boldsymbol{C_i}$ as given by Lemma 4. The union of a set of co-safety hyperproperties remains

a co-safety hyperproperty, which translates to taking disjunction of the HyperLTL$_1$ representations of these properties. Hence, $\varphi_{\boldsymbol{C}}$ is a co-k-hypersafety.

Similarly, for k-hypersafety consider a conjunction of HyperLTL$_1$ formulas

$$\varphi_{\boldsymbol{S}} = \varphi_1 \ \wedge \ \ldots \ \wedge \ \varphi_n$$

where φ_i $(1 \leq i \leq n)$ is a closed HyperLTL$_1$ formula representing a k-hypersafety property $\boldsymbol{S_i}$ as given by Corollary 1. The intersection of a set of safety hyperproperties remains a safety hyperproperty, which translates to taking conjunction of the HyperLTL$_1$ representations of these properties. Hence, $\varphi_{\boldsymbol{S}}$ is a k-hypersafety. ■

3.2 Monitorability in HyperLTL

In this section, we define the notion of *monitorability* for HyperLTL formulas. Moreover, we identify classes of HyperLTL$_1$ formulas that are monitorable.

Intuitively, a formula is *monitorable*, if there exist cases where the valuation of the formula can be computed at run time. For example, Ltl formula $\mathbf{GF}p$ is not monitorable, since there is no way to tell at run time, whether or not in the future, p will be visited infinitely often. Likewise, a HyperLTL$_2$ formula of the form $\varphi = \forall\exists\psi$ is not monitorable since one has to have *all* traces of the system to declare either a negative or positive verdict about φ.

On the contrary, formulas in LTL_S (e.g., $\mathbf{G}p$) may be evaluated to *false* at run time and formulas in LTL_C (e.g., $\mathbf{F}p$) may be evaluated to *true* at run time. Also, consider the secret sharing scheme example given earlier in the chapter:

$$\varphi_{\boldsymbol{SS_k}} \quad = \quad \forall\pi_1 \cdots \forall\pi_k. \, (\mathbf{G}\neg a_1(\pi_1, \ldots, \pi_k) \quad \vee \quad \ldots \quad \vee \quad \mathbf{G}\neg a_k(\pi_1, \ldots, \pi_k)) \tag{3.3}$$

This formula is monitorable because, if $a_1 \cdots a_k$ propositions become true in at most any k traces, then $\varphi_{\boldsymbol{SS_k}}$ can be declared as violated permanently for all future executions.

Inspired by the 3-valued linear temporal logic [14] (LTL$_3$), we now define runtime verification semantics for HyperLTL (denoted HyperLTL-3). The semantics utilize three truth values $\mathbb{B}_3 = \{\top, \bot, ?\}$, where '?' means that given the current execution(s) at run time, it is not possible to tell whether the formula is satisfied or violated; i.e., both cases are possible in this or future executions.

Definition 10 (LTL_3 semantics)**.** *Let $u \in \Sigma^*$ denote a finite word. The truth value of an* Ltl *formula φ with respect to u, denoted by $[u \models \varphi]$, is an element of the set $\mathbb{B}_3 = \{\top, \bot, ?\}$, and is defined as follows:*

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

■

Definition 11 (HyperLTL-3 semantics)**.** *Let $M \in \mathcal{P}^*(\Sigma^*)$ be a finite set of finite traces. The truth value of a* HyperLTL *closed formula φ with respect to M, denoted by $[M \models \varphi]$, is an element of the set $\mathbb{B}_3 = \{\top, \bot, ?\}$, and is defined as follows:*

$$[M \models \varphi] = \begin{cases} \top & \text{if } \forall T \in \mathcal{P}(\Sigma^\omega).\, (M \leq T).\, \Pi \models_T \varphi \\ \bot & \text{if } \forall T \in \mathcal{P}(\Sigma^\omega).\, (M \leq T).\, \Pi \not\models_T \varphi \\ ? & \text{otherwise} \end{cases}$$

■

Hence, a HyperLTL formula φ is monitorable, if and only if, there exists a finite set of finite prefixes M, for which a future infinite extension may result in valuation $\top$ or $\bot$. Otherwise, the formula is not monitorable; i.e., evaluation of any M results in ?.

Definition 12 (monitorability)**.** *A* HyperLTL *formula φ is* monitorable *iff*

$$\exists M \in \mathcal{P}^*(\Sigma^*).\, [M \models \varphi] \in \{\bot, \top\}$$ ■

In Definition 12, if φ is of the form $\forall\pi.\phi(\pi)$ or $\exists\pi.\phi(\pi)$, then obviously, the necessary condition for φ to be monitorable is that the inner Ltl formula ϕ must be monitorable. Table 3.1 (respectively, Table 3.2) refers to universally (respectively, existentially) quantified HyperLTL_1 formulas and summarizes their monitorability. We present the formula along with the type of Ltl property the trace variables are quantified over. The formulas considered here, are formed by Boolean and temporal operators applied to formulas in LTL_S (respectively, LTL_C). By exploring the monitorability of various formulas in HyperLTL_1, we see that the set of monitorable formulas in HyperLTL_1 includes

Formula	**Property of ϕ**	$\top$	$\bot$	**Runtime evidence (proof)**
$\forall \pi.\, \phi$	$\phi \in LTL_S$	✗	✓	$\exists M.\, \exists u \in M.\, [u \models \phi] = \bot$
$\forall \pi.\, \phi$	$\phi \in LTL_C - LTL_S$	✗	✗	$\nexists M.\, \exists u \in M.\, [u \models \phi] = \bot$
$\forall \pi_1 \ldots \forall \pi_k.\, (\phi_1(\pi_1) \vee \cdots \vee \phi_k(\pi_k))$	$\phi_1, \ldots, \phi_k \in LTL_S$	✗	✓	$\exists M.\, \exists u_1 \ldots u_k \in M.\, [u_1 \models \phi_1] = \bot \wedge \cdots \wedge [u_k \models \phi_k] = \bot$
$\forall \pi_1 \ldots \forall \pi_k.\, (\phi_1(\pi_1) \wedge \cdots \wedge \phi_k(\pi_k))$	$\exists i (1 \leq i \leq k).\, \phi_i \in LTL_S$	✗	✓	$\exists M.\, \exists u_i \in M.\, [u_i \models \phi_i] = \bot$
$\forall \pi_1 \ldots \forall \pi_k.\, (\phi_1(\pi_1) \vee \cdots \vee \phi_k(\pi_k))$	$\exists i (1 \leq i \leq k).\, \phi_i \notin LTL_S$	✗	✗	$\nexists M.\, \exists u_i \in M.\, [u_i \models \phi_i] = \bot$
$\forall \pi_1 . \forall \pi_2 . (\phi_1(\pi_1)\, \mathbf{U}\, \phi_2(\pi_2))$	$\phi_1, \phi_2 \in LTL_S$	✗	✓	$\exists M.\, \exists u_1, u_2 \in M.\, [u_1, u_2 \models \phi_1(u_1)\, \mathbf{U}\, \phi_2(u_2)] = \top$

Table 3.1: Monitorability of universally quantified HyperLTL$_1$ formulas.

properties outside of k-safety and co-k-safety hyperproperties. For example, the formula $\forall \pi_1.\, \forall \pi_2.\, (\mathbf{G} p(\pi_1) \wedge \mathbf{F} q(\pi_2))$ is neither a safety hyperproperty nor a co-safety hyperproperty. However, it is monitorable and can be declared violated at run time.

Along with this, we provide the evidence that can show whether a formula can be satisfied or falsified at run time, based on Definition 12. Observe that this evidence is, in fact, the proof of monitorability of the formula as well. For example:

- For the formula $\varphi = \forall \pi.\, \phi$ (first row of Table 3.1), if $\phi = \mathbf{G} p$ (i.e., an LTL safety property), then a finite trace $u = p \ldots \neg p$ violates ϕ. The existence of such a monitorable finite trace in a set of finite traces, i.e., $M = \{p \ldots \neg p\}$, violates φ as well. Hence, φ is monitorable as well. Such a formula, however, cannot be declared satisfied at run time since that would require monitoring every trace in the infinite domain of π.

- In Table 3.1, for the formula in the third row, the runtime evidence that violates the formula is a set of finite traces such that every property $\phi_1, \ldots, \phi_k$ is violated by a finite trace in this set. The secret sharing example (see Section 3.1) corresponds to this property which is violated if a set of finite traces reveal each of the k shares. Also, ***GMNI*** and ***OD*** fall in this category of formulas.

- In Table 3.2, formulas such as $\varphi = \exists.\, \phi$, if $\phi = \mathbf{F} p$, then a finite trace $\ldots p$ satisfies ϕ and hence, φ. Even though φ is monitorable, it cannot be falsified at run time.

Formula	**Property of ϕ**	$\top$	**Runtime evidence (proof)**	$\bot$
$\exists \pi.\, \phi$	$\phi \in LTL_C$	✓	$\exists M.\, \exists u \in M.\, [u \models \phi] = \top$	✗
$\exists \pi.\, \phi$	$\phi \in LTL_S - LTL_C$	✗	$\nexists M.\, \exists u \in M.\, [u \models \phi] = \top$	✗
$\exists \pi_1 \ldots \exists \pi_k.\, (\phi_1(\pi_1) \wedge \cdots \wedge \phi_k(\pi_k))$	$\phi_1, \ldots, \phi_k \in LTL_C$	✓	$\exists M.\, \exists u_1 \ldots u_k \in M.\, [u_1 \models \phi_1] = \top \wedge \cdots \wedge [u_k \models \phi_k] = \top$	✗
$\exists \pi_1 \ldots \exists \pi_k.\, (\phi_1(\pi_1) \vee \cdots \vee \phi_k(\pi_k))$	$\exists i (1 \leq i \leq k).\, \phi_i \in LTL_C$	✓	$\exists M.\, \exists u_i \in M.\, [u_i \models \phi_i] = \top$	✗
$\exists \pi_1 \ldots \exists \pi_k.\, (\phi_1(\pi_1) \wedge \cdots \wedge \phi_k(\pi_k))$	$\exists i (1 \leq i \leq k).\, \phi_i \notin LTL_C$	✗	$\nexists M.\, \exists u_i \in M.\, [u_i \models \phi_i] = \top$	✗
$\exists \pi_1 . \exists \pi_2 . (\phi_1(\pi_1)\, \mathbf{U}\, \phi_2(\pi_2))$	$\phi_1, \phi_2 \in LTL_C$	✓	$\exists M.\, \exists u_1, u_2 \in M.\, [u_1, u_2 \models \phi_1(u_1)\, \mathbf{U}\, \phi_2(u_2)] = \top$	✗

Table 3.2: Monitorability of existentially quantified HyperLTL$_1$ formulas.

Note that, the satisfaction (respectively, violation) of a HyperLTL formula with a $\forall$ (respectively, $\exists$) quantifier cannot be declared for any program if the domain of the bounded trace variable is infinite.

Observe that the formulas on highlighted rows in both tables are not monitorable. The third row formulas, in Tables 3.1 and 3.2, correspond to monitorable k-safety and co-k-safety hyperproperties, respectively. The fourth formula, in Table 3.1 (respectively, Table 3.2) is an example of a formula that is neither a k-hypersafety nor a co-k-hypersafety property if $\exists i, j.\, \phi_i \in LTL_S$ and $\phi_j \notin LTL_S$ (respectively, $\exists i, j.\, \phi_i \in LTL_C$ and $\phi_j \notin LTL_C$).

Note that, these tables do not capture all formulas in HyperLTL$_1$, and only shows some relevant ones pertaining to monitorability of k-safety hyperproperties. However, the monitorability of all other formulas can be derived from Definition 12 and the given tables.

In Table 3.3, we take disjunctions and conjunctions of formulas from Tables 3.1 and 3.2. The runtime evidence for whether a formula of the given syntactic form can be declared satisfied or violated at run time follows trivially.

Theorem 2. *Every k-safety hyperproperty and every co-k-safety hyperproperty that satisfies Theorem 1 is monitorable.*

Proof. The proof follows from whether $\exists M \in \mathcal{P}^*(\Sigma^*)$ satisfies Definition 12 for a k-safety or co-k-safety hyperproperty, that is syntactically represented in HyperLTL$_1$ by formulas

Formula	Property of ϕ	$\top$	$\bot$
$\forall\pi_1.\,\phi_1 \;\vee\; \exists\pi_2.\,\phi_2$	$\phi_1 \in LTL_S, \phi_2 \in LTL_C$	✓	×
$\forall\pi_1.\,\phi_1 \;\wedge\; \exists\pi_2.\,\phi_2$	$\phi_1 \in LTL_S, \phi_2 \in LTL_C$	×	✓
$\forall\pi_1.\,\phi_1 \;\vee\; \exists\pi_2.\,\phi_2$	$\phi_1 \in LTL_S, \phi_2 \notin LTL_C$	×	×
$\forall\pi_1.\,\phi_1 \;\wedge\; \exists\pi_2.\,\phi_2$	$\phi_1 \notin LTL_S, \phi_2 \in LTL_C$	×	×

Table 3.3: Monitorability of HyperLTL formulas with conjunction or disjunction of formulas from Tables 3.1 and 3.2

given in Theorem 1. The runtime evidence for these hyperproperties in Tables 3.1 and 3.2 shows that such an M indeed exists for every k-safety and co-k-safety hyperproperty. ■

We note that the above classification also includes HyperLTL$_1$ formulas that are monitorable, but are neither k-hypersafety nor co-k-hypersafety.

Theorem 3. *The set of all monitorable* HyperLTL$_1$ *formulas is strictly larger than the set of monitorable k-hypersafety and co-k-hypersafety properties.* ■

3.3 Complexity of Verification of Safety Hyperproperties at Run Time

In this section, we consider safety hyperproperties in general, not just k-safety hyperproperties. An example from security is as follows, "for any k, a system cannot output all k shares of a secret from a k-secret sharing."

$$\boldsymbol{SecS} = \bigcup_{k} \boldsymbol{SS_k}$$

An important characteristic of this class of hyperproperties is that they are refinement closed or subset closed, i.e., any subset of an element of the set characterizing safety hyperproperties is also in the set. This class of hyperproperties includes the whole class of k-safety hyperproperties as well as some liveness hyperproperties.

