Abstractions for Model Checking System Security

Jason Douglas Franklin

CMU-CS-12-113
April 2012

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Anupam Datta, Chair
Sagar Chaki,
Virgil Gligor,
Jeannette Wing
John Mitchell, Stanford University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the Department of Homeland Security Scholarship and Fellowship Program, the National Science Foundation Graduate Research Fellowship, the NSF Science and Technology Center TRUST.

Keywords: Software Model Checking, Semantic Security, Abstraction, Parametricity, Refinement

To those that favor thought before blind action.

Abstract

Systems software such as operating systems, virtual machine monitors, and hypervisors form the foundation of security for desktop, cloud, and mobile platforms. Despite their ubiquity, these security-critical software systems regularly suffer from serious vulnerabilities in both their design and implementation. A critically important goal is to automatically provide verifiable security guarantees for systems software.

Software model checking is a promising approach toward automated verification of programs. Yet, the size and complexity of systems software presents a major challenge to model checking their security properties. In this thesis, we develop a framework that enables automated, verifiable security guarantees for a wide range of systems software. Central to the framework are novel abstractions that improve the scalability of model checking the security of systems. The abstractions exploit structure common to systems software and the non-determinism inherent in adversary models.

Our key insight is that much of the complexity of model checking security properties of systems software stems from two sources: 1) system data-structures and functions that operate over them with a form of well-structured data-flow, and 2) adversary-controlled data structures and functions that operate over them. We develop a family of techniques to abstract these components of a system.

We introduce the problem of *semantic security verification* and develop three abstractions to improve the scalability of model checking secure systems: 1) *Constrained-to-System-Interface (CSI) adversary* abstraction, a technique to scalably integrate flexible system-specific adversary models into the verification process, 2) *Small Model Analysis*, a family of parametric verification techniques that scale even when a system and adversary operate over unbounded but finite data structures, and 3) *Havoc* abstraction, a source-level analysis technique that reduces the verification complexity associated with system components that operate on adversary-controlled data structures.

We prove that our abstractions are *sound*—no attacks are missed by the abstractions. Further, we prove a completeness result that provides theoretical justification for our zero false positive rate. Finally, we prove a refinement theorem that carries our results to a hypervisor, implemented in a subset of the C programming language.

We perform a number of case studies focused on verifying hypervisors which were designed to enforce a variety of security properties in the presence of adversary-controlled guest operating systems. These empirical evaluations demonstrate the effectiveness of our abstractions on several hypervisors. We identify previously unknown vulnerabilities in the design of SecVisor, the C code of ShadowVisor (a prototype hypervisor), and successfully model check their code using the CBMC model checker after fixing the vulnerabilities. Without the abstractions, CBMC - a state-of-the-art model checker - is unable to model check these systems; in all attempts, it either exhausts system resources or times out after an extended period. With the abstractions, CBMC model checks the systems in a few seconds.

Acknowledgments

First and foremost, I wish to thank my parents, Beth and Doug Franklin, for all their support over the years and my advisor, Anupam Datta, and collaborator, Sagar Chaki, for their patience, guidance, and mentoring.

Additionally, I would like to thank the following groups:

Committee members: Sagar Chaki, Anupam Datta, Virgil Gligor, John Mitchell, and Jeannette Wing.

Undergraduate advisors: Eric Bach and Mary Vernon.

Professors from undergraduate studies: Jin-Yi Cai, Somesh Jha, and Marvin Solomon.

Professors and researchers from graduate school and internships: David Anderson, Mor Harchol-Balter, Arie Gurfinkel, Garth Gibson, Yuri Gurevich, Daniel Kroening, K. Rustan M. Leino, Adrian Perrig, Ofer Strichman, Luis von Ahn, and Helen Wang.

Co-authors: John Bethencourt, Deepak Garg, Limin Jia, Michael Kaminsky, Dilsun Kaynar, Mark Luk, Jonathan M. McCune, Amar Phanishayee, Arvind Seshadri, Parisa Tabriz, Lawrence Tan, Michael Carl Tschantz, Jamie Van Randwyk, Amit Vasudevan, and Vijay Vasudevan.

Department staff: Deborah Cavlovich and Catherine Copetas.

And finally, those friends that haven't been mentioned above: Nels Beckman, Jim Cipar, Mike Dinitz, Alex Grubb, Shiva Kaul, Leonid Kontorvich, Elie Krevat, Dan Lee, Stephen Magill, Rowena Mittal, Abe Othman, Milo Polte, and Elaine Shi.

Contents

1 Introduction **1**

2 The Semantic Security Verification Problem **7**

2.1 Introduction 7
2.2 Definition of Semantic Security 7
2.3 Verification Framework Goals 8
2.4 The CSI-Adversary Abstraction 8
2.4.1 Kernelized Systems and their Adversary Models 9
2.4.2 Adversary Model Framework 9
2.4.3 Flexible, System-Specific Adversary Models 10
2.4.4 Expressiveness of CSI-Adversaries 10
2.4.5 Flexibility, Scalability, and Simplicity 11
2.5 Scalability and Abstraction 12
2.5.1 Overcoming Data Structure Complexity 12
2.5.2 Overcoming Adversary-induced Complexity 13
2.6 Interpreting Verification Results 15

3 Parametric Verification **17**

3.1 Introduction 17
3.2 Motivating Example: SecVisor 20
3.3 Small Model Analysis 23
3.3.1 PGCL Syntax 23
3.3.2 PGCL Semantics 27
3.3.3 Specification Logic 29
3.3.4 Small Model Theorem 35
3.3.5 Proofs of Small Model Theorems 37
3.4 Case Studies 42
3.4.1 SecVisor 43

3.4.2 sHype Security Architecture 48
3.5 Expressiveness and Limitations 53
3.6 Conclusion 56

4 Parametric Verification with Hierarchical Data Structures 59
4.1 Introduction 59
4.2 Address Space Translation and Separation 62
4.3 Definitions of *PGCL*$^+$ and PTSL 65
4.3.1 *PGCL*$^+$ Syntax 65
4.3.2 ShadowVisor Code in *PGCL*$^+$ 66
4.3.3 *PGCL*$^+$ Semantics 69
4.3.4 Specification Formalism 72
4.3.5 Small Model Theorems 76
4.4 Case Studies 77
4.4.1 ShadowVisor 78
4.4.2 Xen 80
4.5 Conclusion 83

5 The Havoc Adversary Abstraction 85
5.1 Introduction 85
5.2 Havoc-Function Abstraction 87
5.2.1 Problem Statement 89
5.2.2 Soundness 90
5.2.3 Details 91
5.3 Case Studies 94
5.3.1 ShadowVisor 94
5.3.2 SecVisor 102
5.4 Conclusion 107

6 Towards Refinement 109
6.1 Introduction 109
6.2 Definition of *PGCL*$^{++}$ 111
6.2.1 *PGCL*$^{++}$ Syntax 111
6.2.2 *PGCL*$^{++}$ Semantics 113
6.2.3 ShadowVisor Code in *PGCL*$^{++}$ 116
6.3 Specification Formalism 117
6.4 Small Model Theorems 119
6.5 *MiniCee* Definition 119
6.5.1 *MiniCee* Syntax 120

6.5.2 *MiniCee* Semantics 122
6.5.3 Concrete Code 123
6.6 Towards Refinement 124
6.7 Conclusion 131

7 Related Work **133**
7.1 Parametric Verification for Correctness 133
7.2 Parametric Verification for Security 134
7.3 Model Checking System Security 135
7.4 Bug Finding 135
7.5 Operating System Verification 136
7.6 QBF Solving 137
7.7 Program Summarization 137

8 Conclusions **139**

A Proofs of Small Model Theorems **141**
A.1 Proofs 141
A.1.1 Introductory Lemmas 141
A.1.2 Store Projection Lemmas 143
A.1.3 Store Projection and Command Lemmas 147
A.1.4 Store Generalization Lemmas 150
A.1.5 Proofs of Lemmas Presented in Main Paper 154

B Proofs of Small Model Theorems for Hierarchical Data Structures **159**
B.1 Proofs 159
B.1.1 Introductory Lemmas 159
B.1.2 Store Projection Lemmas 160
B.1.3 Store Projection and Command Lemmas 164
B.1.4 Store Generalization Lemmas 170
B.1.5 Proofs of Small Model Theorems 175
B.1.6 Proof of Theorem 9 181
B.1.7 Proof of Theorem 10 182
B.1.8 Proof of Corollary 11 183

C Proofs of Small Model Theorems with Write-only Variables **185**
C.1 Proofs 185
C.1.1 Introductory Lemmas 185
C.1.2 Store Projection Lemmas 186

C.1.3 Store Projection and Command Lemmas 190
C.1.4 Store Generalization Lemmas 197
C.1.5 Proofs of Small Model Theorems 202
C.1.6 Proof of Theorem 14 208
C.1.7 Proof of Theorem 15 210
C.1.8 Proof of Corollary 16 210

Bibliography **213**

List of Figures

3.1 Functional overview of SecVisor. 21
3.2 PGCL Syntax 24
3.3 PGCL program for SecVisor. 26
3.4 Rules for expression evaluation. 29
3.5 Rules for commands 30
3.6 Syntax of PTSL 31
3.7 This figure depicts the transition from an initial secure state to the compromised state resulting from an Approved Page Remapping exploit. The attacker modifies KPT entry 01 to point to the writable physical page 00. Subsequently, the CPU executes the unapproved code in page 00 in kernel mode. 43
3.8 This figure depicts the transition from an initial secure state to the compromised state resulting from an Writable Virtual Alias exploit. The attacker modifies KPT entry 00 to be a writable alias for physical page 01 then injects code into 01 using the alias. 45
3.9 Verification results for models with increasing page table sizes. 46

4.1 Typical two-level page table structure. 62
4.2 *PGCL*$^+$ Syntax. Note the addition of nesting where z denotes depth. 66
4.3 ShadowVisor model in *PGCL*$^+$. 68
4.4 Rules for expression evaluation. 70
4.5 Rules for commands 71
4.6 Syntax of PTSL ($1 \leq z \leq d$). In ESF, $Æ_y$ is $\forall$ or $\exists$, at least one $Æ_y$ is $\exists$. 73
4.7 ShadowVisor's vulnerable shadow page fault handler. 78

5.1 A CSI-adversary operates on data that is read by a kernel. 88
5.2 Overview of abstraction process. 91

6.1 *PGCL*$^{++}$ Syntax, z denotes depth. 112
6.2 Rules for expression evaluation in *PGCL*$^{++}$. 114
6.3 Rules for *PGCL*$^{++}$ commands 115

6.4 ShadowVisor model in *PGCL*$^{++}$. 118
6.5 *MiniCee* Syntax 121
6.6 Rules for *MiniCee* commands 122
6.7 ShadowVisor Adversary 124
6.8 ShadowVisor Page Fault Handler 125
6.9 ShadowVisor SPT Invalidation 126
6.10 ShadowVisor New Context Load 126

List of Tables

4.1 ShadowVisor verification with increasing PT size. * means out of 1GB memory limit; Vars, Clauses = # of CNF variables and clauses generated by CBMC. 79
4.2 Xen verification with increasing PT size. * means out of 1GB memory limit; Vars, Clauses = # of CNF variables and clauses generated by CBMC. 82

Chapter 1

Introduction

Background. Systems depend on reference monitors to correctly enforce their security policies in the presence of adversaries who actively interfere with their execution. Operating systems, virtual machine monitors, hypervisors, and web browsers implement the reference-monitor concept by protecting security-sensitive operations and enforcing a security policy. For example, the address translation subsystem of a hypervisor is designed to enforce address space separation in the presence of malicious guest operating systems. If it fails, software running above the hypervisor that relies on address separation is vulnerable to attack. This dependence makes systems software the foundation of security for important platforms including mobile devices, cloud data centers, and desktop computers.

Problem and Challenges. Software systems today lack verifiable security guarantees. This lack of assurance has lead to a bleak security landscape where security-critical systems are continuously vulnerable to compromise. A critically important goal is to automatically provide verifiable security guarantees for systems software. Verifiable security requires that a system, adversary, and security property be formally specified and a formal verification be performed [21]. In this thesis, we make a contribution in this space by developing abstractions that enable automated formal verification of a class of security properties for a class of systems that includes several research and commercial hypervisors. A formal security verification guarantees that a security property holds when a system executes in the presence of an adversary.

The primary reason for the poor assurance levels of systems software is the high cost of verification [50]. Verification costs are dependent on two factors: the level of automation and the complexity of systems software. These costs result in systems being developed and deployed without any guarantees about their security.

Semantic Security. The focus of this thesis is the problem of *semantic security verification*: verifying that every execution of a system in conjunction with a system-specific adversary satisfies a security property [39, 56]. For example, we verify that a security hypervisor satisfies an address separation property in the presence of a malicious guest operating system (OS). We refer to our approach as the verification of "semantic" security to distinguish it from related work aiming to identify patterns of behavior which may or may not lead to security vulnerabilities or searching for bugs without an explicit adversary model [42, 18, 79].

Tiny Hypervisors Aren't Sufficient. Recent progress in developing hypervisors with small code size and narrow interfaces has made automated verification of security hypervisors a more tractable goal [50, 70]. Security lore would have us believe that formally verifying semantic security properties of hypervisors to demonstrate the absence of security flaws requires only a small code base. This perspective has resulted in hypervisors becoming the de facto standard foundation for high-assurance systems [35, 67, 68, 70]. Unfortunately, this "hypervisor security hypothesis" remains unsubstantiated; hypervisors of all sizes regularly suffer from security vulnerabilities in both their designs and implementations [34, 74, 3]. Even hypervisors designed with security as their primary goal have serious security flaws [34, 74, 3]. Further evidence is provided by the fact that we identify serious security flaws in all of the hypervisor implementations we analyze.

Our Approach is Model Checking. Model checking is a promising approach to automatically verify properties of system models [20]. Model checking exhaustively enumerates all states in a model in order to demonstrate the absence of bad behaviors. In the case that a property violation is found, model checkers produce a counterexample, a concrete set of program inputs that lead to the violation. While originally applied to identify bugs in manually-constructed hardware models, model checking has been extended to automatically verify properties of software [19, 7, 46, 17]. The primary challenge with model checking techniques, especially software model checking, is scalability [20]. As system size and complexity grows, the model checker must overcome the exponential growth in the state space or else model checking becomes intractable.

Despite significant progress, the size and complexity of systems software presents a major challenge to model checking semantic security. State-of-the-art model checkers do not scale to the level of complexity of even small hypervisors. The primary challenge is the size and hierarchical nesting of common system data structures, such as page table and access control data structures. By exploiting system structure - in particular, structured data flow common to memory protection subsystems and the nondeterminism inherent in adversary models - we enable scalable software model checking of semantic security. By reducing the costs of semantic security verification, we enable the automated verification of formal security guarantees for an important class of systems.

Thesis Statement. This thesis addresses two primary challenges:

- Scalability concerns with respect to model checking system data structures must be overcome, and
- Adversaries must be scalably integrated in the verification process.

Our thesis statement is that:

> Abstracting data structures in programs with structured data flow and automatically simplifying "havoc" functions that output arbitrary values because of non-determinism inherent in adversary models makes software model checking semantic security feasible.

Abstractions. To demonstrate the validity of this thesis, we develop three abstractions that substantially improve the scalability of model checking security properties of systems: 1) *Constrained-to-System-Interface (CSI) adversary* abstraction, a technique to scalably integrate flexible system-specific adversary models into the verification process, 2) *Small Model Analysis*, a family of parametric verification techniques that scale even when a system and adversary operate over unbounded but finite data structures, and 3) *Havoc* abstraction, a source-level analysis technique that reduces the verification complexity associated with system components that operate on adversary-controlled data structures.

Theoretical Results. We prove that our abstractions are *sound*—no attacks are missed by the abstractions. Further, we prove completeness results for the small model abstraction, providing the theoretical basis for our observed zero false positive rate. Finally, we prove a refinement theorem that carries our results to a hypervisor implemented in a subset of the C programming language.

Case Studies. We perform case studies on three hypervisors; SecVisor, two variants of the popular open-source hypervisor Xen, and ShadowVisor; under a variety of security properties (e.g., $W \otimes X$ and address separation) and adversary models (e.g., adversary-controlled guest OS). These case studies demonstrate the effectiveness of our abstractions: we identify previously unknown vulnerabilities in the design of SecVisor, the C code of ShadowVisor (a prototype hypervisor), and successfully model check their code using the state-of-the-art C Bounded Model Checker (CBMC) after fixing the vulnerabilities. Without the abstractions, CBMC is unable to model check these systems; in all attempts, it either exhausts system resources or times out after an extended period. With the abstractions, CBMC model checks the systems in a few seconds.

Outline. In Chapter 2, we provide an overview of our model checking framework and summarize our contributions. In Chapters 3 and 4, we formally define our system model and develop a family of parametric verification techniques. We apply our techniques to verify both the designs and implementations of commercial and research hypervisors. In Chapter 5, we develop the Havoc abstraction. Finally, in Chapter 6 we prove a refinement theorem that carries our results to a hypervisor implemented in a subset of the C programming language. We detail related work in Chapter 7 and conclude in Chapter 8. Full proofs are included in Appendix A, Appendix B, and Appendix C.

Chapter 2

The Semantic Security Verification Problem

2.1 Introduction

We present an overview of our model checking framework and process including the definition of semantic security, goals of our framework, our flexible approach to defining system-specific adversary models, sources of complexity, approaches to improving scalability, and how to interpret the results of a verification.

2.2 Definition of Semantic Security

Informally, the problem of verifying semantic security is to verify that every execution of a system in conjunction with a system-specific adversary satisfies a security property. For example, we verify that a security hypervisor satisfies an address separation property in the presence of a malicious guest OS. Formally, the problem is stated as follows: given a

CSI-adversary A, a system S, and a security property φ, show that A running in conjunction with S satisfies φ, denoted $A \parallel S \models \varphi$. The adversary and system programs, A and S, are written in a programming language and the security property, φ, is specified using a logic that is described in the next chapter. Note that our formulation closely parallels the model checking problem.

2.3 Verification Framework Goals

We aim to build a framework for verifying semantic security with the following properties:

Expressive: Able to express many systems, security properties, and adversary models.

Flexible: No fixed adversary.

Scalable: Scale to large and complex systems.