Until now we discussed and showed polynomial-time algorithms for safety hyperproperties with bounded 'k' and $\boldsymbol{M_h}$ with bounded cardinality. Here, we discuss complexity class for when k becomes unbounded or when $\boldsymbol{M_h}$ has unbounded cardinality.

3.3.1 Undecidability

Here, we show that a sound and complete runtime monitoring algorithm to check for the satisfaction or violation of a safety hyperproperty with unbounded number of sets of bad traces is unachievable. Let us call the runtime verification of such a hyperproperty RV.

Given a system that generates a finite set of finite traces T and a safety hyperproperty, $\boldsymbol{S}$, characterized by $\boldsymbol{M_h}$, decide whether the property is violated.

Our reduction is from a fixed arbitrary undecidable language.

Theorem 4. *Runtime monitoring of co-safety hyperproperties and safety hyperproperties is in general undecidable.*

Proof. Towards this end, we present a mapping from an arbitrary instance of an undecidable language $\mathtt{L_U}$ to an instance of RV in which the $\boldsymbol{M_h}$ characterizing a safety hyperproperty has unbounded cardinality. Let $\mathtt{L_U} = \{w_0, w_1, \dots\}$ be a known undecidable language.

First, we use a function $f : w \to T$, such that $w \in \Sigma^*$ and $T \in \mathcal{P}^*(\Sigma^*)$ to associate with every word w a finite set of finite traces over the same alphabet. The function f is such that it maps a word w to the set $\{w\}$. Note that f is a computable function. We will now construct the safety hyperproperty $\boldsymbol{S}$, characterized by an infinite set of sets of finite traces $\boldsymbol{M_h}$. Let us say that $\boldsymbol{M_h} = \bigcup_{i=0}^{\infty}\{f(w_i) \mid w_i \in \mathtt{L_U}\}$, contains the bad set of finite traces corresponding to the word w_i, $i \geq 0$. Hence, the RV problem reduces to showing the undecidability of the following language: $\mathtt{L_S} = \{T \mid f(w) \leq T, \forall w \in \mathtt{L_U}\}$

We will show that if $\mathtt{L_S}$ is decidable then the language $\mathtt{L_U}$ is decidable.

Assume that there exists a decider RV for the language $\mathtt{L_S}$.

- Given input word w.
- Now, construct a new TM D as follows:
- Run RV on $f(w)$, i.e., a finite set of finite traces.
  - If RV accepts, then accept
  - If RV rejects, then reject

Clearly, D is a decider for the language $\mathtt{L_U}$, which is a contradiction. Hence, RV is also undecidable. ■

3.3.2 NP-hardness for a Subclass of k-safety Hyperproperties

For a subclass of k-safety hyperproperties, we now show that the complexity class of solving the runtime verification decision problem RV is NP-hard. Our reduction is from the SUBSET SUM problem:

Given a set of n natural numbers $X = \{x_1, x_2, \ldots, x_n\}$ and a number Y, is there a subset of X that adds up exactly to Y?

Theorem 5. *The runtime verification problem for certain safety hyperproperties is NP-hard.*

Proof. Consider a k-hypersafety $\boldsymbol{S_k}$, which says that *'Security is violated if the outputs from multiple executions of the program exactly add up to number $Y' \in \mathbb{N}$'*. The set $\boldsymbol{M_h}$ that characterizes this hyperproperty contains sets with elements that add up to Y. Let us call the decision problem for this instance RV_k.

Toward this end, we present a mapping from an arbitrary instance of the SUBSET SUM problem to RV_k:

Assume that so far we have seen a finite set of finite traces M. Corresponding to each $x_i \in X$, let there exist a finite trace $t_i \in M$ that outputs value $v(t_i) = x_i$, and let $Y = Y'$.

We now show that the instance of SUBSET SUM has a solution iff there exists an answer to the runtime verification problem for the given instance, i.e.,

$$\exists A \subseteq X \,.\, \sum_{i \in A} x_i = Y \iff [M \models \boldsymbol{S_k}] = \{\top, \bot\} \tag{3.4}$$

Now, we distinguish two cases:

($\Rightarrow$) Assume that $A \subseteq X$ is chosen such that $\sum_{i \in A} x_i = Y$. Clearly, we can see that $\exists M' \subseteq M$ such that $\forall t \in M'$, $v(t) \in A$ then $[M \models \boldsymbol{S_k}] = \bot$. Note that we can never report $[M \models \boldsymbol{S_k}] = \top$ as some $T \in \mathcal{P}(\Sigma^\omega) \,.\, M \leq T$ might still violate $\boldsymbol{S_k}$.

($\Leftarrow$) Let us say a decision $[M \models \boldsymbol{S_k}] = \bot$ is given as $\exists M' \subseteq M$ such that $\sum_{t \in M'} v(t) = Y$. We can report a 'yes' to the SUBSET SUM problem since a set A must exist that exactly contains every $v(t)$, $\forall t \in M'$. Since every $v(t)$ is an element of X, we have a set $A \subseteq X$ such that $\sum_{i \in A} x_i = Y$.

Hence, the runtime verification problem for a subclass safety hyperproperties is NP-hard. ■

3.3.3 A Sufficient Condition for Polynomial-Time Runtime Verification

Here, we present a sufficiency condition for runtime verification of a safety hyperproperty in polynomial time.

First, we use a function to describe the 'badness' of a finite trace. Essentially, this badness allows us to measure the contribution of a single finite trace in making a finite set of finite traces violate the given safety hyperproperty. Since, in security policies a single trace itself might not be bad at all, but in a set of finite traces it may be bad, this function also takes into account a context T, that is a finite set of finite traces, when giving a value to a single trace.

Definition 13. *A function $f_T : \Sigma^* \to \mathbb{Z}^*$ maps a finite trace in Σ^* to a value in $\mathbb{Z}^*$ for a context $T \in \mathcal{P}^*(\Sigma^*)$. The function f is such that for a given trace $t \in \Sigma^*$, $f_T(t) \leq f_T(t')$ for every $t' \in \Sigma^*. t \leq t'$ and $f_U(t) \leq f_V(t)$ for any $U, V \in \mathcal{P}^*(\Sigma^*)$ where $U \leq V$.*

Here, $\mathbb{Z}^*$ represents the set of all non-negative integers and the mapping is to $\mathbb{Z}^*$ since the domain of all finite traces is countably infinite. Next, we can evaluate the 'badness' of the given finite set of finite traces by summing the values given to each individual trace in the set. Here too, the range is the set of non-negative integers.

Definition 14. *Let $F : \mathcal{P}^*(\Sigma^*) \to \mathbb{Z}^*$ be a monotone function, where $F(T) = \sum_{t \in T} f_T(t)$. Note that for all $U, V \in \mathcal{P}^*(\Sigma^*)$ if $U \leq V$, then $F(U) \leq F(V)$.*

Using the functions defined above, a safety hyperproperty can be mapped to set of values on the non-negative integer number line.

Running Example Consider the following example of a safety hyperproperty: 'Over every series of executions of system, the quantity of information leak should not exceed x bits'. A system in which the sum of bits of information leak exceeds x violates the policy. For every trace t that leaks some q number of bits, $f_T(t) = q$. For a given set of traces, $F(T)$ evaluates to the total number of bits leaked by this set.

Definition 15. *Let $H : \mathcal{P}(\mathcal{P}^*(\Sigma^*)) \to \mathcal{P}(\mathbb{Z}^*)$ be a function mapping a set of finite sets of finite traces to a set of non-negative integers. Hence, any safety hyperproperty characterized by $\boldsymbol{M_h}$ defines a set of bad points on the non-negative integer number line given as follows:*

$$H(\boldsymbol{M_h}) = \{F(M) : M \in \boldsymbol{M_h}\}$$

The following lemmas show that for a given safety hyperproperty and a given finite set of finite traces, the runtime verification problem can be solved in polynomial time, in the size of the given set of traces and the HyperLTL formula for the hyperproperty, if the mapping of the safety hyperproperty to a range in the set of non-negative integers is a single interval going up to infinity.

Lemma 5. *For any given hypersafety $\boldsymbol{S}$, characterized by $\boldsymbol{M_h}$ of bounded cardinality, and a given finite set of finite traces $T \subseteq \Sigma^*$, if there exists a function f_T computable in polynomial time that satisfies Definition 13 such that $\exists x \in \mathbb{Z}^* : \inf H(\boldsymbol{M_h}) = x$, and $\forall y > x, y \in H(\boldsymbol{M_h})$, then $[T \models \boldsymbol{S}]$ can be evaluated in polynomial time in the size of the input traces T.*

Proof. From Definition 14, $F(\cdot)$ is also a computable function in polynomial time since it is simply a summation of $f.(t)$ of each trace t in the given finite set of traces. Let $H(\boldsymbol{M_h}) = [x, +\infty)$, for some $x \in \mathbb{Z}^*$ and the given $\boldsymbol{M_h}$. Now, to verify if T violates $\boldsymbol{S}$, i.e., $[T \models \boldsymbol{S}] = \bot$, we need to check if $\exists T' \in \mathcal{P}^*(\Sigma^*)$ where $T' \leq T$, such that $F(T') \in H(\boldsymbol{M_h})$. From Definition 14, if $T' \leq T$, then $F(T') \leq F(T)$. Hence, if there does exist a T' such that $F(T') \in H(\boldsymbol{M_h})$, then $F(T) \in H(\boldsymbol{M_h})$ because every number $> F(T')$ is also in $H(\boldsymbol{M_h})$. Therefore, checking $F(T) \geq x$ is sufficient to verify violation. This algorithm clearly runs in polynomial time. ■

In the running example, $H(\boldsymbol{M_h})$ of such a policy is $[x, \infty)$ which can be evaluated in polynomial time.

3.4 Monitoring Algorithm

In this section, we present our algorithm for monitoring k-safety and co-k-safety hyperproperties given by Theorem 1. We start with the sketch of the algorithm.

Algorithm sketch Our algorithm has three key elements. Consider the formula given by Corollary 1.

- In order to monitor such a formula, due to the existence of trace quantifiers, each sub-formula ϕ_i, where $1 \leq i \leq k$, needs to be monitored independently, possibly across different executions. For example, in $\boldsymbol{OD}$, if the initial state of any two pairs of executions correspond to a low input, then the monitor must be able to identify the

initial state of an execution so that it can watch the rest of both executions to ensure that only low outputs are produced. Thus, we assume that our monitoring algorithm is notified when an execution terminates and when a new execution commences.

- In order for the monitor to memorize and combine the independent evaluation of each sub-formula across different executions at run time, we utilize a *Petri net* (see Fig. 3.1 for secret sharing monitor). That is, in the formula given by Corollary 1, on-the-fly evaluation of each ϕ_i (achieved by a component that monitors ϕ_i) becomes the input to a transition of the Petri net. When all inputs are enabled, the transition executes, indicating that the security policy is violated.

- If there exists a trace relation $\sim_{f,p}$ in the formula, then monitoring an execution may depend on evaluation of past and future executions. Hence, it cannot be monitored in isolation. To tackle this problem, we propose a formula *progression* technique, which constructs a formula to be monitored or progressed in the future executions. In such cases, the monitor structure evolves over time (see Fig. 3.3).

In the remainder of this section, we first describe our progression technique in Section 3.4.1. Then, Section 3.4.2 introduces our monitoring algorithm. For properties given in Tables 3.1-3.3, that are not k-safety or co-k-safety hyperproperties, we present the monitoring algorithm in Section 3.4.3 by expanding on the presented technique.

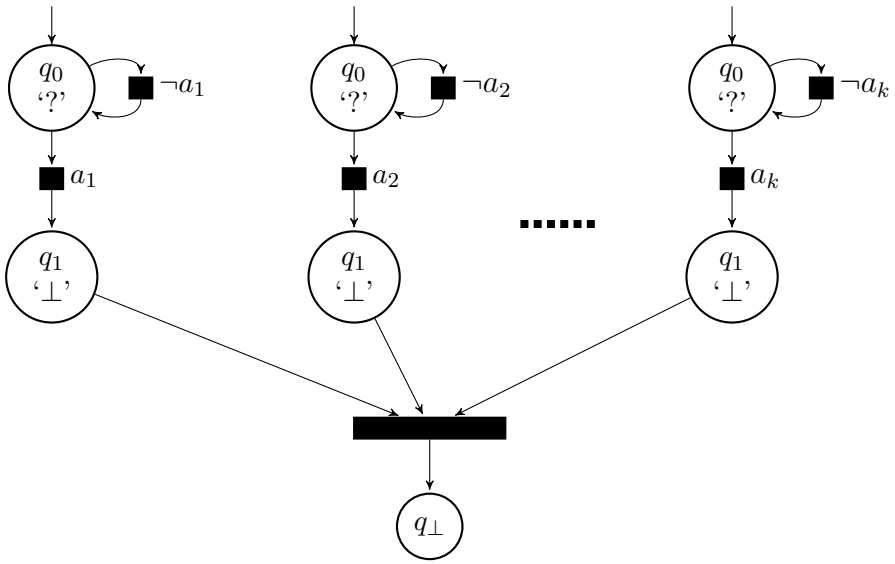


Figure 3.1: Monitor of secret sharing policy.

3.4.1 Progression for Trace Relations

Recall that in Section 2.4.3, we introduced the syntactic operator $\sim_{f,P}$ to address the limitations presented by the operator $=_P$. An example of the use of this operator in a HyperLTL formula is the sixth formula in Table 3.2 which is a co-k-safety hyperproperty. It can be expressed in the syntactic form given by Lemma 4 as $\exists\pi_1.\exists\pi_2.(\pi_1 \sim_{f,P} \pi_2)$, where f maps the last observed ϕ_1 in trace π_1 to $\mathbf{X}\phi_2$.

Unlike existing techniques for formula rewriting [48] and progression [6], which split a formula into goals for the current state and future goals, our formula progression method constructs a formula for execution traces based on the goals satisfied by the currently seen executions. Formally, let $\{u_1, u_2, \dots\}$ be a set of finite traces (representing a set of program executions at run time) and

$$\pi_1 \sim_{f,P} \pi_2 \sim_{f,P} \cdots \sim_{f,P} \pi_n$$

be a trace relation in some HyperLTL$_1$ formula φ to be monitored. We define the *progression function* $\boldsymbol{Pg} : 2^P \to LTL$ inductively as follows:

$$\begin{cases} \boldsymbol{Pg}(u_1[0]) = f(u_1[0] \cap P) \\ \boldsymbol{Pg}(u_j[i+1]) = \boldsymbol{Pg}(u_j[i]) \wedge \mathbf{X}^{i+1} f(u_j[i+1] \cap P) & \text{if } (1 \leq i) \wedge (1 \leq j \leq n-1) \\ \boldsymbol{Pg}(u_{j+1}[i]) = \boldsymbol{Pg}(u_j[\mathit{length}(u_j)]) \wedge f(u_{j+1}[i] \cap P) & \text{if } (i = 0) \wedge (1 < j < n-1) \end{cases}$$

On observing an event of any trace, the progression function is applied according to one of the three cases:

- The first case handles the very first event in the very first trace at run time.
- The second case handles progression within a trace, adding $i+1$ number of next operators applied on f.
- The third case shows how progression is transferred from one execution to the next (up to n times for each trace). Thus, the progression of every new trace depends upon the progression of all the previously observed traces.

Essentially, since the trace relation involves a set of n trace variables in φ, the progression function is applied to every subset of size n of the set of program executions. For instance, for monitoring a 2-safety hyperproperty over m execution traces, $\boldsymbol{Pg}$ needs to be applied $\binom{m}{2}$ times.

3.4.2 Algorithm

Let $\varphi = \forall \pi_1 \cdots \forall \pi_k.\, \phi_1 \vee \cdots \vee \phi_k$ be the HyperLTL$_1$ formula to be monitored. We first categorize formulas that require the progression logic and those that do not, as they will be handled differently. A sub-formula ϕ_i, $1 \leq i \leq k$, is called *observation independent* if it does not contain the $\sim_{f,P}$ relation. For example, all the sub-formulas in the secret sharing scheme example are observation independent. Otherwise, the sub-formula is *observation dependent*. For example, in ***OD*** the formula $\pi \sim_{f,L} \pi'$ is observation dependent. An observation-independent formula is monitored by an LTL$_3$ monitor.

Definition 16 (LTL$_3$ Monitor [14])**.** *Let ϕ be an* Ltl *formula over alphabet Σ. The* monitor $\mathcal{M}^\phi$ *of ϕ is the unique deterministic finite-state automaton (DFA) $(\Sigma, Q, q_0, \Delta, \lambda)$, where Q is a set of states, q_0 is the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and λ is the function $\lambda : Q \to \mathbb{B}_3$, such that for any $u \in \Sigma^*$:*

$$[u \models \phi] = \lambda(\Delta(q_0, u))$$

■

Bauer et al. [14] propose an automated technique to construct such a monitor for a given Ltl formula. For example, Fig. 3.2 shows the LTL$_3$ monitor for formula $a \,\mathbf{U}\, b$.

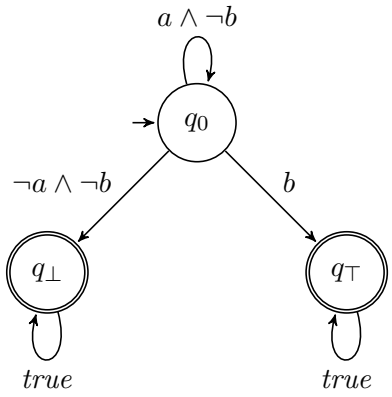


Figure 3.2: LTL$_3$ monitor for formula $a \,\mathbf{U}\, b$.

For an observation-dependent sub-formula, first, a finite trace progresses the formula using the progression function ***Pg*** and then an Ltl monitor is constructed for the progressed formula. As mentioned in the algorithm sketch, we use Petri nets to combine the result of evaluation of inner Ltl formulas in a HyperLTL$_1$ formula across multiple executions.

Definition 17. *A (1-Safe)* Petri net *is defined by a triple* $S = (L, \Sigma, \Delta)$ *where* L *is a set of places,* Σ *is the alphabet, and* $\Delta \subseteq 2^L \times \Sigma \times 2^L$ *is a set of transitions. A transition* τ *is a triple* $({}^\bullet\tau, \sigma, \tau^\bullet)$*, where* ${}^\bullet\tau$ *is the set of input places of* τ *and* $\tau^\bullet$ *is the set of output places of* τ*.* ■

Algorithm 1 provides an online monitoring procedure for an input k-safety HyperLTL$_1$ formula φ with the syntax in Corollary 1. It outputs a value in $\{?, \bot\}$. Recall that a k-safety formula can never be evaluated to $\top$ at run time[1].

We utilize the 'termination-sensitive' policy as a running example to demonstrate the steps of our algorithm:

> *"If any execution of a machine reaches a terminating state, then no two other executions starting from the same initial state should reach differing accepting state"*

This policy is the following 3-safety hyperproperty:

$$\varphi = \forall\pi_1\forall\pi_2\forall\pi_3\,.\,\mathbf{G}t(\pi_1) \;\vee\; \pi_2 \sim_{f,P} \pi_3$$

where t is a proposition for 'not in terminating state' and function f captures the latter part of our policy. In this formula, the set of observation-independent sub-formulas is $\alpha = \{\mathbf{G}t\}$ and the set of observation-dependent sub-formulas is $\beta = \{\pi_2 \sim_{f,P} \pi_3\}$. Fig. 3.3 shows the Petri net that would be created on observing n independent executions. Also, Fig. 3.1 shows the monitor for secret sharing scheme, for which the corresponding HyperLTL$_1$ formula does not include any observation-dependent sub-formulas.

We now describe the algorithm in detail:

1. (Algorithm 1: Line 1) Initially, the set of places in the Petri net (which will be constructed on the fly) contains a final place $q_\bot$ denoting the violation of φ and the output λ is unknown ('?').

2. (Algorithm 1: Lines 2-6) For every observation-independent sub-formula, an LTL$_3$ monitor (a component of the Petri net) is constructed using `construct_component` (Algorithm 2). The leftmost net in Fig. 3.3 (for $\mathbf{G}t$) and all the nets in Fig. 3.1 (for each $\mathbf{G}\neg a_i$) are constructed in this step. Then, for every observation-dependent

[1]Unless all program traces are examined, which would essentially be exhaustive model checking and not runtime verification.

sub-formula μ, a new state q^{μ} is added to the Petri net, which is also an input place for a transition to the output place $q_{\perp}$. For example, this results in place q^{μ} for sub-formula $\mu = \pi_2 \sim_{f,P} \pi_3$ in Fig. 3.3.

3. (Algorithm 2: Lines 1-13) The procedure `construct_component` creates components of the Petri net. It creates an LTL_3 monitor for the first argument, the states and transitions of which are also the places and transitions of the Petri net, respectively (Lines 3-4). If the second argument μ is *true*, then the state q of this component that evaluates to $\perp$ becomes an input place for a transition to the output place $q_{\perp}$ (e.g., in the net that corresponds to Gt in Fig. 3.3) (Lines 9-11). Otherwise, q is *merged* with an existing state q^{μ} using procedure `merge`. This procedure makes any incoming or outgoing transitions to q now point to or from q^{μ}, respectively. The state q is then removed (Lines 6-8). Every component is added to a list of components `comps`. Hence, in our example, after this step `comps` contains only $\mathcal{M}^{\mathbf{G}t}$ (Line 12).

4. (Algorithm 1: Lines 8-11) The monitor continuously gets an event e for evaluation (using `get_input`) from the system under inspection. If the current event marks the beginning of a new trace, then function `reset` re-initializes every component in the list `comps` by moving the token to their initial state. We note that in Lines 16 and 18, components whose current state evaluates to '?' or '$\top$' are removed from `comps`. Next, a new set β' is initialized to β to perform progression on formulas in β' without modifying the original formulas.

5. (Algorithm 1: Lines 12-18) On getting our first event (and thereafter on every event), the function `evaluate` performs transitions on every component in `comps` from the current state and obtains a new state q. If q evaluates to $\perp$, then we check for whether q is actually one of the q^{μ} states. This is done to remove from our list any component whose '$\perp$' state is reached. In our example, since we currently have only monitor $\mathcal{M}^{\mathbf{G}t}$ in `comps`, if indeed our event was $\neg t$, then we remove this component from `comps`.

6. (Algorithm 1: Lines 19-20) Next, we iterate over all observation-dependent sub-formulas in β' and β, such that $\mu' \in \beta'$ is the corresponding formula for $\mu \in \beta$. Formula μ' is progressed over e (and stored within β' itself) by applying the progression function $\boldsymbol{Pg}$ on μ'. By doing this, we are able to capture according to our running example, the initial state (i_j) and subsequently the accepting state (o_j) for an execution.

7. (Algorithm 1: Lines 21-22) Then, `progression_complete`() function returns whether or not the trace relation involving μ' has progressed for all the trace variables it was

defined over. If so, an LTL_3 monitor is constructed for the progressed formula and added to the Petri net. In the example, when progression completes for the second trace (similarly for subsequent traces), then a new component is created for the formula $i_2 \ \wedge \ \mathbf{XX} \ldots o_2$, whose state q, where $\lambda(q) = \bot$, is merged with q^μ, where $\mu = \pi_2 \sim_{f,P} \pi_3$ (recall that this state was introduced initially, for every $\mu \in \beta$).

8. (Algorithm 1: Lines 23-24) If all the input places of the transition to the output place $q_\bot$ contain a token, the transition executes and the monitor returns with the final evaluation $\bot$, meaning that the formula has been falsified.

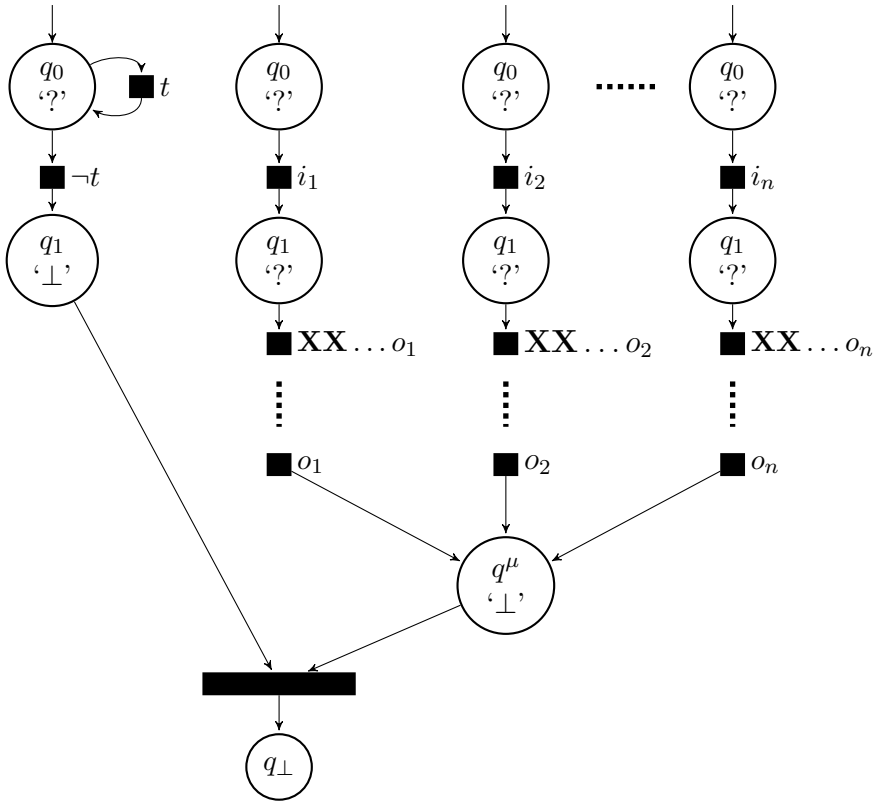


Figure 3.3: Petri net for property termination-sensitive

Observe that, monitoring a co-k-hypersafety follows an identical algorithm except that (1) the state $q_\bot$ is replaced by $q_\top$, denoting satisfaction, (2) all $\bot$'s become $\top$'s, and (2) the token is placed in the final state of a component if it evaluates to $\top$.

Theorem 6. *Let φ be a k-safety* HyperLTL$_1$ *formula. Algorithm 1 returns $\bot$ for an input set $M \in \mathcal{P}^*(\Sigma^*)$ iff $[M \models \varphi] = \bot$.*

Proof. Let $\varphi = \forall \pi_1 \ldots \forall \pi_k.\, \phi_1(\pi_1) \vee \cdots \vee \phi_k(\pi_k)$.

- ($\Rightarrow$) For an input set $M \in \mathcal{P}^*(\Sigma^*)$, where $[M \models \varphi] = \bot$, by contradiction, let us assume that Algorithm 1 returns '?'. The antecedent implies that for all ϕ_i where $(1 \leq i \leq k)$, there exists $m \in M$ such that any extension of m violates ϕ_i. If the algorithm has not yet returned $\bot$, then there exists at least one component M^ϕ of the Petri net that has not yet reached state q such that $\lambda^\phi(q) = \bot$. From Definition 16, we know that the component M^ϕ in the Petri net reports violation if $[m \models \phi_i] = \bot$ contradicting that even though m is observed, component M^ϕ does not report violation. Therefore, if $[M \models \varphi] = \bot$, then Algorithm 1 returns $\bot$.
- ($\Leftarrow$) If Algorithm 1 returns $\bot$, by contradiction, let us assume that $[M \models \varphi] \neq \bot$. The antecedent implies that all input places q^ϕ that have a transition to $q_\bot$, i.e., $(q, \mathit{true}, q_\bot)$, contain a token. By construction of component M^ϕ (Definition 16), on running m over M^ϕ state q^ϕ, such that $(q^\phi, \mathit{true}, q_\bot)$, is reached iff $[m \models \phi] = \bot$. Therefore, for each ϕ_i, where $(1 \leq i \leq k)$, there exists $m_i \in M$, such that $[m \models \phi_i] = \bot$. Therefore, for a trace assignment Π and $\forall T \in \mathcal{P}(\Sigma^\omega)$ such that $\pi_i \to m_i t$ for some $t \in \Sigma^\omega$, we have $m_i t \in T$, and for all $1 \leq i \leq k$, we have $\Pi \not\models_T \varphi$. This contradicts the assumption that $[M \models \varphi] \neq \bot$ from Definition 11. Hence, if Algorithm 1 returns $\bot$ then $[M \models \varphi] = \bot$.