Sound: No missed attacks.

Complete: No false alarms.

2.4 The CSI-Adversary Abstraction

We posit that a general approach to verifiable security should enable one to model a variety of adversaries and parametrically reason about different classes of adversaries. This is critical because while specific domains may have a canonical adversary model (e.g., the standard network adversary model for cryptographic protocols), it is unreasonable to expect that a standard adversary model can be developed for all of system security.

2.4.1 Kernelized Systems and their Adversary Models

A common paradigm for secure system design is to layer software on top of a system core (sometimes called a kernel) with an external interface (i.e., a system call interface) that is exposed to potentially-malicious higher-layer software. A goal of systems with this architecture is to remain secure when executing concurrently with an adversary that has complete control over higher-layer (i.e., lower privilege) software.

Since the adversary has complete control over higher-layer software, we abstract the higher-layer software and use the system-call interface exposed to the adversary as a definition of the adversary's capabilities. This endows the adversary with the ability to make interface calls with arbitrary input parameter values, compute arbitrary functions utilizing fresh values and the return values of interface calls, and the ability to input computed values as parameters to system calls.

2.4.2 Adversary Model Framework

To design our framework for specifying adversary models, we divide an adversary model into three aspects: the adversary's capabilities, complexity, and goals.

Capabilities: We employ capabilities, rather than attacks, to distinguish between an adversary endowed with a set of basic capabilities and one that is limited to a known set of attacks. The former is clearly preferred, as it enables the adversary to carry out known and unknown attacks by combining capabilities in new and novel ways.

Complexity: The adversary's complexity is the number of primitive steps the adversary may take in their attempt to break the system's security. Two options are common: either the adversary is polynomially bounded in some parameter, or the adversary may perform an unbounded number of steps. In our case, the systems of interest are meant to be secure no matter how many attacks are launched, hence we endow the adversary with an unbounded number of computational steps (e.g., system calls, messages sent, etc.).

Goals: In this work, the adversary's goal is simple: use their endowed capabilities to violate the system's security property.

2.4.3 Flexible, System-Specific Adversary Models

We introduce the notion of a **C**onstrained-to-**S**ystem-**I**nterface **(CSI)**-adversary. A key insight underlying the development of these flexible system-specific adversary models is to view a trusted system in terms of the interfaces that the various components expose: larger trusted components are built by combining interface calls in known ways; the adversary is confined to the interfaces it has access to, but may combine interface calls without restriction. Such interface-confined adversaries provide a generic way to model different classes of adversaries. For example, an operating system may consist of subsystems for memory protection, I/O management, and scheduling.

2.4.4 Expressiveness of CSI-Adversaries

CSI-adversary models are widespread in systems security. Our abstract view of an adversary is expressive enough to model a wide variety of systems adversary models. Consider the following examples.

Remote Adversary. A canonical adversary that captures the threats of remote exploitation, the remote adversary, is constrained to the system interface exposed by the networking subsystem.

Malicious Guest OS. An adversary model for hypervisors and VMMs, the malicious guest OS, is a CSI-adversary that is constrained to the VMM's hypercall interface.

Malicious Gadget. An adversary model for web browsers, the gadget adversary [10], is a CSI-adversary constrained to the read and write interfaces for frames guarded by the same origin policy as well as the frame navigation policies.

2.4.5 Flexibility, Scalability, and Simplicity

There are a number of benefits from specifying adversaries according to the CSI-adversary framework.

Flexibility. CSI-adversaries are flexible. The CSI-adversary framework allows adversary models to be strengthened and weakened in a natural way. This allows a system to be verified against a diverse set of possible adversary models. Consider a CSI-adversary model $A = \{\mathtt{I}_1, \mathtt{I}_2, \ldots, \mathtt{I}_\mathtt{n}\}$ composed of n interfaces and $A' \subset A$. We say that A' is weaker than A (or A is stronger than A') since the set of traces of interface calls that A' can generate is a subset of the traces of A.

Simplicity. Specifying a CSI-adversary is simple; an adversary is just the composition of a system interface. The definition of composition may vary; in our case studies, we use parallel composition with interleaving semantics.

Scalability. Finally, the CSI-adversary model forms the basis for a class of abstractions (discussed in Chapter 5) that improve the scalability of verification.

In our work on the logic of secure systems [21, 36], we developed an alternative formulation of the CSI-adversary framework tailored to hand-written proofs of security properties.

2.5 Scalability and Abstraction

Next, we describe the challenges inherent in model checking hypervisors and give an overview of our results. In addition to the CSI-adversary abstraction, we develop two classes of abstractions that enable scalable verification.

First, we overcome the challenges associated with verifying systems that operate on large and hierarchical data structures, by exploiting system structure inherent in the memory management subsystems that enforce memory protection.

Secondly, we address the scalability challenges introduced by the non-deterministic nature of CSI-adversaries. The two classes of abstractions are complementary, and in the cases we consider, the simultaneous application of both abstractions is necessary to enable successful verification.

2.5.1 Overcoming Data Structure Complexity

The size and hierarchical nesting of the data structures over which systems operate raises challenges for successful application of model checking. A common approach is to bound data structures sizes to a small cutoff which reduces the cost of verification. However, doing so leaves the question of soundness open; if the system is secure when the system and adversary operate on larger data structures. We consider the following question: Does the verification of a bounded system imply the verification of the system with unbounded, but finite data structures?

To answer this question, we develop a family of sound and complete parametric verification techniques that scale even when reference monitors and adversaries operate over unbounded, but finite data structures. Specifically, we develop a parametric guarded command language for modeling reference monitors and adversaries. We also present a parametric temporal specification logic for expressing security policies that the monitor is expected to enforce.

Our central technical results are a set of small model theorems (SMTs). We develop our results incrementally by developing SMTs first, for a single parametric array, and second, for the more complex case of a tree of parametric arrays. For a single array, our theorems state that in order to verify that a policy is enforced by a reference monitor with an arbitrarily large data structure, it is sufficient to model check the monitor with just one entry in its data structure. We apply our methodology to verify the designs of a number of hypervisors including SecVisor, the sHype mandatory-access-control extension to Xen, the shadow paging mechanisms of Xen version 3.0.3, and ShadowVisor, a research hypervisor developed for the x86 platform. Our approach is able to prove that sHype and a variant of the original SecVisor design correctly enforces the expected security properties in the presence of powerful adversaries. Chapter 4 generalizes our SMTs to apply to a tree of parametric arrays of arbitrary depth. This generalization requires new conceptual and technical insights, and brings interesting systems (such as multi-level paging and context caching as used in Xen) within the scope of analysis.

2.5.2 Overcoming Adversary-induced Complexity

A CSI-adversary model defines a security boundary between unknown, adversarial code and a system. A system must correctly enforce its security properties given arbitrary behavior by the CSI-adversary. CSI-adversaries pass data to the system as parameters in the system calls. Because this data originates from the adversary, it is considered non-deterministic. In many systems, the adversary can pass an address of an adversary-controlled data structure. For example, in shadow paging hypervisors, a malicious guest OS has control of the kernel page tables. In cases like these, the adversary controls both the size of and contents of the data structure. Thus, model checking is complicated since the model checker will try (and fail) to exhaustively enumerate data structures of all sizes. Because of the non-deterministic nature of CSI-adversaries, our SMTs do not apply to these adversary-controlled data structures and the code that manipulates them.

Consider the case of hypervisors designed to enforce address separation properties among adversarially-controlled guest operating systems running on top of them and between the hypervisor and guest's address spaces. Observing that significant model checking complexity stems from adversary-controlled data structures (e.g., kernel page tables) and functions that operate over them (e.g., for reading and writing kernel page tables), we develop an automated technique to abstract these components of a system.

The abstraction involves: (i) detecting "havoc" functions, i.e., functions that return non-deterministic values, and (ii) replacing calls to havoc functions with non-deterministic assignments. We develop an approach to detecting havoc functions based on checking validity of Quantified Boolean Formulas (QBF).

We prove that our abstraction is *sound*—no attacks are missed by the abstraction. In addition, our approach includes an efficient technique to proving a form of completeness directly at the source level which we term local completeness. We formulate the problem of proving completeness as a Quantified Boolean Formula validity problem. This formulation allows us to exploit recent progress in the development of efficient QBF solvers and to operate directly at the source code level. We implement the abstraction technique on top of state-of-the-art QBF solvers.

Our empirical evaluation demonstrates the effectiveness of our abstractions on real software: we identify two previously unknown vulnerabilities in the C code of ShadowVisor (a prototype hypervisor), and successfully model check its code using the CBMC model checker after fixing the vulnerabilities. In addition, we model check two semantic security properties of the SecVisor hypervisor. The vulnerabilities allow a malicious guest OS to gain complete control of the underlying hypervisor, rendering its protections useless. Without the abstractions, CBMC is unable to model check these systems; in all attempts, it either exhausts system resources or times out after an extended period. With the abstractions, CBMC verifies that the hypervisor's C implementation correctly enforces address separation in the presence of a strong adversary model in approximately 3 seconds.

2.6 Interpreting Verification Results

An important question to ask is: What does the outcome of a verification imply about the security of a system? Our verification techniques are sound, meaning that we never suffer from false negatives. Therefore, there are two possible outcomes of a verification: 1) verification fails and a counterexample is generated, and 2) verification succeeds and the property holds.

In the failure case, the model checker outputs the trace that may lead to a property violation. Only if the verification framework is complete are we guaranteed that a violation is not a false positive and hence represents a real vulnerability. In this case, the counterexample contains a set of inputs which represent an exploit. While our small model theorems are backed by provable guarantees of completeness, our havoc abstraction has weaker local completeness guarantees. While the potential to produce false positives exists, we have yet to experience a false positive in any of our case studies. We believe that custom tailoring our abstractions to a class of programs of interest has proven to be a successful strategy for build a precise verification framework that experiences few false positives.

Given that the framework developed in this thesis is provably sound, a successful verification guarantees that a security property of a system holds when executing in parallel with an adversary. Interpreting the security implications of successful verifications must be performed on a case-by-case basis. For example, consider the case of ShadowVisor. During verification, we identified a vulnerability whereby a CSI-adversary constrained to

ShadowVisor's system call interface could allocate a physical memory page in the ShadowVisor's protected memory region. This region contains ShadowVisor's code and data. This allocation is clearly a violation of address separation and could lead to an attack where the adversary overwrites hypervisor code and thereby hijacks system control-flow. Once we fixed the vulnerability, verification was successful. We now know that there are no traces whereby the adversary can successfully violate the address separation property. Because our adversary model allows any sequence of interface calls, verification implies that specific attacks composed of sequences of interface calls can not violate our system's security property. The ability of a CSI-adversary to encompass a wide variety of specific attacks is one of the key strengths of our methodology.

Chapter 3

Parametric Verification

3.1 Introduction

A major challenge in verifying reference monitors in systems software and hardware is *scalability*: typically, the verification task either requires significant manual effort (e.g., using interactive theorem proving techniques) or becomes computationally intractable (e.g., using automated finite state model checking techniques). The development of systems software such as microkernels and hypervisors with relatively small code size and narrow interfaces alleviates the scalability problem. However, another significant factor for automated verification techniques is the size of the data structures over which the programs operate. For example, the complexity of finite state model checking reference monitors in hypervisors and virtual machine monitors grows exponentially in the size of the page tables used for memory protection.

This chapter develops a verification technique that scales even when reference monitors and adversaries operate over very large data structures. The technique extends parametric verification techniques developed for system correctness to the setting of a class of secure systems and adversaries. Specifically, we develop a parametric guarded command language for modeling reference monitors and adversaries, and a parametric temporal specification logic for expressing security policies that the monitor is expected to enforce. Data structures such as page tables are modeled in the language using an array where the number of rows in the array is a parameter that can be instantiated differently to obtain systems of different sizes. The security policies expressed in the logic also refer to this parameter.

The central technical results of the chapter are a set of *small model theorems* that state that for any system M expressible in the language, any security property φ expressible in the logic, and any natural number n, $M(n)$ satisfies $\varphi(n)$ if and only if $M(1)$ satisfies $\varphi(1)$ where $M(i)$ and $\varphi(i)$ are the instances of the system and security policy, respectively, when the data structure size is i. For example, $M(n)$ may model a hypervisor operating on page tables of size n in parallel with an adversary that is interacting with it, and $\varphi(n)$ may express a security policy that any of the n pages (protected by the page table) containing kernel code is not modified during the execution of the system. The consequence of the small model theorem is that in order to verify that the policy is enforced for an arbitrarily large page table, it is sufficient to model check the system with just one page table entry. The small model analysis framework is described in Section 3.3.

The twin design goals for the programming language are *expressivity* and *data independence*: the first goal is important in order to model practical reference monitors and adversaries; the second is necessary to prove small model theorems that enable scalable verification. The language provides a parametric array where the number of rows is a parameter and the number of columns is fixed. In order to model reference monitor operations such as synchronizing kernel page tables with shadow page tables (commonly used in software-based memory virtualization), it is essential to provide support for *atomic whole array operations* over the parametric array. In addition, such whole array operations are needed in order to model an adversary that can non-deterministically set the values of an entire column of the parametric array (e.g., the permission bits in the kernel page table if the adversary controls the guest operating system). On the other hand, all operations are row-independent, i.e., the data values in one row of the parametric array do not affect the values in a different row. Also, the security properties expressible as reachability properties in the specification logic refer to properties that hold either in all rows of the array or in some row. Intuitively, it is possible to prove a small model theorem (Theorem 1) for all programs in the language with respect to such properties because of the row-independent nature of the operations. Row-independence is also the reason for the existence of the simulation relations necessary to prove the small model theorem for the temporal formulas in our specification logic (Theorem 2). The logic is expressive enough to capture separation-style access control policies commonly enforced by hypervisors and virtual machines as well as history-dependent policies (e.g., no message send after reading a sensitive file).

We apply this methodology to verify that the designs of two hypervisors—SecVisor [70] and the sHype [68] mandatory-access-control extension to Xen [9]—correctly enforce the expected security properties in the presence of adversaries. For SecVisor, we check the security policy that a guest operating system should only be able to execute user-approved code in kernel mode. The original version of SecVisor does not satisfy this property in a small model with one page table entry. We identify two attacks while attempting to model check the system, repair the design, and then successfully verify the small model. For sHype, we successfully check the Chinese Wall Policy [15], a well-known access control policy used in commercial settings. These applications are presented in Section 3.4. In addition, we further demonstrate the expressivity of the programming language by showing in Section 3.5 how to encode any finite state reference monitor that operates in a row-independent manner as a program; the associated security policy can be expressed in the logic.

This thesis builds on a line of work on data independence, introduced in an influential paper by Wolper [78]. He considered programs whose control-flow behavior was completely independent of the data over which they operated. His motivating example was the alternating bit protocol. Subsequent work on parametric verification, which we survey in the related work (Chapter 7), has relaxed this strong independence assumption to permit the program's control-flow to depend in limited ways on the data. The closest line of work to ours is by Emerson and Kahlon [25], and Lazic et al. on verifying correctness of cache coherence protocols [55, 54]. However, there are significant differences in our technical approach and results. In particular, since both groups focus on correctness and not security, they do not support the form of whole array operations that we do in order to model atomic access control operations in systems software as well as a non-deterministic adversary.

3.2 Motivating Example: SecVisor

We use SecVisor [70], a security hypervisor, as our motivating example. We informally discuss the design of SecVisor below. In subsequent sections, we show how our small model analysis approach enables us to model SecVisor and verify that it satisfies the desired security properties even with arbitrarily large protection data structures (page tables).

SecVisor supports a single guest operating system, GUEST, which executes in two privilege levels – user mode and kernel mode. In addition, GUEST supports two types of executable code – approved and unapproved. We assume that the set of approved code is fixed in advance and remains unchanged during system execution. Figure 3.1 shows a functional overview of SecVisor. K is the GUEST kernel. A is the adversary. S is SecVisor. PM stands for physical memory. KPT and SPT are the kernel and shadow page tables, respectively. A shaded shape represents an active element or code while an unshaded

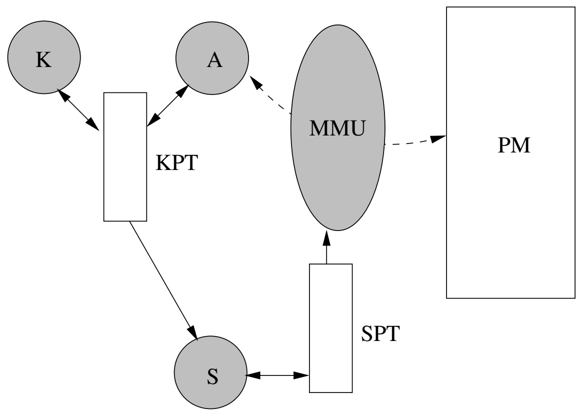


Figure 3.1: Functional overview of SecVisor.

shape is a data structure. The direction of arrows indicates the kind of access (read or write) of data structure by element. We now describe SecVisor's protection mechanisms, adversary model, and security properties.

Protection Mechanisms Memory is organized in pages, which are indexed and accessed via page tables. Each page table entry contains the starting address and other attributes (e.g., read-write-execute permissions, approved or unapproved status) of the corresponding page. GUEST maintains a Kernel Page Table (KPT). However, in order to provide the desired security guarantees even when GUEST is compromised, SecVisor sets its memory protection bits in a separate shadow page table (SPT). The SPT is used by the Memory Management Unit (MMU) to determine whether a process should be allowed to access a physical memory page. To provide adequate functionality, the SPT is synchronized with the KPT when necessary. This is useful, for example, when GUEST transitions between user and kernel modes, and when the permissions in KPT are updated.

Adversary Model. SecVisor's attacker controls everything except the Trusted Computing Base (TCB) – the CPU, MMU, and physical memory (PM). The attacker is able to read and write the KPT, and thus modify the SPT indirectly via synchronization. Therefore, to achieve the desired security properties, it is critical that SecVisor's page table synchronization logic be correct. These capabilities model a very powerful and realistic attacker who is aware of vulnerabilities in GUEST's kernel and application software, and uses these vulnerabilities to locally or remotely exploit the system.

Security Properties. SecVisor's design goal is to ensure that only approved code executes in the kernel mode of GUEST. To this end, SecVisor requires that the following two properties be satisfied: (i) *execution integrity*, which mandates that GUEST should only execute instructions from memory pages containing approved code while in kernel mode, and (ii) *code integrity*, which stipulates that memory pages containing approved code should only be modifiable by SecVisor and its TCB. We now describe these two properties in more detail. We refer to memory pages containing approved and unapproved code as approved and unapproved pages, respectively.

Execution Integrity. We assume that only code residing in executable memory pages can be loaded and executed. Any attempt to violate this condition results in an exception. Therefore, to satisfy execution integrity we require that in kernel mode, the executable permission of all unapproved memory pages are turned off.

Code Integrity. In general, memory pages are accessed by software executing on the CPU and peripheral devices (via DMA). Since all non-SecVisor code and all peripheral devices are outside SecVisor's TCB, the code integrity requirement reduces to the following property: approved pages should not be modifiable by any code executing on the CPU, except for SecVisor, or by any peripheral device. Thus, to satisfy code integrity, SecVisor marks all approved pages as read-only to all entities other than itself and its TCB.

3.3 Small Model Analysis

In this section, we describe our small model analysis approach to analyze security properties of parametric systems. In our approach, a parametric system is characterized by a single parametric array P, which is instantiated to some finite but arbitrary constant size during any specific system execution (e.g., SecVisor's page tables). In addition, the system has a finite number of other variables (e.g., SecVisor's mode bit). Parametric systems are modeled as programs in our model parametric guarded command language (PGCL), while security properties of interest are expressed as formulas in our parametric temporal specification logic (PTSL). We prove small model theorems that imply that a PTSL property φ holds on a PGCL program *Prog* for arbitrarily large instantiations of P if and only if φ holds on *Prog* with a small instance of P. In the rest of this section, we present PGCL and PTSL, interleaved with the formal SecVisor model and security properties, as well as the small model theorems that enable parametric verification of SecVisor and other similar systems.

3.3.1 PGCL Syntax

For simplicity, we assume that all variables in PGCL are Boolean. The parametric array P is two-dimensional, where the first dimension (i.e., number of rows) is arbitrary, and the second dimension (i.e., number of columns) is fixed. All elements of P are also Boolean. Note that Boolean variables enable us to encode finite valued variables, finite arrays with finite valued elements, records with finite valued fields, and relations and functions with finite domains over such variables.

Let K be the set of numerals corresponding to the natural numbers $\{1, 2, \ldots\}$. We fix the number of columns of P to some $q \in K$. Note that this does not restrict the set of systems we are able to handle since our technical results hold for any q. Let $\top$ and $\bot$ be, respectively,

Natural Numerals	$\mathtt{K}$			
Index Variable	$\mathtt{i}$			
Boolean Variables	$\mathtt{B}$			
Parametric Variable	$\mathtt{n}$			
Expressions	$\mathtt{E}$	$::=$	$\top \mid \bot \mid * \mid \mathtt{B} \mid \mathtt{E} \vee \mathtt{E} \mid \mathtt{E} \wedge \mathtt{E} \mid \neg \mathtt{E}$	
Parameterized Expressions	$\widehat{\mathtt{E}}$	$::=$	$\mathtt{E} \mid \mathtt{P}_{\mathtt{n,q}}[\mathtt{i}][\mathtt{K}] \mid \widehat{\mathtt{E}} \vee \widehat{\mathtt{E}} \mid \widehat{\mathtt{E}} \wedge \widehat{\mathtt{E}} \mid \neg \widehat{\mathtt{E}}$	
Instantiated Guarded Commands	$\mathtt{G}$	$::=$	$\mathtt{GC}(\mathtt{K})$	
Guarded Commands	$\mathtt{GC}$	$::=$	$\mathtt{E}\ ?\ \mathtt{C}$	Guarded command
		$\mid$	$\mathtt{GC} \parallel \mathtt{GC}$	Parallel composition
Commands	$\mathtt{C}$	$::=$	$\mathtt{B} := \mathtt{E}$	Assignment
		$\mid$	$\mathtt{for}\ \mathtt{i}\ :\ \mathtt{P}_{\mathtt{n,q}}\ \mathtt{do}\ \widehat{\mathtt{E}}\ ?\ \widehat{\mathtt{C}}$	Parametric loop
		$\mid$	$\mathtt{C};\mathtt{C}$	Sequencing
Parameterized Commands	$\widehat{\mathtt{C}}$	$::=$	$\mathtt{P}_{\mathtt{n,q}}[\mathtt{i}][\mathtt{K}] := \widehat{\mathtt{E}}$	Param. array assignment
		$\mid$	$\widehat{\mathtt{C}};\widehat{\mathtt{C}}$	Sequencing

Figure 3.2: PGCL Syntax

the representations of the truth values **true** and **false**. Let *B* be a set of Boolean variables, *I* be a set of variables used to index into a row of $\mathtt{P}$, and $\mathtt{n}$ be the variable used to store the number of rows of $\mathtt{P}$.

The syntax of PGCL is shown in Figure 3.2. Expressions in PGCL include natural numbers, Boolean variables, a parameterized array $\mathtt{P}_{\mathtt{n,q}}$, a variable $\mathtt{i}$ for indexing into $\mathtt{P}_{\mathtt{n,q}}$, a variable $\mathtt{n}$ representing the size of $\mathtt{P}_{\mathtt{n,q}}$, propositional expressions over Boolean variables and elements of $\mathtt{P}_{\mathtt{n,q}}$. For notational simplicity, we often write $\mathtt{P}$ to mean $\mathtt{P}_{\mathtt{n,q}}$. The commands in PGCL include guarded commands that update Boolean variables and elements of $\mathtt{P}_{\mathtt{n,q}}$, and parallel and sequential compositions of guarded commands. A guarded command executes by first evaluating the guard; if it is true, then the command that follows is executed. The parallel composition of two guarded commands executes by non-deterministically picking one of the commands to execute. The sequential composition of two commands executes the first command followed by the second command.

SecVisor in PGCL

We give an overview of our PGCL model of SecVisor followed by the PGCL program for SecVisor.

Example 1. *For our purposes, the key unbounded data structures maintained by SecVisor are the KPT and the SPT. Hence, we represent these two page tables using our parameterized array* $\mathtt{P}_{\mathtt{n},\mathtt{q}}$. *Without loss of generality, we assume that the KPT and the SPT have the same number of entries. Thus, each row of* $\mathtt{P}_{\mathtt{n},\mathtt{q}}$ *represents a KPT entry, and the corresponding SPT entry. The columns of* $\mathtt{P}_{\mathtt{n},\mathtt{q}}$ – `KPTRW`, `KPTX`, `KPTPA`, `SPTRW`, `SPTX` *and* `SPTPA` – *represent the permissions and page types of KPT and SPT entries. Specifically,* `P[i][KPTRW]` *and* `P[i][KPTX]` *refer to read/write and execute permissions for the* `i`*-th KPT entry. Also,* `P[i][KPTPA]` *refers to the type, i.e., kernel code (*`KC`*), kernel data (*`KD`*), or user memory (*`UM`*), of the page mapped by the* `i`*-th KPT entry. The* `SPTRW`*,* `SPTX`*, and* `SPTPA` *columns are defined analogously for SPT entries. Finally, the variable* `MODE` *indicates if the system is in* `KERNEL` *or* `USER` *mode.*

The SecVisor program is a parallel composition of four guarded commands:

$$\mathtt{SecVisor} \equiv \mathtt{Kernel\_Entry} \parallel \mathtt{Kernel\_Exit} \parallel \mathtt{Sync} \parallel \mathtt{Attacker}$$

These four guarded commands represent, respectively, entry to kernel mode, exit from kernel mode, page table synchronization, and attacker action. We now describe each of these commands in more detail. Note, in particular, the extensive use of whole array operations to model the protection mechanisms in SecVisor as well as the non-deterministic adversary updates to the KPT.

Kernel Entry. If the system is in user mode then a valid transition is to kernel mode. On a transition to kernel mode, SecVisor's entry handler sets the SPT such that the kernel

```
Kernel_Entry ≡
  $\neg$kernelmode ? kernelmode := $\top$;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = UM ?
      $P_{n,q}$[i][SPTRW] := $\top$; $P_{n,q}$[i][SPTX] := $\bot$;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = KC ?
      $P_{n,q}$[i][SPTRW] := $\bot$; $P_{n,q}$[i][SPTX] := $\top$;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = KD ?
      $P_{n,q}$[i][SPTRW] := $\top$; $P_{n,q}$[i][SPTX] := $\bot$
```

(a)

```
Kernel_Exit ≡
  kernelmode ? kernelmode := $\bot$;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = UM ?
      $P_{n,q}$[i][SPTRW] := $\top$; $P_{n,q}$[i][SPTX] := $\top$;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = KC ?
      $P_{n,q}$[i][SPTRW] := $\bot$; $P_{n,q}$[i][SPTX] := $\bot$;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = KD ?
      $P_{n,q}$[i][SPTRW] := $\top$; $P_{n,q}$[i][SPTX] := $\bot$
```

(b)

```
Sync ≡
  $\top$ ? for i : $P_{n,q}$ do
    $\top$ ? $P_{n,q}$[i][SPTPA] :=
      $P_{n,q}$[i][KPTPA]
```

(c)

```
Attacker ≡
  $\top$ ? for i : $P_{n,q}$ do
    $P_{n,q}$[i][KPTPA] := $*$;
    $P_{n,q}$[i][KPTRW] := $*$;
    $P_{n,q}$[i][KPTX] := $*$
```

(d)

Figure 3.3: PGCL program for SecVisor.

code becomes executable and the kernel data and user memory become non-executable. This prevents unapproved code from being executed during kernel mode, and is modeled by the guarded command shown in Figure 3.3(a).

Kernel Exit. If the system is in kernel mode then a valid transition is to transition to user mode. On a transition to user mode, the program sets: (i) the mode to user mode, and (ii) the SPT such that user memory becomes executable and kernel code pages become non-executable. This is modeled by the guarded command shown in Figure 3.3(b).

Page Table Synchronization. SecVisor synchronizes the SPT with the KPT when the kernel: (i) wants to use a new KPT, or (ii) modifies or creates a KPT entry. An attacker can modify the kernel page table entries. Hence, SecVisor must prevent the attacker's modifications from affecting the SPT fields that enforce code and execution integrity. SecVisor's design specifies that to prevent adversary modification of sensitive SPT state, SecVisor may not copy permission bits from the kernel page table during synchronization with the shadow page table. We model this by the guarded command shown in Figure 3.3(c).

Attacker Action. The attacker arbitrarily modifies every field of every KPT entry, including the read/write permissions, execute permissions, and physical address mapping for user memory, kernel code, and kernel data. We model this by the guarded command shown in Figure 3.3(d) where the expression $*$ non-deterministically evaluates to either **true** or **false**.

3.3.2 PGCL Semantics

We present the operational semantics of PGCL as a relation on "stores". Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{B}$ be the truth values $\{\textbf{true}, \textbf{false}\}$. For any numeral k we write $\lceil k \rceil$ to mean the natural number represented by k in standard arithmetic. Often, we write k to mean $\lceil k \rceil$ when the context disambiguates such usage. For two natural numbers j and k such that $j \leq k$, we write $[j,k]$ to mean the set $\{j,\ldots,k\}$ of numbers in the closed range between j and k. We write $Dom(f)$ to mean the domain of a function f. Then, a store σ is a tuple $(\sigma^B, \sigma^n, \sigma^P)$ such that:

- $\sigma^B : \mathtt{B} \to \mathbb{B}$ maps Boolean variables to $\mathbb{B}$;
- $\sigma^n : \mathbb{N}$ is the value of $\mathtt{n}$;
- $\sigma^P : [1, \sigma^n] \times [1, \lceil \mathtt{q} \rceil] \to \mathbb{B}$ is a function that maps $\mathtt{P}$ to a two-dimensional Boolean array.

Equivalently, by Currying, we also treat σ^P as a function of type $[1,\sigma^n] \to [1,\lceil \mathtt{q} \rceil] \to \mathbb{B}$. In the rest of this chapter, we omit the relevant superscript of σ when it is clear from the context. For example, we write $\sigma(b)$ to mean $\sigma^B(b)$. The rules for evaluating PGCL expressions under stores are presented in Figure 3.4. These rules are defined via induction on the structure of expressions. To define the semantics of PGCL, we have to first present the notion of store projection.

Definition 1 (Store Projection)**.** *Let* $\sigma = (\sigma^B, \sigma^n, \sigma^P)$ *be any store. For* $i \in [1, \sigma^n]$ *we write* $\sigma \downharpoonleft i$ *to mean the store* $(\sigma^B, 1, X)$ *such that* $X(1) = \sigma^P(i)$.

Intuitively, $\sigma \downharpoonleft i$ is constructed by retaining σ^B, setting σ^n to 1, and projecting away all but the i-th row from σ^P. Note that since projection retains σ^B, it does not affect the evaluation of expressions that do not refer to elements of $\mathtt{P}_{\mathtt{n},\mathtt{q}}$.

We overload the $\cdot[\cdot \mapsto \cdot]$ operator in the following way. First, for any function $f : X \to Y$, $x \in X$ and $y \in Y$, we write $f[x \mapsto y]$ to mean the function that is identical to f, except that x is mapped to y. Second, for any PGCL expression or guarded command X, variable v, and expression e, we write $\mathtt{X}[\mathtt{v} \mapsto \mathtt{e}]$ to mean the result of replacing all occurrences of v in X simultaneously with e.

Store Transformation. For any PGCL command c and stores σ and σ', we write $\{\sigma\}\ \mathtt{c}\ \{\sigma'\}$ to mean that σ is transformed to σ' by the execution of c. The rules defining $\{\sigma\}\ \mathtt{c}\ \{\sigma'\}$, via induction on the structure of c, are shown in Figure 3.5. Most of the definitions are straightforward. For example, the "GC" rule states that σ is transformed to σ' by executing the guarded command e ? c if: (i) the guard e evaluates to **true** under σ, and (ii) σ is transformed to σ' by executing the command c.

The "Unroll" rule states that if c is a `for` loop, then $\{\sigma\}\ \mathtt{c}\ \{\sigma'\}$ if there exists appropriate intermediate stores that represent the state of the system after the execution of each iteration of the loop. The premise of the "Unroll" rule involves the instantiation of the loop variable i with specific values. This is achieved via the $\gg$ notation, which we define next.

$$\frac{}{\langle\top,\sigma\rangle \to \textbf{true}} \qquad \frac{}{\langle\bot,\sigma\rangle \to \textbf{false}} \qquad \frac{}{\langle *,\sigma\rangle \to \textbf{true}} \qquad \frac{}{\langle *,\sigma\rangle \to \textbf{false}}$$

$$\frac{\mathtt{b} \in dom(\sigma^B)}{\langle \mathtt{b},\sigma\rangle \to \sigma^B(\mathtt{b})} \qquad \frac{\lceil \mathtt{k} \rceil \le \sigma^n \qquad \lceil \mathtt{l} \rceil \le \lceil \mathtt{q} \rceil}{\langle \mathtt{P_{n,q}}[\mathtt{k}][\mathtt{l}],\sigma\rangle \to \sigma^P(\lceil \mathtt{k} \rceil, \lceil \mathtt{l} \rceil)}$$

$$\frac{\langle \mathtt{e},\sigma\rangle \to t \qquad \langle \mathtt{e}',\sigma\rangle \to t'}{\langle \mathtt{e} \vee \mathtt{e}',\sigma\rangle \to t''} \text{ where } t'' = \textbf{true} \text{ if } t = \textbf{true} \text{ or } t' = \textbf{true}, \text{ and } t'' = \textbf{false} \text{ otherwise.}$$

$$\frac{\langle \mathtt{e},\sigma\rangle \to t \qquad \langle \mathtt{e}',\sigma\rangle \to t'}{\langle \mathtt{e} \wedge \mathtt{e}',\sigma\rangle \to t''} \text{ where } t'' = \textbf{true} \text{ if } t = \textbf{true} \text{ and } t' = \textbf{true}, \text{ and } t'' = \textbf{false} \text{ otherwise.}$$

$$\frac{\langle \mathtt{e},\sigma\rangle \to t}{\langle \neg\mathtt{e},\sigma\rangle \to t'} \text{ where } t' = \textbf{true} \text{ if } t = \textbf{false}, \text{ and } t' = \textbf{false} \text{ otherwise.}$$

Figure 3.4: Rules for expression evaluation.

Definition 2 (Loop Variable Instantiation)**.** *Let σ and σ' be two stores such that $\sigma^n = \sigma'^n = N$, and $\widehat{\mathtt{e}} \mathbin{?} \widehat{\mathtt{c}} \in \widehat{\mathtt{E}} \mathbin{?} \widehat{\mathtt{C}}$ be a guarded command containing the index variable $\mathtt{i}$. Then for any $\lceil \mathtt{j} \rceil \in [1,N]$, we write $\{\sigma\}\ (\widehat{\mathtt{e}} \mathbin{?} \widehat{\mathtt{c}})(\mathtt{i} \gg \mathtt{j})\ \{\sigma'\}$ to mean:*

$$\{\sigma \downharpoonright \lceil \mathtt{j} \rceil\}\ (\widehat{\mathtt{e}} \mathbin{?} \widehat{\mathtt{c}})[\mathtt{i} \mapsto \mathtt{1}]\ \{\sigma' \downharpoonright \lceil \mathtt{j} \rceil\} \bigwedge$$

$$\forall k \in [1,N] \centerdot k \neq \lceil \mathtt{j} \rceil \Rightarrow \sigma^P(k) = \sigma'^P(k)$$

Thus, $\{\sigma\}\ (\widehat{\mathtt{e}} \mathbin{?} \widehat{\mathtt{c}})(\mathtt{i} \gg \mathtt{j})\ \{\sigma'\}$ means that σ' is obtained from σ by first replacing $\mathtt{i}$ with $\mathtt{j}$ in $\widehat{\mathtt{e}} \mathbin{?} \widehat{\mathtt{c}}$, and then executing the resulting guarded command.