■

Observation 1. *The complexity of Algorithm 1 to monitor a k-safety* HyperLTL$_1$ *formula φ is*

$$O\left(\binom{n}{k} + \sum_{\phi \in \varphi} x^\phi\right)$$

where n is the number of finite executions and x^ϕ is the complexity of synthesizing a monitor for Ltl *sub-formula ϕ in φ.* ■

3.4.3 Monitoring beyond k-hypersafety

In the previous section, we showed the monitoring procedure for a k-safety (respectively, co-k-safety) hyperproperty such as the one given in the third row of Table 3.1 (respectively,

Table 3.2). Monitoring of other monitorable HyperLTL$_1$ formulas, such as the ones described in Table 3.3, is straightforward using Algorithm 1:

- Formulas that are conjunctions or disjunctions of a safety hyperproperty with a co-safety hyperproperty, should be first reduced to monitoring of the monitorable part of the formula. For example, the formula $\forall \pi_1 . \phi_1 \ \vee \ \exists \pi_2 . \phi_2$, where ϕ_2 is a co-safety property, can be reduced to monitoring of only $\exists \pi_2 . \phi_2$. This is because a formula with a $\forall$ quantifier can never be declared satisfied and due to conjunction it reduces to checking for satisfaction of ϕ_2 by some trace. Hence, Algorithm 1 will monitor the reduced part only. Similarly, for the second row of Table 3.3, since ϕ_1 is a safety property, its violation by a trace can be detected, which is sufficient to falsify the complete formula.
- Notice that, conjunctions or disjunctions of monitorable k-safety HyperLTL$_1$ formulas can be monitored by enabling the transition to the final state of the Petri net either when all of the input places contain a token or at least one input place contains a token, respectively. Examples of such formulas are in row 4 of Tables 3.1 and 3.2.

Algorithm 1: k-safety hyperproperty monitoring algorithm

Input: $\varphi = \forall\pi_1 \cdots \forall\pi_k.\, \phi_1 \vee \cdots \vee \phi_k, \alpha, \beta$
Output: $\lambda \in \{?, \bot\}$

$L := \{q_\bot\}$; $T := \{\}$; $\lambda :=?$; $\texttt{comps} := \{\}$;
forall $\phi \in \alpha$ **do**
  $\texttt{construct\_component}(\phi, \mathit{true})$;
forall $\mu \in \beta$ **do**
  $L := L \cup \{q^\mu\}$;
  $T := T \cup \{(q^\mu, \mathit{true}, q_\bot)\}$;
while *true* **do**
  $\texttt{get\_input}(e^i_j, \texttt{new\_trace})$;
  if $\texttt{new\_trace} = \mathit{true}$ **then**
    $\texttt{reset}\,(\texttt{comps})$;
    $\beta' := \beta$;
  forall $M^\phi \in \texttt{comps}$ **do**
    $q := \texttt{evaluate}(M^\phi, e^i_j)$;
    if $\lambda^\phi(q) = \bot$ **then**
      if $\exists \mu \in \beta \,.\, q = q^\mu$ **then**
        $\texttt{comps} := \texttt{comps} - \{M^\mu\}$;
      else
        $\texttt{comps} := \texttt{comps} - \{M^\phi\}$;
  forall $\mu' \in \beta'$ *and* $\mu \in \beta$ **do**
    $\mu' := \boldsymbol{Pg}(e^i_j)$;
    if $\texttt{progression\_complete}(\mu', \mu)$ **then**
      $\texttt{construct\_component}(\mu', \mu)$;
  if $({}^\bullet\tau, \texttt{true}, \{q_\bot\}^\bullet)$ *is enabled* **then**
    $\lambda := \bot$; **return** λ ;

Algorithm 2: Constructs components of the Petri net

```
1  construct_component(Formula φ, Formula μ) {
2  M^φ := (Σ, Q^φ, q_0^φ, Δ^φ, λ^φ) (see Definition 16);
3  L := L ∪ Q^φ;
4  T := T ∪ Δ^φ;
5  Let q ∈ Q^φ, where λ^φ(q) = ⊥ (only one such state is in Q [14]);
6  if μ ≠ true then
7      merge(q, q^μ);
8      T := T − {(q^μ, q^μ)};
9  else
10     T := T ∪ (q, true, q_⊥) ;
11     T := T − {(q, q)};
12 comps := comps ∪ {M^φ};
13 }
```

3.5 Implementation and Results

In this section, we evaluate Algorithm 1.

3.5.1 Experimental Settings

Parameters and data sets We use a dataset collected for a study at Microsoft Research [53] as well as sets of synthetically generated traces. The dataset involves GPS location data of 21 users taken over a period of eight weeks in the region of Seattle, USA. Besides the Microsoft Research dataset, we also use some synthetically generated datasets using Poisson, normal, and uniform distributions to ensure the robustness of our experiments. Each trace, in all of these datasets, corresponds to the continuous movement of a single user on a single day. The rationale behind using such traces is that a server might log locations of a user from the time a user opens an application until the time the user closes it. The traces are anonymized, that is, the trace itself does not reveal the identity of the user it belongs to. Each dataset consists of up to 200 finite traces with different lengths.

Security policies Another parameter we experiment with is three k-safety hyperproperties that specify the security, privacy, and anonymity of a user's GPS location data:

- *Anonymity* (***GMNI***) - If the initial location for a set of anonymized traces remains the same, all traces must eventually reach the final location reported by any trace:

$$\forall \pi_1. \forall \pi_2. ((\mathbf{G}\lambda_H(\pi_1) \ \wedge \ \mathbf{G}\lambda_H(\pi_2) \ \wedge \ \pi_1[0] \quad =_L \quad \pi_2[0]) \quad \Rightarrow \quad \pi_1 \quad \sim_{f,L} \quad \pi_2)$$

  where f maps x, the final location in a trace, to LTL formula $\mathbf{F}(x)$.

- *Privacy* (***OD***) - Assuming the traces are deanonymized, the locations visited by a user must be the same in all the traces of the same user.

$$\forall \pi_1. \forall \pi_2. (\pi_1 \sim_{f,L} \pi_2)$$

  where function f maps every location x to $\mathbf{F}(x)$.

- *Security* - Suppose that visiting a set of k locations, uniquely identifies some secure information about a user. Then, over all traces the user must not report having visited all of these k locations.

$$\forall \pi_1 \cdots \forall \pi_k. (\neg \mathbf{F} l_1 \ \wedge \ \cdots \ \wedge \ \neg \mathbf{F} l_k)$$

  where $\{l_i \mid 1 \leq i \leq k\}$ is the set of locations that reveals the secure information.

Metrics The metrics used for evaluation are (1) the total number of generated Petri net components, (2) the length of the progressed formulas, and (3) running time of the monitor. We note that there is a direct correlation between the number of components in the Petri net and the length of the progressed formula with the memory and time overhead to report a violation or satisfaction. With increasing number of components, the memory consumption increased and vice versa. Similarly as the length of progressed formulas grew bigger it increased the time to generate the components on-the-fly.

3.5.2 Results and Analysis

For each of the distributions, we generate 100 synthetic datasets for evaluation. The plotted values are the means of the results obtained from all the datasets, for each distribution. We plot the standard error within the figures itself.

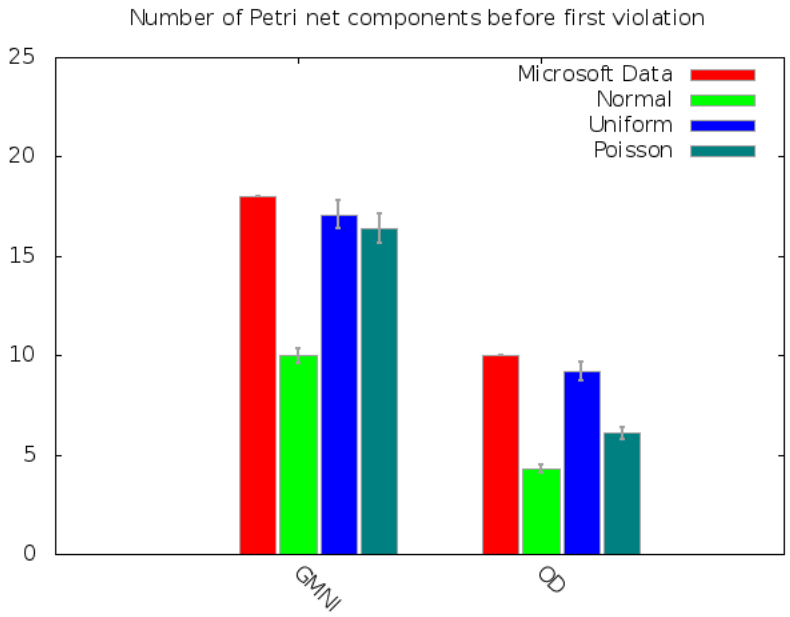


Figure 3.4: Number of Petri net components generated before detection of first violation

Number of Petri net components Fig. 3.4 shows that the number of components generated for the ***GMNI*** property before the first violation is detected is greater than that for property ***OD***. This is because ***GMNI*** is less strict than ***OD*** since the dependency is only on the first and last observed locations, whereas ***OD*** requires every location visited by a user in one trace to be visited in every other trace. This results in the property ***OD*** being violated in fewer observed traces. For the security property which consists of only observation-independent sub-formulas, the number of components remains a fixed number k for any number of traces and violations. The difference between the result reported for the normal distribution and the other datasets, for the ***GMNI*** property, is high possibly due to the reduced probability of seeing the last location of a trace in another trace. For the ***OD*** property, the probability of seeing all the locations of one trace in another trace is reduced further for both normal and Poisson distributions. Hence, we see that a detection is detected sooner resulting in fewer number of components being created.

To analyze the number of components that would be created if all the traces were evaluated by the monitor, we let the algorithm report all the violations instead of terminating at the first violation. We evaluated the violation of property ***GMNI***. It was seen that the number of violations reported were close to twice the number of components created—

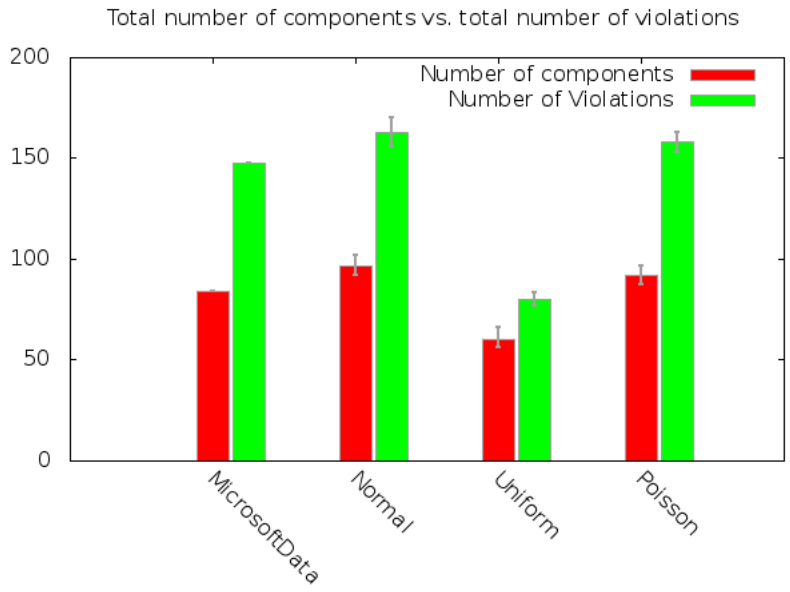


Figure 3.5: Total components vs. violations

which was less than 50% of the traces evaluated (see Fig. 3.5). Here we see that for the normal distribution the number of components created is much greater as compared to the uniform distribution as the probability of the same locations being visited among traces is higher for the uniform distribution due to which the number of unique progressed formulas are fewer. The total number of violations and components for evaluation of ***OD*** was significantly greater due to the property being more strict as explained earlier.

Table 3.4: Trace length of monitored formulas before first violation

GMNI	***OD***	security
8	238	12

Length of progressed formulas On comparing the length of formulas for each of the properties, ***OD*** formula is much longer as compared to both ***GMNI*** and security (see Table 3.4) since it captures every event observed by a trace. ***GMNI*** has a fixed, much smaller length since the formula is dependent only on the first and last observation. Thus, the length of the formula remains the same for all trace lengths. Observation-independent

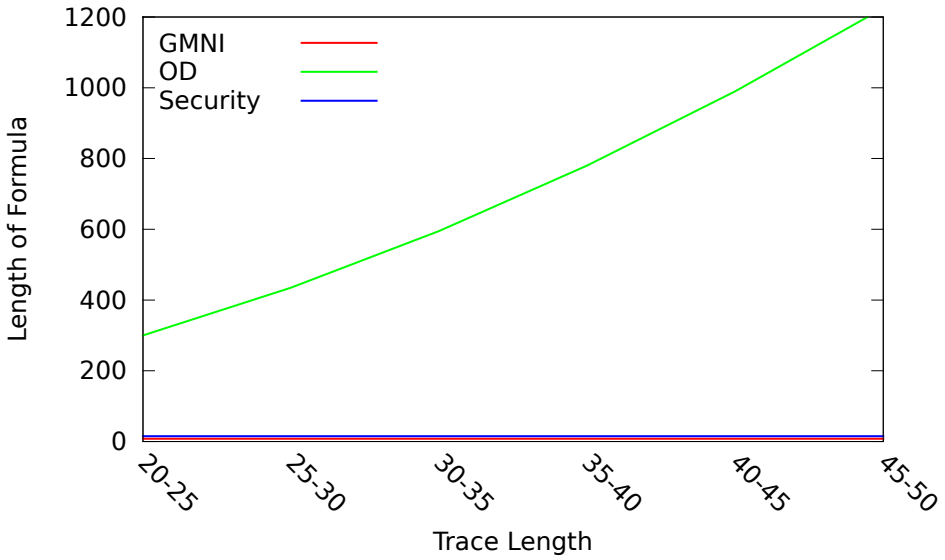


Figure 3.6: Length of formulas vs. trace length

sub-formulas result in a fixed length for security. We analyze the dependency of the length of the formula to be monitored on the length of traces for property ***OD***. As the length of each trace increases the length of the formula increases (see Fig. 3.6); i.e., a longer trace implies tracking of user location more frequently which results in an increasing length of the formula.