3.3.3 Specification Logic

Next we present our specification logic. We support two types of specifications – reachability properties and temporal logic specifications. Reachability properties are useful for

$$\frac{\sigma^n = \sigma'^n = \lceil \mathtt{k} \rceil \qquad \{\sigma\}\ \mathtt{gc}\ \{\sigma'\}}{\{\sigma\}\ \mathtt{gc(k)}\ \{\sigma'\}}\text{Parameter Instantiation}$$

$$\frac{\langle \mathtt{e},\sigma\rangle \to t}{\{\sigma\}\ \mathtt{b} := \mathtt{e}\ \{\sigma[\sigma^B \mapsto \sigma^B[\mathtt{b} \mapsto t]]\}}\text{Assign} \qquad \frac{\langle \mathtt{e},\sigma\rangle \to \textbf{true} \qquad \{\sigma\}\ \mathtt{c}\ \{\sigma'\}}{\{\sigma\}\ \mathtt{e}\ ?\ \mathtt{c}\ \{\sigma'\}}\text{GC}$$

$$\frac{\{\sigma\}\ \mathtt{gc}\ \{\sigma'\} \vee \{\sigma\}\ \mathtt{gc}'\ \{\sigma'\}}{\{\sigma\}\ \mathtt{gc} \parallel \mathtt{gc}'\ \{\sigma'\}}\text{Parallel} \qquad \frac{\{\sigma\}\ \mathtt{c}\ \{\sigma''\} \qquad \{\sigma''\}\ \mathtt{c}'\ \{\sigma'\}}{\{\sigma\}\ \mathtt{c};\mathtt{c}'\ \{\sigma'\}}\text{Sequential}$$

$$\frac{\langle \mathtt{e},\sigma\rangle \to t \qquad \lceil \mathtt{i} \rceil \leq \sigma^n \qquad \lceil \mathtt{j} \rceil \leq \lceil \mathtt{q} \rceil}{\{\sigma\}\ \mathtt{P_{n,q}[i][j]} := \mathtt{e}\ \{\sigma[\sigma^P \mapsto \sigma^P[(\lceil \mathtt{i} \rceil, \lceil \mathtt{j} \rceil) \mapsto t]]\}}\text{Parameterized Array Assign}$$

$$\frac{\sigma^n = N \qquad \exists \sigma_1, \ldots, \sigma_{N+1} \centerdot \sigma = \sigma_1 \wedge \sigma' = \sigma_{N+1} \wedge \forall \lceil \mathtt{j} \rceil \in [1,N] \centerdot \{\sigma_{\lceil \mathtt{j} \rceil}\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}})(\mathtt{i} \gg \mathtt{j})\ \{\sigma_{\lceil \mathtt{j} \rceil + 1}\}}{\{\sigma\}\ \mathtt{for\ i\ :\ P_{n,q}\ do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}\ \{\sigma'\}}\text{Unroll}$$

Figure 3.5: Rules for commands

checking whether the target system is able to reach a state that exhibits a specific condition, e.g., a memory page storing kernel code is made writable. Reachability properties are expressed via "state formulas." In addition, state formulas are also used to specify the initial condition under which the target system begins execution.

Syntax. The syntax of state formulas is defined in Figure 3.6. Note that we support three distinct types of state formulas – universal, existential, and generic – which differ in the way they quantify over rows of the parametric array P. Specifically, universal formulas allow one universal quantification over P, existential formulas allow one existential quantification over P, while generic formulas allow one universal and one existential quantification over P.

In contrast, temporal logic specifications enable us to verify a rich class of properties over the sequence of states observed during the execution of the target system, e.g., once a

$$
\begin{array}{rlll}
\text{Basic Propositions} & \mathsf{BP} & ::= & \mathtt{b}\ ,\ \mathtt{b} \in B \\
 & & \mid & \neg\mathsf{BP} \\
 & & \mid & \mathsf{BP} \wedge \mathsf{BP}
\end{array}
$$

$$
\begin{array}{rlll}
\text{Parametric Propositions} & \mathsf{PP}(\mathtt{i}) & ::= & \{\mathtt{P}_{\mathtt{n},\mathtt{q}}[\mathtt{i}][\mathtt{l}] \mid \lceil \mathtt{l} \rceil \leq \lceil \mathtt{q} \rceil\} \\
 & & \mid & \neg\mathsf{PP}(\mathtt{i}) \\
 & & \mid & \mathsf{PP}(\mathtt{i}) \wedge \mathsf{PP}(\mathtt{i})
\end{array}
$$

$$
\begin{array}{rlll}
\text{Universal State Formulas} & \mathsf{USF} & ::= & \mathsf{BP} \\
 & & \mid & \forall \mathtt{i}.\mathsf{PP}(\mathtt{i}) \\
 & & \mid & \mathsf{BP} \wedge \forall \mathtt{i}.\mathsf{PP}(\mathtt{i})
\end{array}
$$

$$
\begin{array}{rlll}
\text{Existential State Formulas} & \mathsf{ESF} & ::= & \mathsf{BP} \\
 & & \mid & \exists \mathtt{i}.\mathsf{PP}(\mathtt{i}) \\
 & & \mid & \mathsf{BP} \wedge \exists \mathtt{i}.\mathsf{PP}(\mathtt{i})
\end{array}
$$

$$
\begin{array}{rlll}
\text{Generic State Formulas} & \mathsf{GSF} & ::= & \mathsf{USF} \\
 & & \mid & \mathsf{ESF} \\
 & & \mid & \mathsf{USF} \wedge \mathsf{ESF}
\end{array}
$$

$$
\begin{array}{rlll l}
\mathsf{PTSL}\ \text{Path Formulas} & \mathsf{TLPF} & ::= & \mathsf{TLF} & \text{“state formula”} \\
 & & \mid & \mathsf{TLF} \wedge \mathsf{TLF} & \text{“conjunction”} \\
 & & \mid & \mathsf{TLF} \vee \mathsf{TLF} & \text{“disjunction”} \\
 & & \mid & \mathbf{X}\ \mathsf{TLF} & \text{“in the next state”} \\
 & & \mid & \mathsf{TLF}\ \mathbf{U}\ \mathsf{TLF} & \text{“until”}
\end{array}
$$

$$
\begin{array}{rlll l}
\mathsf{PTSL}\ \text{Formulas} & \mathsf{TLF} & ::= & \mathsf{USF} \mid \neg\mathsf{USF} & \text{“propositions and their negations”} \\
 & & \mid & \mathsf{TLF} \wedge \mathsf{TLF} & \text{“conjunction”} \\
 & & \mid & \mathsf{TLF} \vee \mathsf{TLF} & \text{“disjunction”} \\
 & & \mid & \mathbf{A}\ \mathsf{TLPF} & \text{“for all computation paths”}
\end{array}
$$

Figure 3.6: Syntax of PTSL

sensitive file is read, in all future states network sends are forbidden. In our approach, such specifications are expressed as formulas of the temporal logic PTSL. In essence, PTSL is a subset of the temporal logic ACTL* [20] with USF as atomic propositions.

SecVisor's Security Properties in PTSL. We now formalize SecVisor's security properties in our specification logic. We assume that only kernel code is approved by SecVisor. We require that SecVisor begins execution in KERNEL mode, where only kernel code pages are executable. Thus, the initial state of SecVisor is expressed by the following USF state formula:

$$\varphi_{init} \triangleq \mathtt{MODE} = \mathtt{KERNEL} \wedge \forall \mathtt{i}\,.\, \mathtt{P}[\mathtt{i}][\mathtt{SPTX}] \Rightarrow (\mathtt{P}[\mathtt{i}][\mathtt{SPTPA}] = \mathtt{KC})$$

In addition, the execution and code integrity properties are expressed in our specification logic as follows:

1. Execution Integrity: Recall that this property requires that in kernel mode, only kernel code should be executable. It is stated as follows:

   $$\varphi_{exec} \triangleq \mathtt{MODE} = \mathtt{KERNEL} \Rightarrow (\forall \mathtt{i}\,.\, \mathtt{P}[\mathtt{i}][\mathtt{SPTX}] \Rightarrow (\mathtt{P}[\mathtt{i}][\mathtt{SPTPA}] = \mathtt{KC}))$$

   To verify this property, we check if the system is able to reach a state where its negation holds. The negation of φ_{exec} is expressed as the following ESF state formula:

   $$\neg\varphi_{exec} \triangleq \mathtt{MODE} = \mathtt{KERNEL} \wedge (\exists \mathtt{i}\,.\, \mathtt{P}[\mathtt{i}][\mathtt{SPTX}] \wedge \neg(\mathtt{P}[\mathtt{i}][\mathtt{SPTPA}] = \mathtt{KC}))$$

2. Code Integrity: Recall that this property requires that every kernel code page should be read-only. It is expressed as follows:

   $$\varphi_{code} \triangleq \forall \mathtt{i}\,.\, ((\mathtt{P}[\mathtt{i}][\mathtt{SPTPA}] = \mathtt{KC}) \Rightarrow (\neg \mathtt{P}[\mathtt{i}][\mathtt{SPTRW}]))$$

   To verify this property, we check if the system is able to reach a state where its negation holds. The negation of φ_{code} is expressed as the following ESF state formula:

$$\neg\varphi_{code} \triangleq \exists \mathtt{i}\,\text{.}\,((\mathtt{P[i][SPTPA]} = \mathtt{KC}) \wedge \mathtt{P[i][SPTRW]})$$

Semantics. We now present the semantics of our specification logic. We start with the notion of satisfaction of formulas by stores.

Definition 3. *The satisfaction of a formula* π *by a store* σ *(denoted* $\sigma \models \pi$*) is defined, by induction on the structure of* π*, as follows:*

- $\sigma \models \mathtt{b}$ *iff* $\sigma^B(\mathtt{b}) = \mathbf{true}$.
- $\sigma \models \mathtt{P}_{\mathtt{n},\mathtt{q}}[\mathtt{k}][\mathtt{l}]$ *iff* $\lceil \mathtt{k} \rceil \leq \sigma^n$ *and* $\sigma^P(\lceil \mathtt{k} \rceil, \lceil \mathtt{l} \rceil) = \mathbf{true}$.
- $\sigma \models \neg\pi$ *iff* $\sigma \not\models \pi$.
- $\sigma \models \pi_1 \wedge \pi_2$ *iff* $\sigma \models \pi_1$ *and* $\sigma \models \pi_2$.
- $\sigma \models \pi_1 \vee \pi_2$ *iff* $\sigma \models \pi_1$ *or* $\sigma \models \pi_2$.
- $\sigma \models \forall \mathtt{i}\,\text{.}\,\pi$ *iff* $\forall i \in [1, \sigma^n]\,\text{.}\,\sigma \downharpoonleft i \models \pi[\mathtt{i} \mapsto 1]$.
- $\sigma \models \exists \mathtt{i}\,\text{.}\,\pi$ *iff* $\exists i \in [1, \sigma^n]\,\text{.}\,\sigma \downharpoonleft i \models \pi[\mathtt{i} \mapsto 1]$.

The definition of satisfaction of Boolean formulas and the logical operators are standard. Parametric formulas, denoted $\mathtt{P}_{\mathtt{n},\mathtt{q}}[\mathtt{k}][\mathtt{l}]$, are satisfied if and only if the index $\mathtt{k}$ is in bounds, and the element at the specified location is **true**. An universally quantified formula, $\forall \mathtt{i}\,\text{.}\,\pi$, is satisfied by σ if and only if all projections of σ satisfy $\pi[\mathtt{i} \mapsto 1]$. Finally, an existentially quantified formula, $\exists \mathtt{i}\,\text{.}\,\pi$, is satisfied by σ if and only if there exists a projection of σ that satisfies $\pi[\mathtt{i} \mapsto 1]$.

We now present the semantics of a PGCL program as a *Kripke structure*. We use this Kripke semantics subsequently to prove a small model theorem for PTSL specifications.

Kripke Semantics. Let gc be any PGCL guarded command and $\mathtt{k} \in \mathtt{K}$ be any numeral. We denote the set of stores σ such that $\sigma^n = \lceil \mathtt{k} \rceil$, as $Store(\mathtt{gc}(\mathtt{k}))$. Note that $Store(\mathtt{gc}(\mathtt{k}))$ is finite. Let *Init* be any PTSL formula and $\mathsf{AP} = \mathsf{USF}$ be the set of atomic propositions. Intuitively, a Kripke structure $M(\mathtt{gc}(\mathtt{k}), Init)$ over AP is induced by executing $\mathtt{gc}(\mathtt{k})$ starting from any store $\sigma \in Store(\mathtt{gc}(\mathtt{k}))$ that satisfies *Init*.

Definition 4. *Let Init* $\in$ USF *be any* PTSL *formula. Formally,* $M(\mathtt{gc}(\mathtt{k}), Init)$ *is a four tuple* $(\mathcal{S}, I, \mathcal{T}, \mathcal{L})$*, where:*

- $\mathcal{S} = Store(\mathtt{gc}(\mathtt{k}))$ *is a set of states;*
- $I = \{\sigma | \sigma \models Init\}$ *is a set of initial states;*
- $\mathcal{T} = \{(\sigma, \sigma') \mid \{\sigma\}\mathtt{gc}(\mathtt{k})\{\sigma'\}\}$ *is a transition relation given by the operational semantics of* PGCL*; and*
- $\mathcal{L} : \mathcal{S} \to 2^{\mathsf{AP}}$ *is the function that labels each state with the set of propositions true in that state; formally,*

$$\forall \sigma \in \mathcal{S} \centerdot \mathcal{L}(\sigma) = \{\varphi \in \mathsf{AP} \mid \sigma \models \varphi\}$$

If ϕ is a PTSL formula, then $M, \sigma \models \phi$ means that ϕ holds at state σ in the Kripke structure M. We use a standard inductive definition of the relation $\models$ [20]. Informally, an atomic proposition π holds at σ iff $\sigma \models \pi$; $\mathbf{A}\ \phi$ holds at σ if ϕ holds on all possible (infinite) paths starting from σ. TLPF formulas hold on paths. Informally, a TLF formula ϕ holds on a path Π iff it holds at the first state of Π; $\mathbf{X}\ \phi$ holds on a path Π iff ϕ holds on the suffix of Π starting at second state of Π; $\phi_1\ \mathbf{U}\ \phi_2$ holds on Π if ϕ_1 holds on suffixes of Π until ϕ_2 begins to hold. The definitions for $\neg$, $\wedge$ and $\vee$ are standard.

3.3.4 Small Model Theorem

We now present two small model theorems – one for reachability properties, and one for PTSL specifications. Both theorems relate the behavior of a PGCL program when P has arbitrarily many rows to its behavior when P has a single row. We defer the proofs of the theorems to Section 3.3.5.

Definition 5. *A Kripke structure $M(\mathtt{gc(k)}, Init)$ exhibits a formula φ iff there is a reachable state σ of $M(\mathtt{gc(k)}, Init)$ such that $\sigma \models \varphi$.*

Theorem 1 (Small Model Safety 1)**.** *Let $\mathtt{gc(k)}$ be any instantiated guarded command. Let $\varphi \in$ GSF be any generic state formula, and $Init \in$ USF be any universal state formula. Then $M(\mathtt{gc(k)}, Init)$ exhibits φ iff $M(\mathtt{gc(1)}, Init)$ exhibits φ.*

We now prove a small model theorem that relates Kripke structures via simulation. The following example motivates the form of Theorem 2.

Example 2. *Consider the example of a system that restricts message transmission after a principal accesses sensitive data. Suppose the system consists of an arbitrary number of principals, each of whom is modeled by a row of* P. *Also, suppose that the columns* READ *and* SEND *represent, respectively, that the sensitive data has been read, and that a message has been transmitted. Then, the fact that no principal has read sensitive data is encoded by the following* USF *proposition:*

$$NotRead \triangleq \forall \mathtt{i} \centerdot \neg \mathtt{P}_{\mathtt{n,q}}[\mathtt{i}][\mathtt{READ}]$$

Also, the fact that no principal has sent a message is encoded by the following USF *proposition:*

$$NotSent \triangleq \forall \mathtt{i} \centerdot \neg \mathtt{P}_{\mathtt{n,q}}[\mathtt{i}][\mathtt{SEND}]$$

Therefore, our security property is expressed by the following PTSL *formula:*

$$\mathbf{AG}(NotRead \vee \mathbf{AXG}(NotSent))$$

where $\mathbf{G}\phi$ *is a path formula meaning that* ϕ *holds at all states of a path, and is a shorthand for* $(\phi\ \mathbf{U}\ \mathbf{false})$. *Proving this temporal formula with a small model requires the following small model theorem.*

Simulation. The following small model theorem relies on that fact that PTSL formulas are preserved by simulation between Kripke structures. We use the standard definition of simulation [20], as presented next.

Definition 6. *Let* $M_1 = (\mathcal{S}_1, I_1, \mathcal{T}_1, \mathcal{L}_1)$ *an* $M_2 = (\mathcal{S}_2, I_2, \mathcal{T}_2, \mathcal{L}_2)$ *be two Kripke structures over sets of atomic propositions* AP_1 *and* AP_2 *such that* $\mathsf{AP}_2 \subseteq \mathsf{AP}_1$. *Then* M_1 *is simulated by* M_2*, denoted by* $M_1 \preceq M_2$*, iff there exists a relation* $H \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ *such that the following three conditions hold:*

(C1) $\forall s_1 \in \mathcal{S}_1 \centerdot \forall s_2 \in \mathcal{S}_2 \centerdot (s_1, s_2) \in H \Rightarrow \mathcal{L}_1(s_1) \cap \mathsf{AP}_2 = \mathcal{L}_2(s_2)$

(C2) $\forall s_1 \in I_1 \centerdot \exists s_2 \in I_2 \centerdot (s_1, s_2) \in H$

(C3) $\forall s_1, s_1' \in \mathcal{S}_1 \centerdot \forall s_2 \in \mathcal{S}_2 \centerdot (s_1, s_2) \in H \wedge (s_1, s_1') \in \mathcal{T}_1 \Rightarrow$
$\exists s_2' \in \mathcal{S}_2 \centerdot (s_2, s_2') \in \mathcal{T}_2 \wedge (s_1', s_2') \in H$

It is known [20] that the satisfaction of ACTL* formulas is preserved by simulation. Therefore, since PTSL is a subset of ACTL*, it is also preserved by simulation. This is expressed formally by the following fact, which we state without proof.

Fact 1. *Let* M_1 *and* M_2 *be two Kripke structures over propositions* AP_1 *and* AP_2 *such that* $M_1 \preceq M_2$. *Hence, by Definition 6,* $\mathsf{AP}_2 \subseteq \mathsf{AP}_1$. *Let* φ *be any* PTSL *formula over* AP_2. *Therefore,* φ *is also a* PTSL *formula over* AP_1. *Then* $M_2 \models \varphi \Rightarrow M_1 \models \varphi$.

Theorem 2 (Small Model Simulation)**.** *Let* $\texttt{gc(k)}$ *be any instantiated guarded command. Let Init* $\in \mathsf{GSF}$ *be any generic state formula. Then* $M(\texttt{gc(k)}, \mathit{Init}) \preceq M(\texttt{gc(1)}, \mathit{Init})$ *and* $M(\texttt{gc(1)}, \mathit{Init}) \preceq M(\texttt{gc(k)}, \mathit{Init})$.

The following is a corollary of Theorem 2. Note that this corollary is the dual of Theorem 1 obtained by swapping the types of φ and *Init*.

Corollary 3 (Small Model Safety 2)**.** *Let* $\texttt{gc(k)}$ *be any instantiated guarded command. Let* $\varphi \in \mathsf{USF}$ *be any universal state formula, and Init* $\in \mathsf{GSF}$ *be any generic state formula. Then* $M(\texttt{gc(k)}, \mathit{Init})$ *exhibits* φ *iff* $M(\texttt{gc(1)}, \mathit{Init})$ *exhibits* φ.

Proof. Follows from: (i) the observation that exhibition of a USF formula ϕ is expressible in PTSL as the TLF formula $\mathbf{F}\ \phi$, (ii) Theorem 2, and (iii) Fact 1. □

3.3.5 Proofs of Small Model Theorems

In this section, we prove our small model theorems[1]. We first present a set of supporting lemmas for the proof of Theorem 1. The proofs of these lemmas, along with the statements and proofs of lemmas on which they rely, can be found in the appendix.

Two types of lemmas follow: (i) store projection lemmas that show that the effect of executing a PGCL program carries over from larger stores to unit stores, and (ii) store generalization lemmas that show that the effect of executing a PGCL program carries over from the unit store to larger stores. Intuitively, the two types of lemmas enable the forward and backward reasoning necessary for the proof of Theorem 1's forward and reverse implications, respectively.

The first lemma states that a store σ satisfies an universal state formula φ iff every projection of σ satisfies φ.

[1]The reader may skip this section without loss of continuity.

Lemma 4. *Let* $\varphi \in \mathsf{USF}$ *and* σ *be any store. Then:*

$$\sigma \models \varphi \Leftrightarrow \forall i \in [1, \sigma^n] \centerdot \sigma \downharpoonleft i \models \varphi$$

The next lemma states that if a store σ satisfies a generic state formula φ, then some projection of σ satisfies φ.

Lemma 5. *Let* $\varphi \in \mathsf{GSF}$ *and* σ *be any store. Then:*

$$\sigma \models \varphi \Rightarrow \exists i \in [1, \sigma^n] \centerdot \sigma \downharpoonleft i \models \varphi$$

The next lemma states that if a store σ is transformed to σ' by executing an instantiated guarded command $\texttt{gc(k)}$, then every projection of σ is transformed to the corresponding projection of σ' by executing $\texttt{gc(k)}$.

Lemma 6 (Instantiated Command Projection)**.** *For any stores* σ, σ' *and instantiated guarded command* $\texttt{gc(k)}$*:*

$$\{\sigma\}\ \texttt{gc(k)}\ \{\sigma'\} \Rightarrow \forall i \in [1, \sigma^n] \centerdot \{\sigma \downharpoonleft i\}\ \texttt{gc(1)}\ \{\sigma' \downharpoonleft i\}$$

The last lemma relating store projection and formulas states that if every projection of a store σ satisfies a generic state formula φ, then σ satisfies φ.

Lemma 7. *Let* $\varphi \in \mathsf{GSF}$ *and* σ *be any store. Then:*

$$\forall i \in [1, \sigma^n] \centerdot \sigma \downharpoonleft i \models \varphi \Rightarrow \sigma \models \varphi$$

So far, we used the concept of store projection to show that the effect of executing a PGCL program carries over from larger stores to unit stores (i.e., stores obtained via projection). To prove our small model theorems, we also need to show that the effect of executing a PGCL program propagates in the opposite direction, i.e., from unit stores to larger stores. To this end, we first present a notion, called store generalization, that relates unit stores to those of arbitrarily large size.

Definition 7 (Store Generalization)**.** *Let* $\sigma = (\sigma^B, 1, \sigma^P)$ *be any store. For any* $k \in \mathbb{N}$ *we write* $\sigma \upharpoonleft k$ *to mean the store satisfying the following condition:*

$$(\sigma \upharpoonleft k)^n = k \wedge \forall i \in [1,k] \centerdot (\sigma \upharpoonleft k) \downharpoonleft i = \sigma$$

Intuitively, $\sigma \upharpoonleft k$ is constructed by duplicating k times the only row of σ^P, and leaving the other components of σ unchanged. We now present a lemma related to store generalization, which is needed for the proof of Theorem 1. The lemma states that if a store σ is transformed to σ' by executing an instantiated guarded command $\mathtt{gc(k)}$, then every generalization of σ is transformed to the corresponding generalization of σ' by executing $\mathtt{gc(k)}$.

Lemma 8 (Instantiated Command Generalization)**.** *For any stores* σ, σ' *and instantiated guarded command* $\mathtt{gc(1)}$*:*

$$\{\sigma\}\ \mathtt{gc(1)}\ \{\sigma'\} \Rightarrow \forall \lceil \mathtt{k} \rceil \in \mathbb{N} \centerdot \{\sigma \upharpoonleft \lceil \mathtt{k} \rceil\}\ \mathtt{gc(k)}\ \{\sigma' \upharpoonleft \lceil \mathtt{k} \rceil\}$$

Proof of Theorem 1

We now prove Theorem 1. We utilize the preceding store projection lemmas for the forward implication and the generalization lemmas for the reverse implication.

Proof. For the forward implication, let $\sigma_1, \sigma_2, \ldots, \sigma_n$ be a sequence of states of

$M(\texttt{gc(k)},\mathit{Init})$ such that:

$$\sigma_1 \models \mathit{Init} \bigwedge \sigma_n \models \varphi \bigwedge \forall i \in [1,n-1] \centerdot \{\sigma_i\}\ \texttt{gc(k)}\ \{\sigma_{i+1}\}$$

Since $\varphi \in \mathsf{GSF}$, by Lemma 5 we know that:

$$\exists j \in [1,\lceil \texttt{k} \rceil] \centerdot \sigma_n \downharpoonleft j \models \varphi$$

Let j_0 be such a j. By Lemma 4, since $\mathit{Init} \in \mathsf{USF}$:

$$\sigma_1 \downharpoonleft j_0 \models \mathit{Init}$$

By Lemma 6, we know that:

$$\forall i \in [1,n-1] \centerdot \{\sigma_i \downharpoonleft j_0\}\ \texttt{gc(1)}\ \{\sigma_{i+1} \downharpoonleft j_0\}$$

Therefore, $\sigma_n \downharpoonleft j_0$ is reachable in $M(\texttt{gc(1)},\mathit{Init})$ and $\sigma_n \downharpoonleft j_0 \models \varphi$. Hence, $M(\texttt{gc(1)},\mathit{Init})$ exhibits φ. For the reverse implication, let $\sigma_1,\sigma_2,\ldots,\sigma_n$ be a sequence of states of $M(\texttt{gc(1)},\mathit{Init})$ such that:

$$\sigma_1 \models \mathit{Init} \bigwedge \sigma_n \models \varphi \bigwedge \forall i \in [1,n-1] \centerdot \{\sigma_i\}\ \texttt{gc(1)}\ \{\sigma_{i+1}\}$$

For each $i \in [1,n]$, let $\widehat{\sigma_i} = \sigma_i \upharpoonleft \lceil \texttt{k} \rceil$. Therefore, since $\mathit{Init} \in \mathsf{USF}$, by Lemma 4, we know:

$$\forall j \in [1,\lceil \texttt{k} \rceil] \centerdot \widehat{\sigma_1} \downharpoonleft j \models \mathit{Init} \Rightarrow \widehat{\sigma_1} \models \mathit{Init}$$

Also, since $\varphi \in \mathsf{GSF}$, by Lemma 7 we know that:

$$\forall j \in [1, \lceil \mathtt{k} \rceil] \centerdot \widehat{\sigma_n} \downharpoonright j \models \varphi \Rightarrow \widehat{\sigma_n} \models \varphi$$

Finally, by Lemma 8, we know that:

$$\forall i \in [1, n-1] \centerdot \{\widehat{\sigma_i}\} \ \mathtt{gc(k)} \ \{\widehat{\sigma_{i+1}}\}$$

Therefore, $\widehat{\sigma_n}$ is reachable in $M(\mathtt{gc(k)}, \mathit{Init})$ and $\widehat{\sigma_n} \models \varphi$. Hence, $M(\mathtt{gc(k)}, \mathit{Init})$ exhibits φ. This completes the proof.

□

Proof of Theorem 2

We now prove Theorem 2.

Proof. Recall the conditions **C1–C3** in Definition 6 for simulation. For the first simulation, we propose the following relation $\mathcal{H}$ and show that it is a simulation relation:

$$(\sigma, \sigma') \in \mathcal{H} \Leftrightarrow \exists i \in [1, \lceil \mathtt{k} \rceil] \centerdot \sigma' = \sigma \downharpoonright i$$

C1 holds because our atomic propositions are USF formulas, and Lemma 4; **C2** holds because $\mathit{Init} \in \mathsf{GSF}$ and Lemma 5; **C3** holds by Definition 4 and Lemma 6. For the second simulation, we propose the following relation $\mathcal{H}$ and show that it is a simulation relation:

$$(\sigma, \sigma') \in \mathcal{H} \Leftrightarrow \sigma' = \sigma \upharpoonright \lceil \mathtt{k} \rceil$$

Again, **C1** holds because our atomic propositions are USF formulas, Definition 7, and

Lemma 4; **C2** holds because *Init* $\in$ GSF, Definition 7, and Lemma 7; **C3** holds by Definition 4 and Lemma 8. This completes the proof. □

Note the asymmetry between Theorem 1 and Theorem 2. Ideally, we would like to prove a dual of Theorem 2 with *Init* $\in$ USF, and the atomic propositions of PTSL being GSF. Then, Theorem 1 would be a corollary of this dual theorem. Unfortunately, such a dual of Theorem 2 is difficult to prove. Specifically, the problem shows up when proving that $M(\texttt{gc(k)}, \mathit{Init}) \preceq M(\texttt{gc(1)}, \mathit{Init})$. Suppose we attempt to prove this by showing that the following relation is a simulation:

$$(\sigma, \sigma') \in \mathcal{H} \Leftrightarrow \exists i \in [1, \lceil \texttt{k} \rceil] \centerdot \sigma' = \sigma \downharpoonleft i$$

Unfortunately, Lemma 5 is too weak to imply that $\mathcal{H}$ satisfies even condition **C1**- specifically this is because in the consequent of Lemma 5's implication we have $\exists i$ instead of $\forall i$. Indeed, since GSF subsumes ESF, replacing $\exists i$ with $\forall i$ in Lemma 5 results in an invalid statement. Essentially, this loss of validity stems from the fact that the whole-array updates in PGCL allow different rows to be assigned different values. On the other hand, Lazic et al. [55] allow only a more restricted form (i.e., reset) of whole-array updates. This enables Lazic et al. to prove simulation, but reduces the expressivity of their modeling language. As noted earlier, we found this additional expressivity in PGCL to be crucial for modeling SecVisor.

3.4 Case Studies

We demonstrate our methodology on two hypervisors: SecVisor and the sHype [68] mandatory-access-control extension to Xen [9].

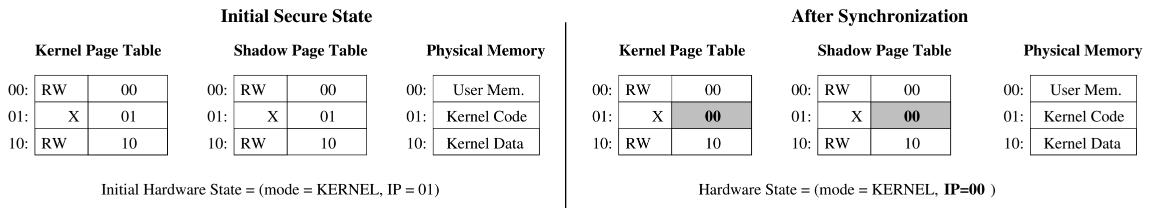


Figure 3.7: This figure depicts the transition from an initial secure state to the compromised state resulting from an Approved Page Remapping exploit. The attacker modifies KPT entry 01 to point to the writable physical page 00. Subsequently, the CPU executes the unapproved code in page 00 in kernel mode.

3.4.1 SecVisor

Recall our model of SecVisor from Section 3.3.1 and the expression of SecVisor's security properties as PTSL formulas from Section 3.3.3.

Initial Failed Verification and Vulnerabilities

We used the Murφ model checker to verify φ_{exec} and φ_{code} on our SecVisor model. Murφ discovered counterexamples to both properties. Based on these counterexamples, we identified vulnerabilities in SecVisor's design. We crafted exploits to ensure that the vulnerabilities were also exploitable in SecVisor's implementation.[2] Both vulnerabilities result from flaws in Sync.

Approved Page Remapping. The first vulnerability, called Approved Page Remapping, was derived from Murφ's counterexample to φ_{exec}. The counterexample involves the attacker modifying a $\mathtt{P}_{\mathtt{n},\mathtt{q}}$ row with $\mathtt{SPTPA} = \mathtt{KC}$ and $\mathtt{SPTX} = \top$. Specifically, the attacker changes the value of KPTPA from KC to UM. The new KPTPA value is then copied by Sync into SPTPA. Since $\mathtt{SPTX} = \top$, this results in a violation of φ_{exec}. The key flaw here, in

[2]These vulnerabilities were identified independently by two of SecVisor's authors during an audit of SecVisor's implementation, but were not reported in any peer-reviewed publication. However, an informal update to the SecVisor paper [70] detailing the vulnerabilities is available.

terms of SecVisor's operation, is that a KPT entry is copied into the SPT without ensuring that kernel code virtual addresses are not mapped to physical pages containing kernel data or user memory. Subsequently, the CPU is in a position to execute arbitrary (and possibly malicious) code. Figure 3.7 illustrates this attack.

To demonstrate that this vulnerability is present in SecVisor's implementation, we crafted an exploit (about 37 lines of C) as a kernel module. This exploit modifies the physical address of a page table entry mapping an approved code page, to point to a page containing unapproved code. When executed on a SecVisor-protected Linux kernel running on an AMD SVM platform, our exploit overwrote a page table entry which originally mapped a physical page containing approved code to point to an arbitrary (unapproved) physical page. SecVisor copied this entry into the SPT, potentially permitting the CPU to execute unapproved code in kernel mode.

Our current exploit implementation requires that SecVisor approve the kernel module containing the exploit code for execution in kernel mode. This is unrealistic since any reasonable approval policy will prohibit the execution of kernel modules obtained from untrusted sources. However, it could be possible for the attacker to run our exploit without loading a kernel module. This could be done by executing pre-existing code in the kernel that modifies the kernel page table entries with attacker-specified parameters. The attacker could, for example, exploit a control-flow vulnerability (such as a buffer overflow) in the kernel to call the kernel page table modification routine with attacker-supplied parameters.

Writable Virtual Alias. The second vulnerability, called Writable Virtual Alias, was derived from Murφ's counterexample to φ_{code}. The counterexample involves the attacker modifying a $\mathtt{P}_{\mathtt{n,q}}$ row with $\mathtt{SPTPA} = \mathtt{UM}$ and $\mathtt{SPTRW} = \top$. Specifically, the attacker changes the value of `KPTPA` from `UM` to `KC`. The new `KPTPA` value is then copied by `Sync` into `SPTPA`. Since $\mathtt{SPTRW} = \top$, this results in a violation of φ_{code}. The key flaw here, in

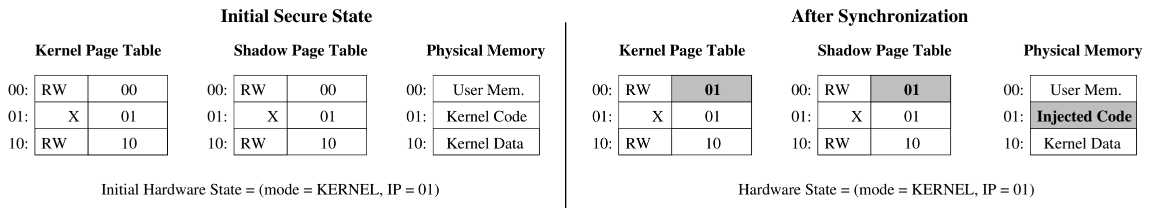


Figure 3.8: This figure depicts the transition from an initial secure state to the compromised state resulting from an Writable Virtual Alias exploit. The attacker modifies KPT entry 00 to be a writable alias for physical page 01 then injects code into 01 using the alias.

terms of SecVisor's operation, is that a KPT entry is copied into the SPT without ensuring that virtual addresses mapped to kernel data or user memory are not replaced by virtual addresses mapped to kernel code. Then, the attacker uses the writable virtual alias to inject arbitrary (and possibly malicious) code into kernel code pages. Figure 3.8 illustrates the attack.

To demonstrate that this vulnerability is present in SecVisor's implementation, we created an exploit using about 15 lines of C code. Our exploit opens `/dev/mem`, maps a user page with write permissions to a physical page (at address `KERNEL_CODE_ADDR`) containing approved kernel code, and writes an arbitrary value into the target physical page via the virtual user page. When executed against SecVisor on an AMD SVM platform running Linux 2.6.20.14, our exploit successfully overwrote an approved kernel code with arbitrary code. An interesting aspect of the exploit is that it can be executed from user mode by an attacker that has administrative privileges.