Chapter 4

The Complexity of Program Repair for Safety Hyperproperties

In this chapter, we show the complexity results for static enforcement of safety hyperproperties through program repair. First, we formally describe the program repair problem in Section 4.1. We present our general result on the complexity of program repair for k_ℓ-safety hyperproperties in Section 4.2. In Section 4.3, we analyze the complexity of the repair problem for 1-safety hyperproperties. Finally, we show results on polynomial-time solutions of a subset of safety hyperproperties in Section 4.4.

Our complexity analysis is based on the structure of $\boldsymbol{M_h}$ used to describe safety hyperproperties. Let $\boldsymbol{M_h} = \{M_0, M_1, \ldots\}$, where each M_i, $i \geq 0$, is a set of finite traces. Our first classification is based on the maximum size of each M_i whereas our second classification is based on the maximum length of sequences in set M_i. The first classification essentially describes k-safety hyperproperties.

For the second classification, we cannot directly limit the length of finite traces in each M_i, as it would only allow one to describe hypersafety violations in the few initial steps. Hence, when we limit the length of sequences, we consider the case where the corresponding sequence of states appears somewhere in the trace rather than necessarily at the beginning.

Specifically, we introduce the notion of $\leq_I$ between traces and sets of traces. In particular, given traces t and t', $t \leq_I t'$ means that t is a sub-trace of t'; i.e., $t \leq_I t'$ iff there exist traces u and v such that $t' = utv$. Likewise, for sets of traces T and T', $T \leq_I T'$ iff $\forall t \in T.\ \exists t' \in T'.\ t \leq_I t'$. Now, we can define k_ℓ-safety hyperproperty as follows:

Definition 18. *A hyperproperty* $\boldsymbol{S}$ *is a* k_ℓ-safety hyperproperty *(is* k_ℓ-safety*) iff*

$$\forall T \in \mathcal{P}(\Sigma^\omega).(T \notin \boldsymbol{S}) \implies \exists M \in \mathcal{P}^*(\Sigma^{\leq \ell}).(M \leq_I T) \wedge (|M| \leq k) \wedge (\forall T' \in \mathcal{P}(\Sigma^\omega).(M \leq_I T') \implies (T' \notin \boldsymbol{S})) \blacksquare$$

As an illustration, if the bad thing given by a hypersafety is described by the set $\{\{\langle a_0 b_0 c_0\rangle, \langle a_1 b_1\rangle\}, \{\langle a_2\rangle\}\}$, then any program that exhibits finite traces $\langle a_0 b_0 c_0\rangle$ and $\langle a_1 b_1\rangle$ or finite trace $\langle a_2\rangle$ violates the hypersafety. This requirement can be specified as a 2_3-safety. Finally, note that if $\ell = \infty$, then k_ℓ-safety is the same as k-safety. And, if k is ∞, then k-safety is the same as (generic) hypersafety.

4.1 Problem Statement

Given is a program $p = \langle I_p, \delta_p\rangle$ and a safety hyperproperty $\boldsymbol{S}$. We assume that the input program p has no deadlocked states (i.e., a state from where no outgoing transition is present). In case the input program is terminating and it terminates at a state, say s, then we add a self-loop (s, s) to s, so that a terminating state is not treated as a deadlocked state during repair.

In this thesis, the goal is to repair p so that the repaired program (denoted p') satisfies $\boldsymbol{S}$ while preserving its existing specification $\mathbb{E}$, where $\mathbb{E}$ is unknown; i.e., during the repair, we only want to reuse the correctness of p with respect to $\mathbb{E}$, so that $p \models \mathbb{E}$ automatically implies $p' \models \mathbb{E}$. We assume that $\mathbb{E}$ is a conjunction of a set of safety hyperproperties and trace properties.[1]

The formal statement of the program repair decision problem is as follows:

Repair decision problem: Given a program $p = \langle I_p, \delta_p\rangle$ and a k-safety hyperproperty $\boldsymbol{S}$, does there exist a program $p' = \langle I_{p'}, \delta_{p'}\rangle$ such that:

1. $I_{p'} = I_p$
2. $\delta_{p'} \subseteq \delta_p$
3. $\forall s_1.((\exists s_0.(s_0, s_1) \in \delta_{p'}) \vee (s_1 \in I_{p'})) \implies \exists s_2.(s_1, s_2) \in \delta_{p'}$
4. $p' \models \boldsymbol{S}$

[1] The existing results for instances where $\boldsymbol{S}$ and $\mathbb{E}$ are both trace properties are discussed in Section 5.2

The first constraint requires that no initial state is removed during repair and the last constraint obviously requires that the repaired program p' satisfies the input hypersafety $\boldsymbol{S}$. Constraints 2 and 3 ensure that satisfaction of the existing specification remains intact, as shown in Theorem 7.

Theorem 7. *Let $\mathbb{E}$ consist of a set of trace properties and safety hyperproperties. If for a program p, such that $p \models \mathbb{E}$, a repaired program p' exists that satisfies the repair decision problem, then $p' \models \mathbb{E}$.*

Proof. Observe that Constraint 3 ensures that the repaired program p' has no deadlocked states. This is due to the fact that from every initial state and every reachable state of p', there exists an outgoing transition. Moreover, due to Constraint 2, every transition of p' is a transition of p. Hence, any infinite trace of p' is an infinite trace of p. That is, $\psi(p') \subseteq \psi(p)$. This in turn guarantees that if $p \models \mathbb{E}$, then $p' \models \mathbb{E}$, since properties and safety hyperproperties are both closed under refinement [34]. ■

Finally, we argue that although our formulation of the repair problem is simple, where during repair traces that violate $\boldsymbol{S}$ are removed without adding new traces, it has been shown to be quite effective. For instance, Bonakdarpour et al. [20] show that a similar formulation can successfully syntheszie highly complex distributed fault-tolerant protocols such as Lamport's Byzantine agreement [55] and Dijkstra's distributed self-stabilizing token ring [37].

4.2 Repair for k-Safety

In this section, we show that, in general, the repair decision problem for k-safety is NP-hard in the size of the state space of the input program. This is achieved by showing that the repair decision problem for 2_1-safety hyperproperty is NP-hard. Since every k_ℓ-safety hyperproperty, for $k \geq 2$ and $\ell \geq 1$, can be reduced to a 2_1-safety hyperproperty using techniques like self-composition [8], NP-hardness result of the repair decision problem for k_ℓ-safety hyperproperties, follows from this result. We define the instance of adding 2_1-safety hyperproperty, 2SS, (based on Definition 18) as follows:

Instance 1 (2SS)**.** 2_1*-Safety Hyperproperty* $\boldsymbol{S_{2SS}}$*, where*

$$\forall T \in \mathcal{P}(\Sigma^\omega).(\exists M \in \boldsymbol{M_{2SS}}.(M \leq_I T) \iff (T \notin \boldsymbol{S_{2SS}}))$$

such that:

$$\boldsymbol{M_{2SS}} = \{\{\langle a_1\rangle, \langle b_1\rangle\}, \{\langle a_2\rangle, \langle b_2\rangle\}, \ldots, \{\langle a_n\rangle, \langle b_n\rangle\} \mid \forall i,\ 1 \leq i \leq n,\ a_i, b_i \in \Sigma\}$$

Theorem 8. *The program repair decision problem for instance 2SS is NP-hard in the state space of the input program.*

Proof. We focus on proving its NP-hardness by a reduction from the *Boolean satisfiability* (SAT) problem [44]:

> *Given is a set of propositional variables, $x_1, x_2, \ldots, x_n$, and a Boolean formula $y = y_1 \wedge y_2 \wedge \cdots \wedge y_M$, where each y_j $(1 \leq j \leq M)$ is a disjunction of one or more literals. Does there exist an assignment of truth values to $x_1, x_2, \ldots, x_n$ such that y is evaluated to true?*

Toward this end, we present a mapping from an arbitrary instance of the SAT problem to an instance of 2SS. Then, we show that the instance of SAT is satisfiable iff the answer to the decision problem for 2SS is affirmative. Given a SAT formula, we identify each entity, namely, program p and the set $\boldsymbol{M_{2SS}}$, of the instance 2SS of the repair decision problem.

Input program p. For each clause y_j in the instance of SAT, we introduce a state d_j $(1 \leq j \leq M)$. These are initial states of p. For each propositional variable x_i, we introduce two states a_i and b_i $(1 \leq i \leq n)$. We also introduce a special state s. Transitions of our instance of p are constructed as follows. First, we include the transition (s, s). Then, for each variable x_i, we include transitions (a_i, s) and (b_i, s). In addition, corresponding to each clause y_j, we include the following transitions:

- If x_i is a literal in y_j, then we include the transition (d_j, a_i).
- If $\neg x_i$ is a literal in y_j, then we include the transition (d_j, b_i).

As an example, if $y_j = x_1 \vee x_2 \vee \neg x_3$, the mapping would have the following transitions starting from d_j: (d_j, a_1), (d_j, a_2), and (d_j, b_3). It would also include the following transitions ending at s: (a_1, s), (b_1, s), (a_2, s), (b_2, s), (a_3, s), (b_3, s), and (s, s) (see Figure 4.1).

2_1-Safety Hyperproperty specification $\boldsymbol{S_{2SS}}$ for program p. Hypersafety will be violated if for some i, the program contains two traces that reach a_i and b_i. Thus, $\boldsymbol{M_{2SS}} = \{\{\langle a_1 \rangle, \langle b_1 \rangle\}, \{\langle a_2 \rangle, \langle b_2 \rangle\}, \ldots, \{\langle a_n \rangle, \langle b_n \rangle\}\}$.

We now show that the given SAT formula is satisfiable iff there exists a program p', a solution to the program repair decision problem defined in Section 4.1, for the instance 2SS. To this end, we distinguish two cases:

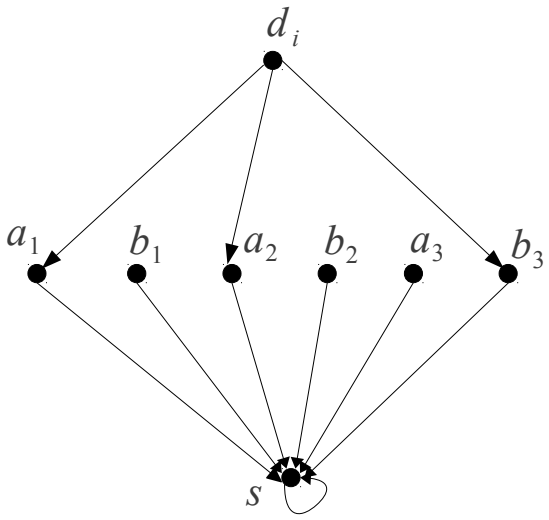


Figure 4.1: Instance 2SS

- ($\Rightarrow$) Since the SAT formula is satisfiable, there exists an assignment of truth values to the propositional variables x_i ($1 \leq i \leq n$), such that each y_j ($1 \leq j \leq M$) is true. Now, we obtain a program, p', that satisfies the constraints of the repair decision problem. The state space and initial states of p' are the same as that of p. The transitions of program p' are the following:
  - For each clause y_j that includes x_i, we include the transition (d_j, a_i) iff x_i is $true$ in the truth assignment used for the satisfaction of the SAT formula.
  - For each clause y_j that includes $\neg x_i$, we include the transition (d_j, b_i) iff x_i is $false$.
  - For each propositional variable x_i or $\neg x_i$ ($1 \leq i \leq n$), we include both transitions (a_i, s) and (b_i, s).
  - The transition (s, s) is also included.

  Now, we show that p' is the repaired program for the instance 2SS. Based on the construction of p', a_i is reachable only if x_i is true and b_i is reachable only if x_i is false. Hence, if p' has a trace that reaches a_i, then it does not have a trace that reaches b_i. Furthermore, from every state reached by p', there is a successor state in p', since p' contains both transitions (a_i, s) and (b_i, s) for each literal x_i in y_j and every state d_j

reaches some a_i or b_i because of satisfiability. Thus, p' does not deadlock. Hence, p' satisfies the constraints of the repair decision problem.

- ($\Leftarrow$) Based on the constraints of the repair decision problem, each state d_j ($1 \leq j \leq M$) in p' must have at least one successor state. If p' includes a transition of the form (d_j, a_i), we set x_i to true. If it includes a transition of the form (d_j, b_i), we set x_i to false. Since p' cannot simultaneously have two traces that reach both a_i and b_i, this truth assignment is consistent; i.e., each variable is assigned only one value. Note that, in the case when both a_i and b_i are unreachable any of the two truth values can be assigned to x_i. Furthermore, based on the construction of the input problem, the transition included from d_j ensures that clause y_j is satisfied; i.e., the given SAT formula is satisfiable.