Repair and Final Successful Verification

Both vulnerabilities in SecVisor are due to `Sync` copying untrusted data into the SPT without validation. Our fix introduces two checks into `Sync`, resulting in a `Secure_Sync`, shown by the following guarded command:

Verification	States	Rules Fired	Time	Memory	Result
3 PTEs	55,296	2,156,544	2.52 sec.	8MB	Success
4 PTEs	1,744,896	88,989,696	343.97 sec.	256MB	Success
5 PTEs	-	-	-	-	Out of Memory

Figure 3.9: Verification results for models with increasing page table sizes.

```
Secure_Sync ≡
  $\top$ ? for i : $P_{n,q}$ do
    $(\neg P_{n,q}[i][SPTX] \wedge \neg(P_{n,q}[i][KPTPA] = KC))$ ? $P_{n,q}[i][SPTPA] := P_{n,q}[i][KPTPA]$
```

The two checks ensure that: (i) executable SPT entries are not changed by Secure_Sync, eliminating the Approved Page Remapping attack, and (ii) KPT entries pointing to kernel code are never copied over to the SPT by Secure_Sync, eliminating the Writable Virtual Alias attack. Murφ is no longer able to find any attacks that violate φ_{exec} or φ_{code} in the fixed SecVisor program. Note that the initial condition, φ_{init}, is expressible in USF, and the negations of φ_{exec} and φ_{code} are expressible in GSF. Therefore, by Theorem 1, we know that the fixed SecVisor satisfies both φ_{exec} and φ_{code} for SPTs and KPTs of arbitrary size.

We added the two checks shown above to SecVisor's SPT synchronization procedure by modifying 105 lines of C code. We checked that our exploits failed against the patched version of SecVisor. More details on the fixes to SecVisor's implementation including pseudo-code are available in a companion technical report [34].

Strength of Our Approach

To highlight the power of our approach, we used Murφ to verify the correct SecVisor with an increasing number of KPT and SPT entries. We used a 3.20GHz Pentium 4 machine with 2GB of memory. Each verification includes three pages of physical memory representing, respectively, kernel code, kernel data, and user memory.

As shown in Table 3.9, we initially included three entries in both the KPT and the SPT. Murφ reported a successful verification after searching over 55,000 states, firing over 2 million rules, and running for 2.5 seconds, with a maximum memory utilization of less than 8MB. We increased the number of entries in both the KPT and the SPT to four and repeated the verification. Murφ successfully verified the new model after searching over 1.7 million states, firing more than 88 million rules, and running around 6 minutes, with a maximum memory utilization of less than 256MB. Since the adversary arbitrarily modifies every bit of a page table entry (about 4 bits), and we add two additional pages, we expected the resulting model to be between 9 and 2^7 times larger. The actual observed explosion was 2^5 times in terms of explored states and required memory.

The verification of a model with five KPT and SPT entries exceeded available memory. Given the observed state space explosion, we estimate that this verification would require about 8GB of memory. Verifying a realistic model with 256MB of paged memory (2^{16} 4KB pages) would require multiple terabytes of memory to represent the state space explicitly. More importantly, successful verification of such a model would not demonstrate the correctness of larger models. In contrast, our SMTs enable us to handle models of unbounded size.

3.4.2 sHype Security Architecture

Next, we explore the expressiveness of PGCL and PTSL by analyzing the Chinese Wall Policy (CWP) [15] as implemented in sHype hypervisor security architecture [68]. sHype is a mandatory-access-control-based system implemented in the Xen hypervisor [9], the research hypervisor rHype [64], and the PHYP commercial hypervisor [75].

Chinese Wall Policy

Access control policies distinguish between two primary types: principals and objects. The Chinese Wall Policy aims to prevent conflicts of interest when a principal can access objects owned by competing parties. It is both a confidentiality and integrity policy since it governs all accesses (e.g., reads and writes). It can be viewed as a two-level separation policy as it partitions resources based on their membership in sets that are contained in other sets.

In sHype's CWP implementation, principals are virtual machine (VM) instances, and objects are abstract workloads, represented concretely by sets of executable programs and their associated input and output data. Workloads are grouped into Chinese Wall types (CW-types), and CW-types are grouped further into Conflict of Interest (CoI) classes. The ability of a VM to gain access to a new workload is constrained by workloads it already has access to. Specifically, a VM may access workloads of, at most, one CW-type for each CoI class, with its first workload access being arbitrary. We now formalize the sHype CWP security property.

Definition 8. *Formally, a sHype CWP is a five tuple* $(WL, CWTypes, CoIClasses, TypeMap, ClassMap)$ *where: (i)* $WL = \{w_1, \ldots, w_L\}$ *is a finite set of* L *workloads, (ii)* $CWTypes = \{cwt_1, \ldots, cwt_T\}$ *is a finite set of* T *CWTypes, (iii)* $CoIClasses = \{coi_1, \ldots, coi_C\}$ *is a finite set of* C *Conflict of Interest classes, (iv)*

TypeMap maps WL to CWTypes, and (v) ClassMap maps WL to CoIClasses.

Encoding sHype CWP in PGCL

Each row of the parameterized array `P` represents a VM, and each column of `P` represents a workload. Thus, $\texttt{P}[i][j] = \top$ iff the i-th virtual machine vm_i has access to the j-th workload w_j.

The CWP confidentiality (read) policy in sHype says that a VM *vm* may access a workload w iff for all other workloads w' that *vm* can access already, either $TypeMap(w') = TypeMap(w)$, or $ClassMap(w') \neq ClassMap(w)$. Moreover, sHype's CWP write policy is equivalent to its read policy. Hence, we can combine the two policies, and express the combination as a safety property as follows:

Definition 9 (sHype CWP Access Property)**.**

$$\forall \texttt{i} \centerdot \bigwedge_{w \in WL} (\texttt{P}[\texttt{i}][w] \Rightarrow (\phi_1(\texttt{i}, w) \vee \phi_2(\texttt{i}, w)))\ , where$$

$$\phi_1(\texttt{i}, w) \equiv \bigvee_{w' \in WL \setminus \{w\}} \texttt{P}[\texttt{i}][w'] \wedge (TypeMap(w') = TypeMap(w))$$

$$\phi_2(\texttt{i}, w) \equiv \bigwedge_{w' \in WL \setminus \{w\}} \texttt{P}[\texttt{i}][w'] \Rightarrow (ClassMap(w') \neq ClassMap(w))$$

Note that WL, *CWTypes* and *CoIClasses* are all finite sets, and *TypeMap* and *ClassMap* have finite domains and ranges. Therefore, sHype's CWP policy is expressible as a USF formula, and its negation as a ESF formula.

Initial State. A system implementing CWP starts in an initial state when no previous accesses have occurred. This is expressed by the following USF formula:

$$Init \triangleq \forall \texttt{i} \centerdot \bigwedge_{w \in WL} \neg \texttt{P}[\texttt{i}][w]$$

Reference Monitor

We compile an arbitrary CWP policy $P = (WL, CWTypes, CoIClasses, TypeMap, ClassMap)$ into a reference monitor that enforces the CWP access property for P by restricting VM accesses to workloads. Specifically, for each $w \in WL$, let $g(i,w)$ be the guard that allows the i-th VM access to workload w under the CWP Access Property. For example, consider the following policy:

$$P = (WL = \{w_1, w_2, w_3, w_4, w_5\},$$

$$CWTypes = \{BankA, BankB, TechA, TechB\},$$

$$CoIClasses = \{Banks, TechCompanies\},$$

$$TypeMap(w_1) = BankA, TypeMap(w_2) = BankA,$$

$$TypeMap(w_3) = BankB, TypeMap(w_4) = TechA,$$

$$TypeMap(w_5) = TechB, ClassMap(w_1) = Banks,$$

$$ClassMap(w_2) = Banks, ClassMap(w_3) = Banks,$$

$$ClassMap(w_4) = TechCompanies,$$

$$ClassMap(w_5) = TechCompanies.)$$

Following Definition 9:

$$g(i, w_1) \triangleq \phi_1(i, w_1) \vee \phi_2(i, w_1) \Leftrightarrow$$

$$(\mathtt{P}[i][w_2] \vee (\neg \mathtt{P}[i][w_2] \wedge \neg \mathtt{P}[i][w_3])) \Leftrightarrow (\mathtt{P}[i][w_2] \vee \neg \mathtt{P}[i][w_3])$$

$$g(i, w_3) \triangleq (\neg \mathtt{P}[i][w_1] \wedge \neg \mathtt{P}[i][w_2])$$

The other guards are defined analogously. Let there be a variable hypercall such that the monitor runs whenever hypercall $= \top$. Moreover, for each $w \in WL$, the column ACCESS(w) is used to request access to w. Then, the following PGCL guarded command implements the CWP policy:

```
access_ref_monitor $\equiv$
  hypercall ?
  hypercall $:= \bot$;
  for i : $P_{n,q}$ do
    $P_{n,q}[i][ACCESS(w_1)] \wedge g(i, w_1)$ ? $P_{n,q}[i][w_1] := \top$;
            $\vdots$
    $P_{n,q}[i][ACCESS(w_L)] \wedge g(i, w_L)$ ? $P_{n,q}[i][w_L] := \top$;
```

The attacker attempts any sequence of accesses of any workloads from any VM. It is expressed by the following guarded command:

```
CWP_Adv $\equiv$
  $\top$ ?
  hypercall $:= *$;
  for i : $P_{n,q}$ do
    $\top$ ? $P_{n,q}[i][ACCESS(w_1)] := *$;
            $\vdots$
    $\top$ ? $P_{n,q}[i][ACCESS(w_L)] := *$
```

Finally, we define the overall sHype system:

$$\texttt{sHype\_CWP} \equiv \texttt{access\_ref\_monitor} \parallel \texttt{CWP\_Adv}$$

sHype is expressible as a PGCL program, its initial state is expressible in USF, and the negation of the sHype CWP access property is expressible in GSF. Therefore, Theorem 1 applies and we need only verify the system with one VM (i.e., a system parameter of one).

We used the Murφ model checker to verify the CWP access property of our sHype model. As before, verifications were run on a 3.20GHz Pentium 4 machine with 2GB of memory. Our initial verification checked a model with one VM, corresponding to a single row in the parameterized array. No counterexamples were found after searching 230 states, firing 1,325 rules, and running for 0.10s while utilizing less than 1MB of memory. By applying Theorem 1, this successful verification extends to any finite number of VMs.

Although no additional verification is necessary to demonstrate the absence of counterexamples, we explored scaling trends and demonstrated the utility of our technique by increasing the number of virtual machines instances. With two virtual machines instances, the verification completed in less than one second after exploring 34,000 states, firing over 370,000 rules, and using close to 8MB of memory. With three virtual machines, the verification required more than one hour and twenty minutes and explored greater than 5,600,000 states, firing over 89,000,000 rules and utilizing almost 1GB of memory. Subsequent verifications with larger numbers of virtual machines exceeded the memory capacity of the machine.

3.5 Expressiveness and Limitations

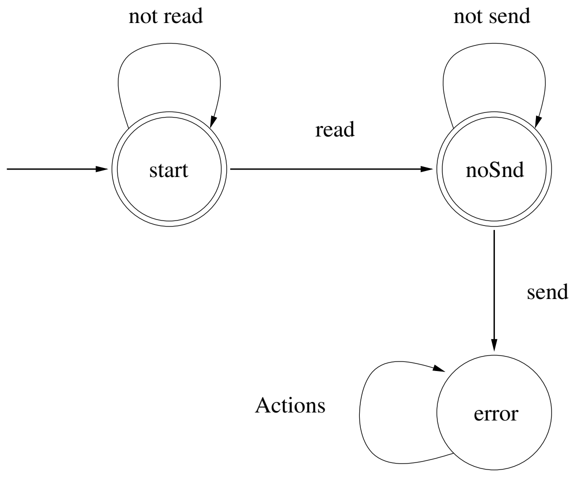
We demonstrate that our approach is expressive enough to model and analyze any "parameterized" reference monitor and policy that is expressible as finite state automata (FSA). We say that a reference monitor is parameterized if it operates over the rows of a parameterized data structure in a "row-independent" and "row-uniform" manner. Specifically, row-uniform means that the same policy is enforced on each row, while row-independent means that the policy does not refer to or depend on the content of any other row.

A FSA is defined as a five-tuple $FSA = (States, Init, Actions, T, Accept)$ where: (i) $States$ is a finite set of states of size S, (ii) $Init \subseteq States$ is the set of initial states, (iii) $Actions$ is a finite set of actions with size A, (iv) $T \subseteq States \times Action \times States$ is the transition relation, and (v) $Accept \subseteq States$ is the set of accepting states. Note that FSA are in general non-deterministic. While this does not provide additional expressive power, it enables us to represent policies compactly.

Consider the policy from Example 2 that restricts message transmission after a principal accesses sensitive data. When parameterized over processes, this policy can be viewed as the following FSA (both states are accepting, indicated by the double circle). In other words, a process respects the policy as long as its behavior is a string of actions accepted by the FSA.

Implementing a reference monitor in PGCL that enforces the policy represented by $FSA = (States, Init, Actions, T, Accept)$ is straightforward, and involves the following steps:

- Encode the finite but unbounded aspect of the policy (i.e., VMs, processes, memory pages, etc...) as the rows of P.
- Each state $\sigma_i \in States$ is encoded by two columns, σ_i and σ_i', which represent the



current and next states of the FSA respectively. We need the σ_i' columns to simulate FSA since FSA is non-deterministic, and could end up in multiple possible states after a sequence of actions.

- Each $a_i \in \mathit{Actions}$ is encoded as a column of `P`. The action columns represent the action performed by the system.
- A formula Init constrains each FSA to start in an initial state. Specifically:

$$\mathsf{Init} \triangleq \forall \texttt{i}.\ \bigwedge_{s \in \mathit{Init}} \texttt{P[i]}[s] \wedge \bigwedge_{s \in \mathit{States} \setminus \mathit{Init}} \neg \texttt{P[i]}[s]$$

- The transition relation T is encoded as a finite number of Boolean variables of the form:

$$\forall \sigma_i, \sigma_k \in \mathit{States}, a_j \in \mathit{Action}\,.\, b_{\sigma_i, a_j, \sigma_k} \Leftrightarrow T(\sigma_i, a_j, \sigma_k)$$

- Then a general reference monitor that enforces the policy represented by FSA is a guarded command that loops over the rows of `P`, considers the action performed by the system by inspecting the action columns, and updates each row in three steps:

1. Sets all σ'_k columns to $\perp$.
2. Sets appropriate σ'_k columns to $\top$ based on the σ_i and a_j columns, and $b_{\sigma_i,a_j,\sigma_k}$.
3. Copies σ'_k columns into the σ_i columns.

Note that this essentially simulates the execution of FSA from the states encoded by the σ_i columns upon seeing the actions encoded by the a_j columns. The reference monitor is described by the following PGCL guarded command.

```
**universal_reference_monitor** $\equiv$
  $\top$ ?
  for i : $P_{n,q}$ do
    $P_{n,q}[i][\sigma'_1] := \perp; \ldots ; P_{n,q}[i][\sigma'_S] := \perp$;
  for i : $P_{n,q}$ do
    $P_{n,q}[i][\sigma_1] \wedge P_{n,q}[i][a_1] \wedge b_{\sigma_1,a_1,\sigma_1}$ ? $P_{n,q}[i][\sigma'_1] := \top$;
              $\vdots$
  for i : $P_{n,q}$ do
    $P_{n,q}[i][\sigma_S] \wedge P_{n,q}[i][a_A] \wedge b_{\sigma_S,a_A,\sigma_S}$ ? $P_{n,q}[i][\sigma'_S] := \top$;
  for i : $P_{n,q}$ do
    $P_{n,q}[i][\sigma_1] := P_{n,q}[i][\sigma'_1]; \ldots ; P_{n,q}[i][\sigma_S] := P_{n,q}[i][\sigma'_S]$;
```

The following guarded command implements an adversary that non-deterministically selects a sequence of input actions (via the action columns of P) to the reference monitors. Clearly, this is the strongest adversary that is constrained to input actions alone.

```
**universal_adv** $\equiv$
  $\top$ ? for i : $P_{n,q}$ do
    $\top$ ? $P_{n,q}[i][a_1] := *; \ldots P_{n,q}[i][a_A] := *$;
```

We model the FSA policy as a formula in PTSL. Our specification logic admits parameterized *row formulas* of the form $\forall i \in \mathbb{N} \,.\, \varphi(i)$ where the index i denotes the i-th formula and it refers only to the variables in the i-th row of P. Given this form, we can encode the security policy represented by the parametric reference monitor as a row formula.

Finally, we can employ a model checker to determine if the reference monitor running in parallel with the adversary implementation satisfies the security property, equivalently:

$$\textbf{universal\_reference\_monitor} \parallel \textbf{universal\_adv} \models \varphi$$

The restrictions of row-uniform and row-independent behavior are required to express parameterized reference monitors in PGCL. These restrictions are a limitation of this work. There exist important cases of reference monitor policies, such as type enforcement [14], that are not row-independent and row-uniform. In general, safety analysis for access-control-based systems in the style of the HRU model [43] allows for the possibility of enforcing richer policies that are not expressible with our restrictions.

3.6 Conclusion

The reference monitors in operating systems, hypervisors, and web browsers must correctly enforce their desired security policies in the presence of adversaries. Despite progress in developing reference monitors with small code sizes, a significant remaining factor in the complexity of automatically verifying reference monitors is the size of the data structures over which they operate. We developed a verification technique that scales even when reference monitors and adversaries operate over unbounded, but finite data structures. Our technique significantly reduces the cost and improves the practicality of automated formal verification for reference monitors. We developed a parametric guarded command language for modeling reference monitors and adversaries, and a parametric temporal specification logic for expressing security policies that the monitor is expected to enforce. The central technical results of this chapter are a set of small model theorems that state that in order to verify that a policy is enforced for a reference monitor with an arbitrarily large data structure, it is sufficient to model check the monitor with just one entry in its data structure. We applied this methodology to verify that the designs of two hypervisors – SecVisor and the sHype mandatory-access-control extension to Xen – correctly enforce the expected security properties in the presence of adversaries.

Next, we extend PGCL and PTSL to include security properties and programs that allow relations between rows of the parameterized array. These extensions enable modeling and analysis of reference monitors that implement more expressive access control policies. Such policies include those with relationships (e.g., ownership, sharing, and communication) between principals. In addition, this extension enables us to apply our results to verify security properties of reference monitor implementations.

Chapter 4

Parametric Verification with Hierarchical Data Structures

4.1 Introduction

A common use of protection mechanisms in systems software is to prevent one execution context from accessing memory regions allocated to a different context. For example, hypervisors, such as Xen [9], are designed to support memory separation not only among guest operating systems, but also between the guests and the hypervisor itself. Separation is achieved by an address translation subsystem that is self-contained and relatively small (around 7000 LOC in Xen version 3.0.3). Verifying security properties of such separation mechanisms is both: (i) important, due to their wide deployment in environments with malicious guests, e.g., the cloud; and (ii) challenging, due to their complexity. Addressing this challenge is the subject of this chapter.

A careful examination of the source code for two hypervisors – Xen and ShadowVisor (a research hypervisor) – reveals that a major source of complexity in separation mechanisms is the size, and hierarchical nesting, of the data-structures over which they operate. For example, Xen's address translation mechanism involves multi-level page tables where a level has up to 512 entries in a 3-level implementation, or up to 1024 entries in a 2-level implementation. The number of levels is further increased by optimizations, such as context caching (see Section 4.2 for a detailed description). Since the complexity of model checking grows exponentially with the size of these data-structures, verifying these separation mechanisms directly is intractable.

We address this problem by developing a parametric verification technique that is able to handle separation mechanisms operating over multi-level data structures of *arbitrary size* and with *arbitrary number of levels*. Specifically, we make the following contributions. First, we develop an extended parametric guarded command language ($PGCL^+$) for modeling hypervisors and adversaries. In particular, $PGCL^+$ supports: (i) nested parametric arrays to model data structures, such as multi-level page tables, where the parameters model the size of page tables at each level; and (ii) whole array operations to model an (indeed, the most powerful possible) adversary which non-deterministically sets the values of data structures under its control.

In addition, the design of $PGCL^+$ is driven by the fact that our target separation mechanisms operate over tree-shaped data structures in a *row independent* and *hierarchically row uniform* manner. Consider a mechanism operating over a tree-shaped multi-level page table. Row independence means that the values in different rows of a page table are mutually independent. Hierarchical row uniformity implies that: (a) for each level i of the page table, the mechanism executes the same command on all rows at level i; (b) the command for a row at level i involves recursive operation over at most one page table at the next level $i+1$; (c) the commands for distinct rows at level i never lead to operations over the same table at level $i+1$. Both row independence and hierarchical uniformity are baked syntactically into $PGCL^+$ via restricted forms of commands and nested whole array operations.

Second, we develop a parametric specification formalism, a strict extension of PTSL referred to by the same name, for expressing security policies of separation mechanisms modeled in *PGCL*$^+$. Our formalism is able to express both safety and liveness properties that involve arbitrary nesting of quantifiers over multiple levels of the nested parametric arrays in *PGCL*$^+$.

Third, we prove a set of *small model theorems* that roughly state that for any system M expressible in *PGCL*$^+$, and any security property φ in our extended specification formalism, an instance of M with a data structure of arbitrary size satisfies φ iff the instance of M where the data structure has 1 element at every level satisfies φ. Note that these theorems apply to the extended language *PGCL*$^+$ and extended specification logic PTSL and hence are strictly more general than those developed in Chapter 3. These theorems yield the best possible reduction – e.g., verifying security of a separation mechanism over an arbitrarily large page table is reduced to verifying the mechanism with just 1 page table entry at each level. This ameliorates the lack of scalability of verification due to data structure size.

Finally, we demonstrate the effectiveness of our approach by modeling, and verifying, shadow paging mechanisms of Xen version 3.0.3 and ShadowVisor, together with associated address separation properties. The models were created manually from the actual source code of these systems. In the case of ShadowVisor, our initial verification identified a previously unknown vulnerability. After fixing the vulnerability, we are able to verify the new model successfully.

The rest of this chapter is organized as follows. Section 4.2 presents an overview of address translation mechanisms and associated separation properties. Section 4.3 presents the parametric modeling language, the specification logic, as well as the small model theorems and the key ideas behind their proofs. Section 4.4 presents the case studies. Finally, Section 4.5 presents our conclusions.

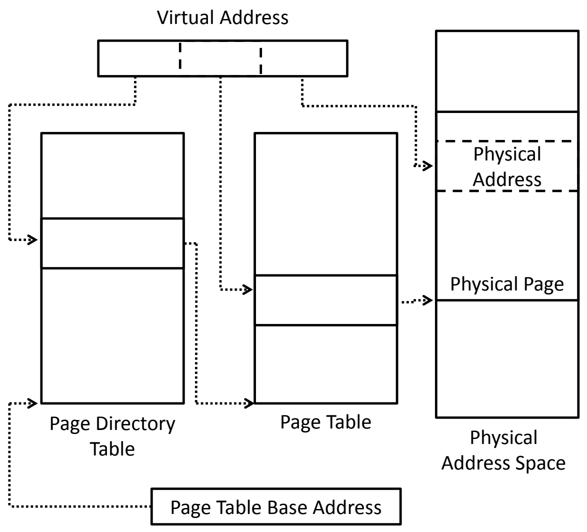


Figure 4.1: Typical two-level page table structure.

4.2 Address Space Translation and Separation

In this section, we give an overview of the systems we target, viz., address space translation schemes, and the properties we verify, viz., address separation.

Translation. Consider a system with memory sub-divided into pages. Each page has a base address (or address, for short). *Address space translation* maps source addresses to destination addresses. In the simplest setting, it is implemented by a single-level page table (PT). Each row of the PT is a pair (x, y) such that x is a source base address and y is its corresponding destination base address.

Multi-level Paging. More sophisticated address translation schemes use multi-level PTs. Figure 4.1 shows a typical two-level address translation. A 2-level PT consists of a top level Page Directory Table (PDT) and a set of leaf PTs. An n-level PT is essentially a set of tables linked to form a tree of depth n. Specifically, each row of a table at level i contains either a destination address, or the starting address of a table at level $i+1$. In addition to addresses, rows contain flags (e.g., to indicate if the row contains a destination addresses or the address of another table). We now present a concrete example.

Example 3. *To perform a translation between address spaces, a source address i is split into two parts, whose sizes are determined during the design of the PT. Let $i = (i_1, i_2)$. To compute the destination address corresponding to i, we first find the row (i_1, o_1) in the PDT where o_1 is the entry at index i_1. The entry o_1 contains an address a_1, a Page Size Extension flag PSE, and a present flag $PRESENT$. If $PRESENT$ is unset, then there is no destination address corresponding to i. Otherwise, if PSE is set, then the destination address is a_1. Finally, if PSE is unset, we find the entry (i_2, a_2) in the table located at address a_1, and return a_2 as the destination address. Note the use of PSE and PRESENT to disambiguate between different types of rows. Also, note the dual use of the address field a_1 as either a destination address or a table address.*

Address Separation. While the systems we target are address translation schemes, the broad class of properties we aim for is address separation. This is a crucial property – in essence, requiring that disjoint source address spaces be mapped to disjoint destination address spaces. Our notion of address separation is conceptually similar to that used by Baumann et al. [12]. Formally, an address translation scheme M violates separation if it maps addresses a_1 and a_2 from two different source address spaces to the same destination address. For example, an OS's virtual memory manager enforces separation between the address spaces of the OS kernel and various processes. Address space separation is a safety property since its violation is exhibited by a finite execution.

Shadow Paging. The key technique, used by hypervisor address translation schemes involving page tables, to ensure memory separation is shadowing. For example, a separation kernel employs shadow paging to isolate critical memory regions from an untrusted guest OS. In essence, the kernel maintains its own trusted version of the guest's PT, called the shadow PT or sPT. The guest is allowed to modify its PT. However, the kernel interposes on such modifications and checks that the guest's modifications do not violate memory separation. If the check succeeds, the sPT is synchronized with the modified guest's PT.

Uses of multi-level PTs. Multi-level PTs are the canonical tree-shaped data-structures that motivates our work in this chapter. In real systems, such PTs are used for various optimizations. One use is to translate large source address spaces without the overhead of one PT entry for each source base address. Another use is to implement context caching, a performance optimization – used by both Xen and VMWare – for shadow paging. Normally, every virtual address space (or context) has its own PT, e.g., for a hypervisor, each process running on each guest OS has a separate context. Suppose that all context PTs are shadowed to a single sPT. When the context changes (e.g., when a new process is scheduled to run), the sPT is re-initialized from the PT of the new context. This hampers performance. Context caching avoids this problem by shadowing each context PT to a separate sPT. In essence, the sPT itself becomes a multi-level PT, where each row of the top-level PT points to a PT shadowing a distinct context.

Our goal is to verify address separation for address translation schemes that operate on multi-level PTs with arbitrary (but fixed) number of levels and arbitrary (but fixed) number of rows in each table, where each row has an arbitrary (but fixed) number of flags. These goals crucially influence the syntax and semantics of *PGCL*$^+$ and our specification formalism, and our technical results, which we present next.

4.3 Definitions of *PGCL*$^+$ and PTSL

In this section, we present our extended language *PGCL*$^+$ and our extended specification formalism for modeling programs and security properties, respectively.

4.3.1 *PGCL*$^+$ Syntax

All variables in *PGCL*$^+$ are Boolean. In contrast to PGCL which includes a single array, *PGCL*$^+$ includes *nested parametric arrays* to a finite depth d. Each row of an array at depth d is a record with a single field $\texttt{F}$, a finite array of Booleans of size q_d. Each row of an array at depth z ($1 \leq z < d$) is a structure with two fields: $\texttt{F}$, a finite array of Booleans of size q_z, and $\texttt{P}$ an array at depth $z+1$. Our results do not depend on the values of d and $\{q_z \mid 1 \leq z \leq d\}$, and hence hold for programs that manipulate arrays that are nested (as describe above) to arbitrary depth, and with Boolean arrays of arbitrary size at each level. Also, Boolean variables enable us to encode finite valued variables, and arrays, records, relations and functions over such variables.

Let 1 and 0 be, respectively, the representations of the truth values **true** and **false**. Let $\texttt{B}$ be a set of Boolean variables, $\texttt{i}_1, \dots, \texttt{i}_d$ be variables used to index into $\texttt{P}_1, \dots, \texttt{P}_d$, respectively, and $\texttt{n}_1, \dots, \texttt{n}_d$ be variables used to store the number of rows of $\texttt{P}_1, \dots, \texttt{P}_d$, respectively. The syntax of *PGCL*$^+$ is shown in Figure 4.2. *PGCL*$^+$ supports natural numbers, Boolean variables, propositional expressions over Boolean variables and $\texttt{F}$ elements, guarded commands that update Boolean variables and $\texttt{F}$ elements, and parallel composition of guarded commands. A `skip` command does nothing. A guarded command $\texttt{e} ~?~ \texttt{c}_1 : \texttt{c}_2$ executes $\texttt{c}_1$ or $\texttt{c}_2$ depending on whether $\texttt{e}$ evaluates to true or false. We write $\texttt{e} ~?~ \texttt{c}$ to mean $\texttt{e} ~?~ \texttt{c} : \texttt{skip}$. The parallel composition of two guarded commands executes by non-deterministically picking one of the commands to execute. The sequential composition of two commands executes the first command followed by the second command.

Natural Numerals	$\mathtt{K}$			
Boolean Variables	$\mathtt{B}$			
Parametric Index Variables	$\mathtt{i}_1,\ldots,\mathtt{i}_d$			
Parameter Variables	$\mathtt{n}_1,\ldots,\mathtt{n}_d$			
Expressions	$\mathtt{E}$	::=	$\mathtt{1} \mid \mathtt{0} \mid * \mid \mathtt{B} \mid \mathtt{E} \vee \mathtt{E} \mid \mathtt{E} \wedge \mathtt{E} \mid \neg \mathtt{E}$	
Param. Expressions ($1 \leq z \leq d$)	$\widehat{\mathtt{E}}_z$	::=	$\mathtt{E} \mid \mathtt{P}[\mathtt{i}_1]\ldots\mathtt{P}[\mathtt{i}_z].\mathtt{F}[\mathtt{K}] \mid \widehat{\mathtt{E}}_z \vee \widehat{\mathtt{E}}_z \mid \widehat{\mathtt{E}}_z \wedge \widehat{\mathtt{E}}_z \mid \neg \widehat{\mathtt{E}}_z$	
Instantiated Guarded Commands	$\mathtt{G}$	::=	$\mathtt{GC}(\mathtt{K}^d)$	
Guarded Commands	$\mathtt{GC}$	::=	$\mathtt{E}\ ?\ \mathtt{C}_1 : \mathtt{C}_1$	
		$\mid$	$\mathtt{GC} \parallel \mathtt{GC}$	Parallel comp.
Commands (depth $1 \leq z \leq d$)	$\mathtt{C}_z$	::=	$\mathtt{B} := \mathtt{E}$ (if $z = 1$)	Assignment
		$\mid$	$\mathtt{for}\ \mathtt{i}_z\ \mathtt{do}\ \widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z : \widehat{\mathtt{C}}_z$	Parametric for
		$\mid$	$\mathtt{C}_z ; \mathtt{C}_z$	Sequencing
		$\mid$	$\mathtt{skip}$	Skip
Param. Commands ($1 \leq z \leq d$)	$\widehat{\mathtt{C}}_z$	::=	$\mathtt{P}[\mathtt{i}_1]\ldots\mathtt{P}[\mathtt{i}_z].\mathtt{F}[\mathtt{K}] := \widehat{\mathtt{E}}_z$	Array assign
		$\mid$	$\widehat{\mathtt{C}}_z ; \widehat{\mathtt{C}}_z$	Sequencing
		$\mid$	$\mathtt{C}_{z+1}$ (if $z < d$)	**Nesting**

Figure 4.2: *PGCL*$^+$ Syntax. Note the addition of nesting where z denotes depth.

Note that commands at depth $z+1$ are nested within those at depth z.

Language Design. Values assigned to an element of an F array at depth z can depend only on: (i) other elements of the same F array; (ii) elements of parent F arrays along the nesting hierarchy (to ensure hierarchical row uniformity); and (iii) Boolean variables. Values assigned to Boolean variables depend on other Boolean variables only. This is crucial to ensure row-independence which is necessary for our small model theorems (cf. Sec. 4.3.5).

4.3.2 ShadowVisor Code in *PGCL*$^+$

We use ShadowVisor as a running example, and now describe its model in *PGCL*$^+$. ShadowVisor uses a 2-level PT scheme. The key unbounded data structures are the guest and shadow Page Directory Table (gPDT and sPDT) at the top level, and the guest and shadow Page Tables (gPTs and sPTs) at the lower level. Since each shadow table has the same size as the corresponding guest table, we model them together in the 2-level *PGCL*$^+$ parametric array.

For simplicity, let PDT be the top-level array P. Elements PDT[i_1].F[gPRESENT] and PDT[i_1].F[gPSE] are the present and page size extension flags for the i_1-th gPD entry, while PDT[i_1].F[gADDR] is the destination address contained in the i_1-th gPD entry. Elements sPRESENT, sPSE, and sADDR are defined analogously for sPD entries. Again for simplicity, let PDT[i_1].PT be the array P[i_1].P. Elements gPTE_PRESENT and gPTE_ADDR of PDT[i_1].PT[i_2].F are the present flag and destination address contained in the i_2-th entry of the PT pointed to by the i_1-th gPDT entry. Elements sPTE_PRESENT and sPTE_ADDR of PDT[i_1].PT[i_2].F are similarly defined for the sPDT. Terms gPDE refers to the set of elements corresponding to a gPDT entry (i.e., gPRESENT, gPSE, and gADDR). Terms gPTE, sPDE and sPTE are defined similarly for the gPT, sPDT, and sPT, respectively.

Our ShadowVisor model (see Figure 4.3) is a parallel composition of four guarded commands shadow_page_fault, shadow_invalidate_page, shadow_new_context, and adversary. Command shadow_page_fault synchronizes sPDT and sPT with gPDT and gPT when the guest kernel: (i) loads a new gPT, or (ii) modifies or creates a gPT entry. To ensure separation, shadow_page_fault only copies destination addresses from the gPT or gPDT that are less than MEM_LIMIT minus an offset that depends on the page size. This results in two distinct checks depending on the level of the table since pages mapped in the PDT are of size MPS_PDT and pages mapped in the PT are of size MPS_PT.

Command shadow_invalidate_page invalidates entries in the sPD and sPT (by setting to zero) when the corresponding guest entries are not present, the *PSE* bits are inconsistent, or the if both structures are consistent and the guest OS invalidates a page. Command shadow_new_context initializes a new context by clearing all the entries of the sPD. Finally, command adversary models the attacker by arbitrarily modifying every gPD entry and every gPT entry.

```
shadow_page_fault ≡
  for $i_1$ do
    $PDT[i_1].F[gPRESENT] \wedge PDT[i_1].F[gPSE] \wedge$
    $PDT[i_1].F[gADDR] < MEM\_LIMIT - MPS\_PDT$ ?
      $PDT[i_1].F[sPDE] := PDT[i_1].F[gPDE]$;
    for $i_2$ do
      $PDT[i_1].F[gPRESENT] \wedge$
      $PDT[i_1].PT[i_2].F[gPTE\_PRESENT] \wedge$
      $PDT[i_1].PT[i_2].F[gPTE\_ADDR] < MEM\_LIMIT - MPS\_PT$ ?
        $PDT[i_1].PT[i_2].F[sPTE] := PDT[i_1].PT[i_2].F[gPTE]$;

shadow_invalidate_page ≡
  for $i_1$ do
    $(PDT[i_1].F[sPRESENT] \wedge \neg PDT[i_1].F[gPRESENT]) \vee$
    $(PDT[i_1].F[sPRESENT] \wedge PDT[i_1].F[gPRESENT] \wedge$
    $(PDT[i_1].F[sPSE] \vee PDT[i_1].F[gPSE]))$ ?
      $PDT[i_1].F[sPDE] := 0$;
  for $i_1$ do
    $PDT[i_1].F[sPRESENT] \wedge PDT[i_1].F[gPRESENT] \wedge$
    $\neg PDT[i_1].F[gPSE] \wedge \neg PDT[i_1].F[sPSE]$ ?
      for $i_2$ do
        $PDT[i_1].PT[i_2].F[sPTE] := 0$;

shadow_new_context ≡
  for $i_1$ do
    $PDT[i_1].F[sPDE] := 0$;

adversary ≡
  for $i_1$ do
    $PDT[i_1].F[gPDE] := *$;
    for $i_2$ do
      $PDT[i_1].PT[i_2].F[gPTE] := *$;
```

Figure 4.3: ShadowVisor model in *PGCL*$^+$.