■

Lemma 6. *The program repair decision problem for k_ℓ-safety hyperproperties, for $k \geq 2$ and $\ell \geq 1$, that can be expressed in* HyperLTL$_1$ *is in general NP-complete.*

Proof. Finkbeiner et al. [43] show a polynomial-time algorithm for model checking HyperLTL$_1$ (alternation-free) formulas. This shows membership in NP of the program repair decision for k_ℓ-safety hyperproperties that can be expressed in an alternation-free HyperLTL formula (Corollary 1). Since every k_ℓ-safety hyperproperty, for $k \geq 2$ and $\ell \geq 1$, can be reduced to a 2_1-safety hyperproperty using techniques like self-composition [8], the NP-hardness result of the repair decision problem for k_ℓ-safety hyperproperties follows from Theorem 8. Thus, the program repair decision problem for k_ℓ-safety hyperproperties, for $k \geq 2$ and $\ell \geq 1$, that can be expressed in HyperLTL$_1$ is in general NP-complete. ■

4.3 Repair for 1-Safety

This section presents our results on repair for 1_ℓ-safety hyperproperty.

Instance 2 (1S2PP)**.** 1_ℓ*-Safety Hyperproperty* $\boldsymbol{S_{1S2PP}}$ *(i.e., Definition 18), where*

$$\forall T \in \mathcal{P}(\Sigma^\omega).(\exists M \in \boldsymbol{M_{1S2PP}}.(M \leq_I T) \iff (T \notin \boldsymbol{S_{1S2PP}}))$$

where $\boldsymbol{M_{1S2PP}} \in \mathcal{P}(\Sigma^{\leq \ell})$.

Here, we show that for the program repair decision problem is NP-hard for instance 1S2PP. Our reduction is again from the SAT problem to an instance of 1S2PP. We start with the mapping.

Input program p. For each clause y_j in the instance of SAT, we introduce a state d_j ($1 \leq j \leq M$). These are initial states of p. Corresponding to each propositional variable x_i, we introduce eight states $P_i, Q_i^1, Q_i^2, R_i, S_i^1, S_i^2, a_i$, and b_i ($1 \leq i \leq n$). Transitions of our instance of p are constructed as follows (see Figure 4.2). For each variable x_i, we include transitions $(P_i, a_i), (R_i, b_i), (a_i, S_i^1), (S_i^1, b_i), (b_i, Q_i^1), (Q_i^1, a_i), (a_i, Q_i^2), (b_i, S_i^2), (Q_i^2, Q_i^2)$, and (S_i^2, S_i^2) in the set of program transitions δ_p. Moreover, for each clause y_j, we include the following transitions:

- If x_i is a literal in y_j, then we include the transition (d_j, P_i).
- If $\neg x_i$ is a literal in y_j, then we include the transition (d_j, R_i).

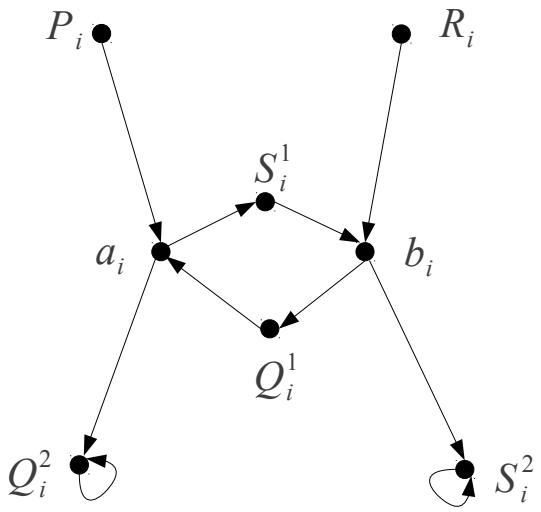


Figure 4.2: Instance 1S2PP

1-Safety Hyperproperty specification S_{1S2PP} for program p. Hypersafety will be violated if any trace passes through the sequence of three states specified in M_{1S2PP}, where

$$M_{1S2PP} = \{\{P_i a_i Q_i^2\}, \{R_i b_i S_i^2\}, \{S_i^1 b_i Q_i^1\}, \{Q_i^1 a_i S_i^1\} \mid 1 \leq i \leq n\}$$

We now show that the given instance of SAT is satisfiable iff the program repair decision problem from Section 4.1 can be solved for instance 1S2PP.

Lemma 7. *If the given SAT formula is satisfiable, then there exists a program p' that satisfies the repair decision problem from Section 4.1 for the instance 1S2PP.*

Proof. Since the SAT formula is satisfiable, there exists an assignment of truth values to the propositional variables x_i $(1 \leq i \leq n)$ such that each y_j $(1 \leq j \leq M)$ is *true*. Now, we obtain a repaired program, p', for the repair decision problem for instance 1S2PP. The state space and initial states of p' are identical to those of p. We derive the transitions of the repaired program p' as follows:

- For each variable x_i, if x_i is *true*, then we include the transitions $(P_i, a_i), (a_i, S_i^1)$, $(S_i^1, b_i), (b_i, S_i^2)$, and (S_i^2, S_i^2).
- For each variable x_i, if x_i is *false*, then we include the transitions $(R_i, b_i), (b_i, Q_i^1)$, $(Q_i^1, a_i), (a_i, Q_i^2)$, and (Q_i^2, Q_i^2).
- For each clause y_j that contains x_i, we include the transition (d_j, P_i) if x_i is *true*.
- For each clause y_j that contains $\neg x_i$, we include the transition (d_j, R_i) if x_i is *false*.

Next, we show that p satisfies the constraints of the repair decision problem defined in Section 4.1. The first two constraints are satisfied by construction. The third condition, deadlock freedom, is satisfied since a trace that reaches P_i has transitions $(P_i, a_i), (a_i, S_i^1), (S_i^1, b_i), (b_i, S_i^2)$, and (S_i^2, S_i^2). Likewise, if the trace reaches R_i, it includes transitions $(R_i, b_i), (b_i, Q_i^1), (Q_i^1, a_i), (a_i, Q_i^2)$ and (Q_i^2, Q_i^2). Finally, the fourth constraint is also satisfied by construction. ■

As an illustration, we show the partial structure of p', for the formula $[(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)]$, where $x_1 = true, x_2 = false, x_3 = false$, and $x_4 = false$ in Figure 4.3. Now, we show that p' meets the requirements of the decision problem.

- The first two constraints of the program are trivially satisfied.
- To see that no computation reaches a deadlock state, we look at a computation for disjunction y_j. Now, let $y_j = x_i \vee \neg x_k \vee x_r$ be a disjunction in the SAT formula. Since y_j evaluates to true, p' includes a transition from $\{(d_j, P_i), (d_j, R_k), (d_j, P_r)\}$. A computation reaching a state in P (respectively, R) eventually reaches a state in T (respectively, Q) which has a self loop. Hence, any trace in p', starting from one of the initial states, does not deadlock.

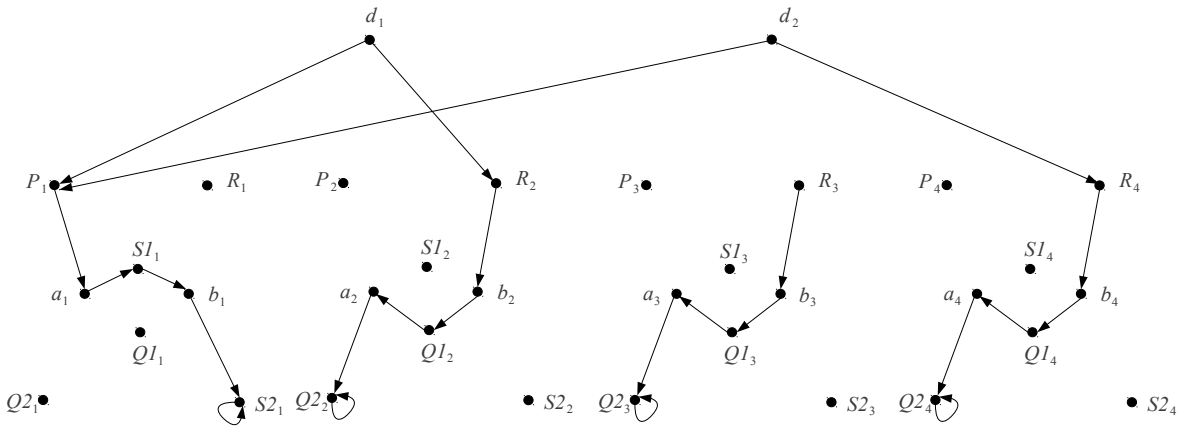


Figure 4.3: The partial structure of the revised program for instance 1S2PP.

- As we can see from the example, no computation reaching a state in P (respectively, R) reaches a state in Q (respectively, S).

Lemma 8. *If there exists a repaired program p' that satisfies the repair decision problem defined in Section 4.1, then the given SAT formula is satisfiable.*

Proof. To ensure deadlock freedom, for each d_j $(1 \leq j \leq M)$, an initial state, there exists an i $(1 \leq i \leq n)$ such that either P_i or R_i must be reachable. Hence, we have:

Observation 2. *For each j $(1 \leq j \leq m)$ there exists i $(1 \leq i \leq n)$, such that either $(d_j, P_i) \in \delta_{p'}$ or $(d_j, R_i) \in \delta_{p'}$.*

Next, we observe that if P_i is reachable starting from some initial state, then R_i is not reachable from any initial state and vice versa. That is,

Observation 3. *If $(d_j, P_i) \in \delta_{p'}$ then $\forall j'.(d_{j'}, R_i) \notin \delta_{p'}$ and vice versa.*

Let us assume by contradiction that, in the repaired program p', both P_i and R_i are reachable from any initial states. To ensure deadlock freedom, both the transitions (P_i, a_i) and (R_i, b_i) must exist in p'. Since p' satisfies the safety hyperproperty $\boldsymbol{S_{1S2PP}}$ neither (a_i, Q_i^2) nor (b_i, S_i^2) can be transitions in p'. Further, since (a_i, S_i^1) and (b_i, Q_i^1) are the only outgoing transitions from a_i and b_i, respectively, they have to be included in p'. Similarly, (S_i^1, b_i) and (Q_i^1, a_i) must be included as transitions in p'. This allows the bad finite traces

$\{S_i^1 b_i Q_i^1\}$ and $\{Q_i^1 a_i S_i^1\}$ to exist in the program, which contradicts the assumption that p' is a repaired program for the given instance. Hence, only one of the states P_i and R_i is reachable.

Now, from d_j, if p' contains a transition to P_i (respectively, R_i), then y_j contains x_i (respectively, $\neg x_i$) and x_i is assigned the truth value *true* (respectively, *false*). Hence, $\forall j.y_j$ evaluates to true. Thus, the SAT formula evaluates to true. Note that, if neither P_i nor R_i is reachable by any d_j, then either of the two truth values can be assigned to x_i.

■

Theorem 9. *The program repair decision problem for 1_ℓ-safety hyperproperty for instance $1S2PP$ is NP-hard in the state space of the input program if $\ell \geq 3$.*

Proof. The NP-hardness of the program repair decision problem for instance 1S2PP follows from Lemmas 7 and 8. ■

Lemma 9. *The program repair decision problem for 1_ℓ-safety hyperproperties, for $\ell \geq 3$, that can be expressed in* HyperLTL$_1$ *is in general NP-complete.*

Proof. Similar to the proof of Lemma 6, we can see membership of the problem in NP. From Theorem 9. it follows that the program repair decision problem for 1_ℓ-safety hyperproperties, for $\ell \geq 3$, that can be expressed in HyperLTL$_1$ is in general NP-complete. ■

Finally, we note that the repair decision problem can be solved in polynomial time for 1_2-safety as shown by Bonakdarpour et al. [17]. This is due to the fact that for a 1_2-safety, the bad things are specified in terms of transitions that program p' should not have. Therefore, there exists an appropriate p' if and only if there exists an appropriate p' with none of the forbidden transitions reachable from any of the initial states. Hence, p' can be designed by first removing those transitions from p and then removing any deadlocks that may result from it. A deadlock state s that is not an initial state can be removed by removing all transitions of the form (s_0, s), where $s_0 \in \Sigma$. While this may create additional deadlocks, they can be removed recursively. We return a 'no' to the decision problem if and only if there is a deadlocked initial state otherwise we return 'yes'. This algorithm is sound and complete in that it returns a 'no' if and only if there does not exist any repaired program p' for the given program p and safety hyperproperty. This is because we only remove two kinds of transitions—(1) those that are forbidden or (2) recursively those that reach a deadlocked state. Thus, after the removal of the forbidden transitions, the algorithm does not remove a transition that was part of any terminating trace since such

a trace does not have any deadlocked states. Hence, if the algorithm returns a 'no' then there cannot exist a repaired program. It is trivially sound because if there does not exist a repaired program then a deadlocked initial state will be reached and the algorithm will return 'no'. Thus, we have:

Theorem 10. *The program repair decision problem for* 1_2*-safety hyperproperties can be solved in polynomial time.*

Observe that for 1_2-safety hyperproperty, the set of bad things is of the form $\{\{\langle a_1 b_1\rangle\}, \{\langle a_2 b_2\rangle\}, \cdots\}$. From the proof of Theorem 9, we can observe that the problem remains NP-hard even if $\boldsymbol{M_{1S2PP}}$ is of the form $\{\{\langle a_1 u_1 b_1\rangle\}, \{\langle a_2 u_2 b_2\rangle\}, \cdots \mid \forall i. u_i \in \Sigma^*\}$. Thus, we have:

Instance 3 (1SnBP)**.** *Elongated* 1_2*-safety hyperproperty* $\boldsymbol{S_{1SnBP}}$ *(i.e., Definition 6), where*

$$\forall T \in \mathcal{P}(\Sigma^\omega).(\exists M \in \boldsymbol{M_{EP}}.(M \leq_I T) \iff (T \notin \boldsymbol{S_{1SnBP}}))$$

where $\boldsymbol{M_{EP}} = \{\{\langle a_1 u_1 b_1\rangle\}, \{\langle a_2 u_2 b_2\rangle\}, \cdots \mid a_i, b_i \in \Sigma, u_i \in \Sigma^*\}$.

Theorem 11. *The program repair decision problem for instance 1SnBP is NP-hard in the size of the input program.*

While Theorem 11 states that the repair decision problem for 1-safety hyperproperties is NP-hard, we will show in Section 4.4 that if the bad things are specified by one pair of predicates, then the repair decision problem can be solved in polynomial time (Theorem 13). This result is also tight in that if elongated pairs are generated from two pairs of predicates (i.e., the 1-hypersafety is of the form $(P \Rightarrow \Box\neg Q) \wedge (R \Rightarrow \Box\neg S)$ in Figure 4.2), then the problem is NP-hard. This proof is similar to that of Theorem 9.

4.4 Polynomial-time Repair for Safety Hyperproperties

In this section, we identify classes of hypersafety properties for which repair can be performed in polynomial time. Our first class of problems is motivated by Theorem 8, where we showed that the repair decision problem is NP-hard for k_ℓ-safety hyperproperties. Specifically, in Section 4.4.1, we identify a subset of k_1-safety hyperproperties for which the repair decision problem can be solved in P.

Our second class of problems is motivated by Theorem 11, where we showed that the repair decision problem is NP-hard for the case where bad things are a set of elongated pairs. Specifically, in section 4.4.2, we show that this problem can be solved in P if the set of bad pairs is generated from a pair of predicates. However, if the elongated pairs are created from two pairs of predicates then the problem is NP-hard.

4.4.1 Polynomial-time Repair for a Class of k_1-Safety Hyperproperty

To illustrate this subset of safety hyperproperties, we begin with the secret sharing scheme discussed in Figure 2.1. Consider a simple instance of this scheme where the secret is split into two parts and both parts are needed for revealing it. In this case, the security requirement is that there cannot be two traces of the program such that one reveals the first part of the secret and another reveals the second part. In other words, there exist two subsets of sets S_1 and S_2, such that if there exists a trace that reaches S_1 (first part of the secret is revealed) and a trace that reaches S_2 (second part of the secret is revealed), then the security requirement is violated. If the program reaches S_1 alone (respectively, S_2 alone), then it does not violate the security requirement.

With this intuition, we define a subset of k_1-safety hyperproperties for which program repair can be achieved in polynomial time in the state space of the input program. Recall that in a k_1-safety hyperproperty, the set of bad things is described by the set $M = \{M_1, M_2, M_3, \ldots\}$, where each $M_i, i \geq 1$, is a set of sequences of length 1 and $|M_i| \leq k$. In other words, for $k = 2$, M is of the form $\{\{\langle a_1\rangle, \langle b_1\rangle\}, \{\langle a_2\rangle, \langle b_2\rangle\}, \ldots\}$. A given program violates this hyperproperty iff for some i, it exhibits a trace that reaches a_i and a trace that reaches b_i.

The sub-class of k_1-safety hyperproperties considered in this section, called *k-generated* safety hyperproperty, corresponds to the case where M can be characterized by k different state predicates, $S_1, S_2, \ldots, S_k$; i.e.,

$$M = \{\{\langle a_1\rangle, \langle a_2\rangle, \cdots, \langle a_k\rangle\} \mid (a_i \in S_i) \ \wedge \ (1 \leq i \leq k)\}.$$

Notice that a 2_1-safety hyperproperty does not correspond to some 2-generated safety hyperproperty. For example, consider a 2-safety hyperproperty where M is $\{\{\langle a_1\rangle, \langle b_1\rangle\}, \{\langle a_2\rangle, \langle b_2\rangle\}\}$. In this case, we cannot use two sets to generate all pairs in M without generating extra pairs that do not belong. Observe that, a generalized secret sharing scheme where the number of shares is k and all must be revealed to identify the secret

Algorithm 3: Algorithm for Program Repair Decision Problem for k-generated Safety Hyperproperties

Input: A program $p = \langle I_p, \delta_p \rangle$ and a hypersafety characterized by M_{kgen}.
Output: Result of the program repair decision problem.

```
1  R ← ReachableStates(I_p);
2  T ← Transitions(p);
3  foreach S_i in M_kgen do
4      R_i ← R − S_i;
5      T_i ← T
6      foreach s in S_i do
7          T_i ← T_i − In(s) − Out(s);
8      if detectCycle(I_p, R_i, T_i) then
9          print "Exists!"
10         return 1;
11
12 print "Does Not Exist!"
13 return 0;
```

can be expressed as a k-generated safety hyperproperty as well. Now, we show that repair for k-generated safety hyperproperty can be achieved in polynomial time. Algorithm 3 shows that since a k-generated safety hyperproperty is characterized by state predicates $M_{kgen} = \{S_1, S_2, \ldots, S_k\}$, we can repair the input program by considering k instances. Each instance corresponds to the case where we try to ensure that states in S_i are not reached although states in $(\bigcup_{j=1, j\neq i}^{k} S_j) - S_i$ may be reached. Solution obtained in any step gives the repaired program. If no solution is found, then the input program cannot be repaired.

Theorem 12. *Algorithm 3 for program repair decision problem for k-generated safety hyperproperties is sound and complete.*

Proof. An infinite trace is detected only for programs, where at least one of the state predicates in M_{kgen} is not reachable. Thus, *soundness* is trivially proved. The search for an infinite trace is performed for each case; S_i $(1 \leq i \leq k)$ is not reachable. In each case, only the transitions reaching or leaving the unreachable states in the state predicate are removed. Therefore, the algorithm is *complete* and if the algorithm returns 0, then there

does not exist a repaired program that satisfies the condition of the problem statement. ■

The running time of the algorithm for the program repair decision problem for k-generated safety hyperproperties is $O(k \cdot n^2)$ since the running time of the algorithm for detecting an infinite trace is $O(n^2)$ which is run for each of the k predicates.

Algorithm 4: Detecting infinite trace

Input: A set of initial states I_p, a set of reachable states R' and set of transitions, T', between these states.

Output: Detects the existence of an infinite computation.

```
1  detectCycle(I_p, R', T')
2  forall elements s_i of I_p do
3      forall elements v of R' do
4          mark(v) = 0;
5          visited(v) = 0;
6      DFS(s_i)
7  DFS(v)
8  mark(v) = 1;
9  visited(v) = 1;
10 forall w such that (v, w) ∈ T' do
11     if visited(w) == 0 then
12         DFS(w);
13     else if mark(w) == 1 then
14         print "Exists!" return true;
15
16 mark(v) = 0;
17 print "Does Not Exist!" return false;
```

4.4.2 Polynomial-time Repair for a Class of 1_ℓ Safety Hyperproperty

In Section 4.3, we showed that if the set of bad things are specified as elongated pairs, the repair decision problem is NP-hard. In this section, we show that if the elongated pairs are generated from two state predicates, say S_1 and S_2, then the problem can be solved in

polynomial time in the state space of the input program. We introduce the corresponding instance as follows:

Instance 4 (1S1BP)**.** *1-Safety hyperproperty* $\boldsymbol{S_{1S1BP}}$ *(i.e., Definition 6), where*

$$\forall T \in \mathcal{P}(\Sigma^{\omega}) : \forall M \in \boldsymbol{M_h} : (M \leq_I T) \implies (T \notin \boldsymbol{S_{1S1BP}})$$

such that $\boldsymbol{M_h} = \{\{a\Sigma^* b\}, \{b\Sigma^* a\} \mid S_1, S_2 \in \mathcal{P}(\Sigma) \ \wedge \ (a \in S_1) \ \wedge \ (b \in S_2)\}$.

Theorem 13. *The program repair decision problem for instance 1S1BP can be solved in polynomial time with* $O(n^2)$ *time complexity in the state space of the input program.*

Proof. By observation, we can see that instance 1S1BP has conditions similar to 2-generated hypersafety except that now the state predicates S_1 and S_2 are defined for one trace. Hence, by making use of Algorithm 3 we see that the repair decision problem for instance 1S1BP can be solved in $O(n^2)$.

■

Running Example In the example described in Fig. 2.1, a repaired program does exist and can be obtained by making at least 1 of the n states that reveal a share of the secret unreachable. One of the solutions obtained is the program in Fig. 4.4.

```
int main( ){
    x_1 = 1, x_2 = 2, ..., x_n = n;
    while(1) {
        ~~current = (rand()%n) + 1~~;
        current = (rand()%(n − 1)) + 1;
        switch(current) {
            case x_1 : output = Input(P_1);
                break;
            case x_2 : output = Input(P_2);
                break;
                ⋮
            case x_n : output = Input(P_n); }
        print output; }
    }
```

Figure 4.4: Secret sharing repaired program

Chapter 5

Related Work

In this chapter, we will discuss the current state-of-the-art work and techniques used in the past for runtime verification and program repair, and more specifically for security policies. We illustrate how our contributions in this thesis improve upon or are different from any other related work.

5.1 Runtime Verification

Runtime verification is being pursued as a lightweight mechanism to complement techniques such as model checking, theorem proving and testing. Unlike enforcement, it only deals with the detection of violations (or satisfactions) of the given property by the program under scrutiny. Here, we report on work for runtime verification for temporal logics and security policies.

5.1.1 Runtime Verification for Linear Temporal Logics

To improve upon the limitations of the two-valued semantics for LTL on finite trace, Bauer et al. [12] proposed the three-valued semantics and used it for runtime verification. Here, they described how to construct a deterministic finite-state machine with three output symbols. Bauer et al. further extended this work to four-valued semantics which resolves the inconclusive verdict of 'unknown' by answering either presumably true or presumably false. Pnueli et al. [69] gave the definition of monitorable properties and Bauer et al. [14]

showed that the class of monitorable properties is richer than the union of safety and liveness properties.

Formula rewriting based approach was used by Havelund et al. [48], where the future time LTL formula is transformed into a formula expressing what needs to be satisfied by the current observation and a formula that needs to be checked for the remaining execution. The work of Bauer et al. [15] considers the problem of monitoring an LTL formula in a synchronous distributed setting lacking in any centralised decision making authority. The approach used here is again one of formula rewriting (progression).

5.1.2 Verification of Security Policies

Offline verification

Basin et al. [11] develop a model checker for security protocols. Since traditional tools and verification methodologies are not equipped to deal with sets of traces, several results introduce new logics or operators to express hyperproperties. SecLTL extends LTL by using an additional *hide* modality [39]. It allows expression of non-interference as well as the instance until a high level data should remain independent of interference from low level data. The modal μ-calculus does not suffice to express some information flow properties. Epistemic logic has been used to implicitly quantify over traces [41]. However, HyperLTL and HyperCTL* [32] subsume epistemic logic and quantified propositional temporal logic [79].

Static analysis

Sabelfeld et al. [73] survey the literature focusing on static program analysis for enforcement of security policies. In some cases, with compilers using Just-in-time compilation techniques and dynamic inclusion of code at run time in web browsers, static analysis does not guarantee secure execution at run time. Type systems, frameworks for JavaScript [29] and ML [70] are some approaches to monitor information flow. Several tools [40,63,64] add extensions like statically checked information flow annotations to Java language. Clark et al. [30] present verification of information flow for deterministic interactive programs. Our approach, on the other hand, is capable of monitoring a rich subset of k-safety hyperproperties and not just information flow without assistance from static analyzers.

Dynamic analysis

Russo et al. [72] concentrate on permissive techniques for the enforcement of information flow under flow-sensitivity. It has been shown that in the flow-insensitive case, a sound purely dynamic monitor is more permissive than static analysis. However, they show the impossibility of such a monitor in the flow-sensitive case. A framework for inlining dynamic information flow monitors has been presented by Magazinius et al. [59]. The approach by Chudnov et al. [28] uses hybrid analysis instead and argue that due to JIT compilation processes, it is no longer possible to mediate every data and control flow event of the native code. They leverage the results of Russo et al. [72] by inlining the security monitors. Chudnov et al. [27] again use hybrid analysis of 2-safety hyperproperties in relational logic. They check for violation on observing a single run that they call a 'major' trace, which is monitored with alternate 'minor' traces. Hybrid analysis uses the goodness of static analysis and combines it with dynamic analysis. However, dynamic languages like JavaScript make such approaches impractical. Austin et al. [5] implement a purely dynamic monitor, however, restrictions such as "no-sensitive upgrade" were placed. Some techniques deploy taint tracking and labelling of data variables dynamically [65, 83]. Zdancewic et al. [82] verify information flow for concurrent programs. Decker et al. [36] provide verification techniques for first-order theories for reasoning about data that can be applied to check for secure execution of multi-threaded, object oriented systems. Most of the techniques cited above, aim to monitor security policies that are 2-safety hyperproperties, on observing a single run, whereas, our work is for any k-safety hyperproperty, when multiple runs are observed.

5.2 Program Repair

Automated model repair is a relatively new area of research. To the best of our knowledge, this paper is the first work on applying model repair in the context of security policies specified by hyperproperties. Since automated program repair is a nontrivial extension of the model checking problem, we report work on the same as well.

5.2.1 Repair for Temporal Logics

Model repair with respect to CTL properties was first considered by Buccafurri et al. [22] for concurrent programs and protocols. The authors integrate model checking with program repair to develop an algorithm with high computational cost that uses AI techniques of

abductive reasoning and theory revision. Chatzieleftheriou et al. [26] studied model repair for CTL using abstraction techniques to deal with the state explosion problem, . The theory of model repair for memoryless LTL properties was considered by Jobstmann et al. [51] in a game-theoretic fashion. The repair problem is modelled as a Buchi game that is the product of a modified version of the program and an automaton for the LTL specification. Repair is achieved by finding the winning strategy. Memoryless strategy is used to avoid adding new variables to the program. The authors show that deciding whether such a memoryless strategy exists is NP-complete, and present a heuristic to find an efficient repair for a given memoryless strategy.

Bonakdarpour et al. [17] use a graph-theoretic perspective. They show that repair with respect to multiple safety properties and one liveness property can be performed in polynomial time. However, the repair problem for two liveness properties becomes NP-complete. In terms of our work, they show that repair for 1_2-safety lies in complexity class P and reported that repairing with respect to 1_ℓ-safety, for $\ell > 2$, remained an open problem.

5.2.2 Repair for Various Systems

Bonakdarpour et al. [18] show that repairing programs, where distributed processes can only partially observe the global state of the program, is NP-complete for even one safety or one liveness property. For probabilistic systems such as discrete-time Markov chains, and probabilistic temporal logics, Bartocci et al. [9] use modified parametric probabilistic model checking to reduce the model repair problem to a non-linear optimization problem with a minimal-cost objective function. Samanta et al. [75] consider program repair for Boolean programs that could be used to model some Boolean circuits.

5.2.3 Checking of Safety Hyperproperties

Yasouka et al. [81] give results on the hardness of checking policies that can be specified as safety hyperproperties, in loop-free Boolean programs. They show that the problem for 2-safety hyperproperties such as non-interference is coNP-complete. For a safety hyperproperty that is not a k-safety hyperproperty, on quantifying the security of the program, determining whether this quantity is less than a constant is PP-hard. Since model repair is generally a harder problem than verification, our results on model repair for k-safety hyperproperties give us insight into complexity results of algorithms that try to revise the programs to satisfy the hyperproperty. Barthe et al. [8] show that k-safety can be reduced

to a safety property through self-composition. Even though the repair problem is in polynomial time if self-composition is used, it is at the cost of replicating the state space. The result of this thesis shows the complexity of repair, if expansion of the state space is not allowed.

5.2.4 Synthesis and Repair for Security Policies

Synthesizing security protocols from BAN logic [23] specifications has been studied by Saidi et al. [74]. The authors describe a synthesis tool with protocol goals specified in this logic, and when combined with a proof system it can be used to generate protocols satisfying those goals. Unlike our work that repairs an existing protocol, the technique given here synthesizes a protocol from scratch and, hence, cannot reuse the previous efforts made in designing an existing protocol. The work of Shmatikov et al. [78] uses the finite state tool Murϕ to model the participants in a protocol together with an intruder model, to check a set of safety properties by state space exploration. Chatterjee et al. [25] study the automatic synthesis of fair non-repudiation protocols. The objectives of the participants, the trusted third party and the protocol is formalized as path formulas in LTL and satisfaction of the objectives guarantees satisfaction of the protocol. The paper demonstrates the effectiveness of assume-guarantee synthesis in synthesizing fair exchange protocols. The approach proposed by Martinelli et al. [60] gives an automated framework for synthesis of controller programs for enforcing security policies.

In the context of repairing security protocols, Pimentel et al. have proposed applying formulation of protocol patch methods to repair security protocols automatically [66, 67]. In order to guide the location of the fault in a protocol, they use Abadi and Needham's principles [1] for the prudent engineering practice for cryptographic protocols. However, by its nature, the work by Pimentel et al. applies to protocols where principles given by the authors are not followed. Armando et al. [4] propose a general model for security protocols based on a set-rewriting formalism that coupled with the use of LTL allows for the specification of assumptions on principals and communication channels along with complex security properties. Corin et al. [35] propose a linear-time temporal logic with past operators for the specification of security protocols and their properties. The model checking procedure that they provide determines whether the given property holds on any given symbolic execution trace.

5.3 Runtime Enforcement of Security Policies

Schneider [76] started work on understanding which security policies are enforceable. The author considers the Execution Monitoring enforcement mechanism that monitors each execution step and terminates the execution if the next action would result in a violation. One of the important findings of the paper is that only those security policies that are safety properties are enforceable using this mechanism. Other mechanisms could include raising exceptions or inserting and deleting actions into the system to enforce a policy as was done by Ligatti et al. [56,57] using the edit automata. Ligatti et al. [58] introduced the mandatory-edit automata that interacts with the system and receives requests from the system. Basin et al. [10] distinguish between actions that are only observable and those that are controllable by an enforcement mechanism. They give necessary and sufficiency conditions for policy enforcement based on execution monitoring. This further allows them to reason about security policies involving timing constraints. Falcone et al. [42] study the problem of enforcement for *Safety-progress* hierarchy of regular properties. They introduce an enforcement monitor based on finite sets of control states, a memory device and enforcement operations on the input events. However, all these works on enforcement are for trace-based properties and not hyperproperties.

Chapter 6

Conclusion

Several complex systems allow easy inference about secure and private data on observing multiple executions of a system. To detect such violations and enforce additional policies on existing systems, in this thesis, we focused on two aspects of ensuring a system satisfies a rich class of security policies, namely 1) runtime verification and 2) static enforcement through program repair. Our specification language is a subset of hyperproperties — safety hyperproperties—which allows expressing policies that are not trace-based (e.g., information flow).

6.1 Summary

6.1.1 Runtime Verification

For the runtime verification of safety hyperproperties, first, we showed that the runtime verification of any safety hyperproperty with unbounded number of bad traces is in general undecidable. Following this, we showed that if we restricted the specifications to only k-(co)-safety hyperproperties, the problem is NP-hard. We also showed a sufficiency condition for a safety hyperproperty to be monitorable in polynomial time.

Moreover, we concentrated on obtaining a monitoring algorithm for a class of hyperproperties —k-(co)-safety hyperproperties with a bounded number of bad traces. This class allows specification of important security policies such as *information flow*.

In order to have syntactic means, we characterized k-(co)-safety hyperproperties in HyperLTL, a temporal logic that allows explicit quantification over execution traces.

Next, we introduced HyperLTL-3, a generalization of LTL_3 [14] (a logic designed for runtime verification of Ltl) to the context of HyperLTL. Different classes of monitorable HyperLTL formulas, by syntactic means, were identified and in particular it was observed that every k-(co)-safety hyperproperty is monitorable. Finally, we introduced a runtime verification algorithm for monitoring k-safety/co-k-safety hyperproperties and studied its performance with respect to different metrics.

6.1.2 Program Repair

Focusing on the problem of repairing a given program with respect to security policies, we characterized safety hyperproperties in terms of a set $\boldsymbol{M_h}$ of 'bad' sets of finite sequences. Two factors were considered in identifying the expressiveness of M: (1) the size of sets in $\boldsymbol{M_h}$ (k-safety hyperproperties), and (2) the maximum length of finite sequences in the sets in M, hence, k_ℓ-safety hyperproperties. We made the following contributions for program repair with respect to safety hyperproperties;

- We showed that in general the repair decision problem for k_ℓ-safety hyperproperties is NP-hard, although it can be solved in polynomial time when $k = 1$ and $\ell = 2$.
- Additionally, we showed that this bound is tight in that the problem is NP-hard if ($k = 2$) or if ($k = 1$ and $\ell = 3$). This result implies that repair for invariance properties (e.g., $p \Rightarrow \Box q$) can be achieved in polynomial time. However, repair for most interesting security policies such as information flow is still NP-hard.
- We also proved that the repair decision problem can be solved in polynomial time for the class of k-generated safety hyperproperties, where the set M can be generated from k state predicates (e.g., in secret sharing). The significance of this result is that one can identify a stronger safety hyperproperty that is k-generated and repair with respect to that hyperproperty. The resulting program, thus, is guaranteed to satisfy the desired safety hyperproperty.

6.2 Future Work

Since the current work only concerns safety hyperproperties, it can be extended further to include runtime verification and static enforcement through program repair for hyper-liveness and probabilistic hyperproperties. Furthermore, for program repair and runtime verification, our work can be extended as given in the following sections.

6.2.1 Runtime Verification

There are numerous interesting research avenues to extend this work. Among these, we are currently generalizing our work on distributed monitoring of LTL properties [61] to monitor hyperproperties in a distributed setting. Another interesting research direction would be to look at how one can monitor hyperproperties by analyzing execution as well as an abstract model of the system at run time. This idea is especially beneficial for monitoring hyperliveness properties.

6.2.2 Program Repair

An interesting line of work would be to devise repair algorithms for security policies in multi-agent distributed systems. Designing a technique for runtime enforcement of hyperproperties also constitutes another open problem.

References

[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 1996.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[3] R. J. Anderson. A security policy model for clinical information systems. In *IEEE Symposium on Security and Privacy*, pages 30–43, 1996.

[4] A. Armando, R. Carbone, and L. Compagna. LTL model checking for security protocols. *Journal of Applied Non-Classical Logics*, 19(4):403–429, 2009.

[5] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *ACM Transactions on Programming Languages and Systems*, pages 113–124, 2009.

[6] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Ann. Math. Artif. Intell.*, 22(1-2):5–27, 1998.

[7] M. Balliu, M. Dam, and G. Le Guernic. Epistemic temporal logic for information flow security. *CoRR*, abs/1208.6106, 2012.

[8] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[9] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S. A. Smolka. Model repair for probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 326–340, 2011.

[10] D. A. Basin, V. Jugé, F. Klaedtke, and E. Zalinescu. Enforceable security policies revisited. *ACM Transactions on Information Systems and Security*, 16(1):3, 2013.

[11] D. A. Basin, S. Mödersheim, and L. Viganò. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[12] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, pages 260–272, 2006.

[13] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, pages 126–138, 2007.

[14] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[15] A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 85–100, 2012.

[16] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.

[17] B. Bonakdarpour, A. Ebnenasir, and S. S. Kulkarni. Complexity results in revising UNITY programs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1):1–28, January 2009.

[18] B. Bonakdarpour and S. S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Principles of Distributed Systems (OPODIS)*, pages 408–427, 2008.

[19] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Distributed synthesis of fault-tolerance. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2006. Full version available as a Technical Report MSU-CSE-06-27 at Computer Science and Engineering Department, Michigan State University, East Lansing, Michigan.

[20] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Symbolic synthesis of masking fault-tolerant programs. *Springer Journal on Distributed Computing (DC)*, 25(1):83–108, March 2012.

[21] G. Boudol. Secure information flow as a safety property. In *Formal Aspects in Security and Trust*, pages 20–34, 2008.

[22] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by ai techniques. *Artificial Intelligence*, 112:57–104, 1999.

[23] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *Proceedings of the Royal Society of London*, pages 233–71, 1989.

[24] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[25] K. Chatterjee and V. Raman. Synthesizing protocols for digital contract signing. In *Proceedings of 13th International Conference Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 152–168, 2012.

[26] G. Chatzieleftheriou, B. Bonakdarpour, S. A. Smolka, and P. Katsaros. Abstract model repair. In *NASA Formal Methods Symposium (NFM)*, pages 341–355, 2012.

[27] A. Chudnov, G. Kuan, and D. A. Naumann. Information flow monitoring as abstract interpretation for relational logic. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 48–62, 2014.

[28] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proceedings of CSF*, pages 200–214, 2010.

[29] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of PLDI*, pages 50–62, 2009.

[30] D. Clark and S. Hunt. Non-interference for deterministic interactive programs. In *Proceedings of Formal Aspects in Security and Trust*, pages 50–66, 2008.

[31] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, 1987.

[32] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 265–284, 2014.

[33] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *Computer Security Foundations Workshop*, pages 31–45, 2005.

[34] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[35] R. Corin, S. Etalle, and A. Saptawijaya. A logic for constraint-based security protocol analysis. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 155–168, 2006.

[36] N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 341–356, 2014.

[37] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[38] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[39] R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl. Model checking information flow in reactive systems. In *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 169–185, 2012.

[40] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, P. L. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*

[41] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Reasoning about knowledge: A response by the authors. *Minds and Machines*, 7(1):113, 1997.

[42] Y. Falcone, L. Mounier, J. Fernandez, and J. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

[43] B. Finkbeiner, M. N. Rabe, and C. Sanchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Proceedings of the 27th International Conference on Computer-Aided Verification (CAV)*, 2015. To appear.

[44] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[45] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[46] G. Le Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, pages 75–89, 2006.

[47] K. Havelund, M. R. Lowry, and J. Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

[48] K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Automated Software Engineering (ASE)*, pages 135–143, 2001.

[49] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Applying formal methods to a certifiably secure software system. *IEEE Transactions on Software Engineering*, 34(1):82–98, 2008.

[50] Information Assurance Technology Analysis Center (IATAC). Soar on software security assurance, July 2007. https://buildsecurityin.us-cert.gov/bsi/dhs/902-BSI.html.

[51] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238, 2005.

[52] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *ICISS*, pages 56–70, 2008.

[53] J. Krumm and A.J. Brush. MSR GPS privacy dataset. http://research.microsoft.com/~jckrumm/GPSData2009, 2009.

[54] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[55] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[56] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.

[57] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[58] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, pages 87–100, 2010.

[59] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.

[60] F. Martinelli and I. Matteucci. A framework for automatic generation of security controller. *Software Testing, Verification and Reliability*, 22(8):563–582, 2012.

[61] M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015. To appear.

[62] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580", 1989".

[63] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[64] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels, 1998.

[65] S. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. 197(1):3–16, 2008.

[66] J. C. L. Pimentel, R. Monroy, and D. Hutter. A method for patching interleaving-replay attacks in faulty security protocols. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 117–130, 2007.

[67] J. C. L. Pimentel, R. Monroy, and D. Hutter. On the automated correction of security protocols susceptible to a replay attack. In *European Symposium Research Computer Security (ESORICS)*, pages 594–609, 2007.

[68] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[69] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, pages 573–586, 2006.

[70] F. Pottier and V. Simonet. Information flow inference for ml. In *Proceedings of Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 319–330, 2002.

[71] A. W. Roscoe. CSP and determinism in security modelling. In *IEEE Symposium on Security and Privacy*, pages 114–127, 1995.

[72] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the XXrd IEEE Computer Security Foundations Symposium (CSF)*, pages 186–199, 2010.

[73] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[74] H. Saidi. Toward automatic synthesis of security protocols. AAAI archives, 2002.

[75] R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic generation of local repairs for boolean programs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–10, 2008.

[76] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3:30–50, February 2000.

[77] A. Shamir. How to share a secret. *Communication of ACM*, 22(11):612–613, 1979.

[78] V. Shmatikov and J. C. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283(2):419–450, 2002.

[79] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for büchi automata with appplications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[80] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, pages 352–367, 2005.

[81] H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. *Journal of Computer Security*, 19(6):1029–1082, 2011.

[82] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–, 2003.

[83] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical report, EECS Department, University of California, Berkeley, Oct 2009.