For brevity, we write `c` to mean `1 ? c`. Since all PGCL variables are Boolean, we write $\texttt{x} < \texttt{C}$ to mean the binary comparison between a finite valued variable `x` and a constant `C`.

4.3.3 *PGCL*$^+$ Semantics

We now present the operational semantics of *PGCL*$^+$ as a relation on stores. The semantics of *PGCL*$^+$ closely parallel the semantics of PGCL. However, the addition of nested parametric arrays complicates the definitions of stores and store projection.

Let $\mathbb{B}$ be the truth values $\{\textbf{true}, \textbf{false}\}$. Let $\mathbb{N}$ denote the set of natural numbers. For two natural numbers j and k such that $j \leq k$, we write $[j,k]$ to mean the set of numbers in the closed range from j to k. For any numeral k we write $\lceil k \rceil$ to mean the natural number represented by k in standard arithmetic. Often, we write k to mean $\lceil k \rceil$ when the context disambiguates such usage.

We write $Dom(f)$ to mean the domain of a function f; $(\mathtt{t}, \mathtt{t}')$ denotes the concatenation of tuples $\mathtt{t}$ and $\mathtt{t}'$; $\mathtt{t}_{i,j}$ is the subtuple of $\mathtt{t}$ from the i^{th} to the j^{th} elements, and $\mathtt{t}_i$ means $\mathtt{t}_{i,i}$. Given a tuple of natural numbers $\mathtt{t} = (t_1, \dots, t_z)$, we write $\otimes(\mathtt{t})$ to denote the set of tuples $[1,t_1] \times \cdots \times [1,t_z]$. Recall that, for $1 \leq z \leq d$, q_z is the size of the array $\mathtt{F}$ at depth z. Then, a store σ is a tuple $(\sigma^B, \sigma^n, \sigma^P)$ such that:

- $\sigma^B : \mathtt{B} \to \mathbb{B}$ maps Boolean variables to $\mathbb{B}$;
- $\sigma^n \in \mathbb{N}^d$ is a tuple of values of the parameter variables;
- σ^P is a tuple of functions defined as follows:

$$\forall z \in [1,d] \centerdot \sigma^P_z : \otimes(\sigma^n_{1,z}, q_z) \to \mathbb{B}$$

We omit the superscript of σ when it is clear from the context. The rules for evaluating *PGCL*$^+$ expressions under stores are defined inductively over the structure of *PGCL*$^+$ expressions, and shown in Figure 4.4. To define the semantics of *PGCL*$^+$, we first present

$$\frac{}{\langle 1,\sigma\rangle \to \textbf{true}} \qquad \frac{}{\langle 0,\sigma\rangle \to \textbf{false}} \qquad \frac{}{\langle *,\sigma\rangle \to \textbf{true}} \qquad \frac{}{\langle *,\sigma\rangle \to \textbf{false}}$$

$$\frac{\mathtt{b} \in dom(\sigma^B)}{\langle \mathtt{b},\sigma\rangle \to \sigma^B(\mathtt{b})} \qquad \frac{\langle \mathsf{e},\sigma\rangle \to t}{\langle \neg \mathsf{e},\sigma\rangle \to [\neg]t} \qquad \frac{(\lceil \mathtt{k}_1\rceil,\ldots,\lceil \mathtt{k}_z\rceil,\lceil \mathtt{r}\rceil) \in Dom(\sigma^P_z)}{\langle \mathtt{P}[\mathtt{k}_1]\ldots\mathtt{P}[\mathtt{k}_z].\mathtt{F}[\mathtt{r}],\sigma\rangle \to \sigma^P_z(\lceil \mathtt{k}_1\rceil,\ldots,\lceil \mathtt{k}_z\rceil,\lceil \mathtt{r}\rceil)}$$

$$\frac{\langle \mathsf{e},\sigma\rangle \to t \qquad \langle \mathsf{e}',\sigma\rangle \to t'}{\langle \mathsf{e} \vee \mathsf{e}',\sigma\rangle \to t[\vee]t'} \qquad \frac{\langle \mathsf{e},\sigma\rangle \to t \qquad \langle \mathsf{e}',\sigma\rangle \to t'}{\langle \mathsf{e} \wedge \mathsf{e}',\sigma\rangle \to t[\wedge]t'}$$

Figure 4.4: Rules for expression evaluation.

the notion of store projection.

We overload the $\mapsto$ operator as follows. For any function $f : X \to Y$, $x \in X$ and $y \in Y$, we write $f[x \mapsto y]$ to mean the function that is identical to f, except that x is mapped to y. $\mathsf{X}[y \mapsto w]$ is a tuple that equals X, except that $(\mathsf{X}[y \mapsto w])_y = w$. For any *PGCL*$^+$ expression or guarded command $\mathtt{X}$, variable $\mathtt{v}$, and expression $\mathtt{e}$, we write $\mathtt{X}[\mathtt{v} \mapsto \mathtt{e}]$ to mean the result of replacing all occurrences of $\mathtt{v}$ in $\mathtt{X}$ simultaneously with $\mathtt{e}$. For any $z \in \mathbb{N}$, $\mathtt{1}^z$ denotes the tuple of z $\mathtt{1}$'s.

Definition 10 (Store Projection)**.** *Let* $\sigma = (\sigma^B,\sigma^n,\sigma^P)$ *be any store and* $1 \le z \le d$*. For* $\mathsf{k} = (k_1,\ldots,k_z) \in \otimes(\sigma^n_1,\ldots,\sigma^n_z)$ *we write* $\sigma \downharpoonleft \mathsf{k}$ *to mean the store* $(\sigma^B,\sigma^m,\sigma^Q)$ *such that:*

1. $\sigma^m = \sigma^n[1 \mapsto 1][2 \mapsto 1]\ldots[z \mapsto 1]$

2. $\forall y \in [1,z] \centerdot \forall X \in Dom(\sigma^Q_y) \centerdot \sigma^Q_y(X) = \sigma^P_y(X[1 \mapsto k_1][2 \mapsto k_2]\ldots[y \mapsto k_y])$

3. $\forall y \in [z+1,d] \centerdot \forall X \in Dom(\sigma^Q_y) \centerdot \sigma^Q_y(X) = \sigma^P_y(X[1 \mapsto k_1][2 \mapsto k_2]\ldots[z \mapsto k_z])$

Note: $\forall z \in [1,d] \centerdot \forall \mathsf{k} \in \otimes(\sigma^n_1,\ldots,\sigma^n_z) \centerdot \sigma \downharpoonleft \mathsf{k} = (\sigma \downharpoonleft \mathsf{k}) \downharpoonleft \mathtt{1}^z$.

Intuitively, $\sigma \downharpoonleft \mathsf{k}$ is constructed by retaining σ^B, changing the first z elements of σ^n to 1 and leaving the remaining elements unchanged, and projecting away all but the k_y-th row

$$\frac{\sigma^n = (\lceil \mathtt{k}_1 \rceil, \ldots, \lceil \mathtt{k}_d \rceil) \qquad \{\sigma\}\ \mathtt{gc}\ \{\sigma'\}}{\{\sigma\}\ \mathtt{gc}(\mathtt{k}_1, \ldots, \mathtt{k}_d)\ \{\sigma'\}}\text{Parameter Instantiation}$$

$$\frac{\langle \mathtt{e}, \sigma \rangle \to \mathbf{true} \wedge \{\sigma\}\ \mathtt{c}_1\ \{\sigma'\} \bigvee \langle \mathtt{e}, \sigma \rangle \to \mathbf{false} \wedge \{\sigma\}\ \mathtt{c}_2\ \{\sigma'\}}{\{\sigma\}\ \mathtt{e}\ ?\ \mathtt{c}_1 : \mathtt{c}_2\ \{\sigma'\}}\text{GC}$$

$$\frac{\{\sigma\}\ \mathtt{c}\ \{\sigma''\} \qquad \{\sigma''\}\ \mathtt{c}'\ \{\sigma'\}}{\{\sigma\}\ \mathtt{c};\mathtt{c}'\ \{\sigma'\}}\text{Sequential} \qquad \frac{\{\sigma\}\ \mathtt{gc}\ \{\sigma'\} \vee \{\sigma\}\ \mathtt{gc}'\ \{\sigma'\}}{\{\sigma\}\ \mathtt{gc} \parallel \mathtt{gc}'\ \{\sigma'\}}\text{Parallel}$$

$$\frac{\langle \mathtt{e}, \sigma \rangle \to t}{\{\sigma\}\ \mathtt{b} := \mathtt{e}\ \{\sigma[\sigma^B \mapsto \sigma^B[\mathtt{b} \mapsto t]]\}}\text{Assign} \qquad \frac{}{\{\sigma\}\ \mathtt{skip}\ \{\sigma\}}\text{Skip}$$

$$\frac{\widehat{\mathtt{e}} \in \widehat{\mathtt{E}}_z \qquad \langle \widehat{\mathtt{e}}, \sigma \rangle \to t \qquad (\lceil \mathtt{k}_1 \rceil, \ldots, \lceil \mathtt{k}_z \rceil, \lceil \mathtt{r} \rceil) \in Dom(\sigma^P_z)}{\{\sigma\}\ \mathtt{P}[\mathtt{k}_1] \ldots \mathtt{P}[\mathtt{k}_z].\mathtt{F}[\mathtt{r}] := \widehat{\mathtt{e}}\ \{\sigma[\sigma^P \mapsto \sigma^P[\sigma^P_z \mapsto [\sigma^P_z[(\lceil \mathtt{k}_1 \rceil, \ldots, \lceil \mathtt{k}_z \rceil, \lceil \mathtt{r} \rceil) \mapsto t]]]]\}}\text{Param. Array Assign}$$

$$\frac{\begin{array}{c}\sigma^n_{1,z} = (1^{z-1}, N) \qquad \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2 \in (\widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z : \widehat{\mathtt{C}}_z)[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_{z-1} \mapsto 1] \\ \forall y \in [1,N] \centerdot \{\sigma \downharpoonleft (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft (1^{z-1}, y)\}\end{array}}{\{\sigma\}\ \mathtt{for}\ \mathtt{i}_z\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2\ \{\sigma'\}}\text{Unroll}$$

Figure 4.5: Rules for commands

of the parametric array at depth y for $1 \leq y \leq z$. Note that since projection retains σ^B, it does not affect the evaluation of expressions that do not refer to elements of P.

Store Transformation. For any *PGCL*$^+$ command c and stores σ and σ', we write $\{\sigma\}\ \mathtt{c}\ \{\sigma'\}$ to mean that σ is transformed to σ' by the execution of c. We define $\{\sigma\}\ \mathtt{c}\ \{\sigma'\}$ via induction on the structure of c, as shown in Figure 4.5. $[\wedge]$, $[\vee]$, and $[\neg]$ denote logical conjunction, disjunction, and negation, respectively.

The "GC" rule states that σ is transformed to σ' by executing the guarded command $\mathtt{e}\ ?\ \mathtt{c}_1 : \mathtt{c}_2$ if: (i) either the guard e evaluates to **true** under σ and σ is transformed to σ' by executing the command $\mathtt{c}_1$; (ii) or e evaluates to **false** under σ and σ is transformed to σ' by executing $\mathtt{c}_2$. Note the addition of an "else" condition to the "GC" rule in Chapter 3.

The addition of nested parametric arrays requires a new "Unroll" rule. The new "Unroll” rule states that *for an array at depth z* if $\mathtt{c}$ is a `for` loop, then $\{\sigma\}$ $\mathtt{c}$ $\{\sigma'\}$ if each row of σ' results by executing the loop body from the same row of σ. The nesting of for-loops complicates the proofs of our small model theorems. Indeed, we require to reason using mutual induction about loop bodies $(\widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z)$ and commands $(\mathtt{C}_z)$, starting with the loop bodies at the lowest level, and moving up to commands at the highest level.

4.3.4 Specification Formalism

The following specification formalism is a strict extension of PTSL as presented in Chapter 3 to allow formulas over the nested parametric arrays in *PGCL*$^+$. As before, we support both reachability properties and temporal logic specifications. The syntax of state formulas is defined in Figure 4.6. We support three types of state formulas – universal, existential, and generic. Specifically, universal formulas allow only nested universal quantification over $\mathtt{P}$, existential formulas allow arbitrary quantifier nesting with at least one $\exists$, while generic formulas allow one of each.

Temporal logic specifications are expressed in PTSL. In essence, PTSL is a subset of the temporal logic ACTL* [20] with USF as atomic propositions. The syntax of PTSL is defined in Figure 4.6. The quantification nesting allowed in our specification logic allows expressive properties spanning multiple levels of $\mathtt{P}$. Not that these properties were not expressible in the version of PTSL presented in Chapter 3. This will be crucial for our case studies, as shown in Sec. 4.4.

ShadowVisor Security Properties in PTSL. ShadowVisor begins execution with every entry of the sPDT and sPT set to not present. This initial condition is stated in the following USF state formula:

$$\varphi_{init} \triangleq \forall \mathtt{i}_1, \mathtt{i}_2 . \quad \neg \mathtt{PDT}[\mathtt{i}_1].\mathtt{F}[\mathtt{sPRESENT}] \wedge \neg \mathtt{PDT}[\mathtt{i}_1].\mathtt{PT}[\mathtt{i}_2].\mathtt{F}[\mathtt{sPTE\_PRESENT}]$$

Basic Propositions	BP	$::=$	$\mathtt{b}\ ,\ \mathtt{b} \in B \mid \neg\mathsf{BP} \mid \mathsf{BP} \wedge \mathsf{BP}$
Parametric Propositions	$\mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z)$	$::=$	$\{\mathtt{P}[\mathtt{i}_1]\ldots\mathtt{P}[\mathtt{i}_z].\mathtt{F}[\mathtt{r}] \mid \lceil\mathtt{r}\rceil \leq q_z\}$
			$\mid \neg\mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z)$
			$\mid \mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z) \wedge \mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z)$
Universal State Formulas	USF	$::=$	BP
			$\mid \forall\mathtt{i}_1\ldots\forall\mathtt{i}_z.\mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z)$
			$\mid \mathsf{BP} \wedge \forall\mathtt{i}_1\ldots\forall\mathtt{i}_z.\mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z)$
Existential State Formulas	ESF	$::=$	BP
			$\mid Æ_1\mathtt{i}_1\ldots Æ_z\mathtt{i}_z.\mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z)$
			$\mid \mathsf{BP} \wedge Æ_1\mathtt{i}_1\ldots Æ_z\mathtt{i}_z.\mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z)$
Generic State Formulas	GSF	$::=$	$\mathsf{USF} \mid \mathsf{ESF} \mid \mathsf{USF} \wedge \mathsf{ESF}$
PTSL Path Formulas	TLPF	$::=$	$\mathsf{TLF} \mid \mathsf{TLF} \wedge \mathsf{TLF} \mid \mathsf{TLF} \vee \mathsf{TLF}$
			$\mid \mathbf{X}\ \mathsf{TLF} \mid \mathsf{TLF}\ \mathbf{U}\ \mathsf{TLF}$
PTSL Formulas	TLF	$::=$	$\mathsf{USF} \mid \neg\mathsf{USF} \mid \mathsf{TLF} \wedge \mathsf{TLF}$
			$\mid \mathsf{TLF} \vee \mathsf{TLF} \mid \mathbf{A}\ \mathsf{TLPF}$

Figure 4.6: Syntax of PTSL ($1 \leq z \leq d$). In ESF, $Æ_y$ is $\forall$ or $\exists$, at least one $Æ_y$ is $\exists$.

ShadowVisor's separation property states that the physical addresses accessible by the guest must be less than MEM_LIMIT. This requires two distinct conditions depending on the table since pages mapped in the PDT are of size MPS_PDT and pages mapped in the PT are of size MPS_PT. Given a PDT mapped page frame starting at address a, a guest OS can access from a to $a + \mathtt{MPS\_PDT}$ and $a + \mathtt{MPS\_PT}$ for a PT mapped page frame. Hence, to enforce separation, ShadowVisor must restrict the addresses in the shadow page directory to be less than $\mathtt{MEM\_LIMIT} - \mathtt{MPS\_PDT}$ and page table to be less than $\mathtt{MEM\_LIMIT} - \mathtt{MPS\_PT}$. Note that we are making the reasonable assumption that $\mathtt{MEM\_LIMIT} > \mathtt{MAX\_PDT}$ and $\mathtt{MEM\_LIMIT} > \mathtt{MAX\_PT}$ to avoid underflow. This security property is stated in the following USF state formula:

$$\begin{aligned}\varphi_{sep} \triangleq \forall \mathtt{i}_1,\mathtt{i}_2 \,\textbf{.}\quad & (\mathtt{PDT}[\mathtt{i}_1].\mathtt{F}[\mathtt{sPRESENT}] \wedge \mathtt{PDT}[\mathtt{i}_1].\mathtt{F}[\mathtt{sPSE}] \Rightarrow \\ & (\mathtt{PDT}[\mathtt{i}_1].\mathtt{F}[\mathtt{sADDR}] < \mathtt{MEM\_LIMIT} - \mathtt{MPS\_PDT})) \wedge \\ & (\mathtt{PDT}[\mathtt{i}_1].\mathtt{F}[\mathtt{sPRESENT}] \wedge \neg \mathtt{PDT}[\mathtt{i}_1].\mathtt{F}[\mathtt{sPSE}] \wedge \\ & \ \mathtt{PDT}[\mathtt{i}_1].\mathtt{PT}[\mathtt{i}_2].\mathtt{F}[\mathtt{sPTE\_PRESENT}] \Rightarrow \\ & (\mathtt{PDT}[\mathtt{i}_1].\mathtt{PT}[\mathtt{i}_2].\mathtt{F}[\mathtt{sADDR}] < \mathtt{MEM\_LIMIT} - \mathtt{MPT\_PT}))\end{aligned}$$

Semantics. We now present the semantics of our specification logic. We further overload the $\mapsto$ operator such that for any PTSL formula π, variable x, and numeral m, we write $\pi[x \mapsto m]$ to mean the result of substituting all occurrences of x in π with m. We start with the notion of satisfaction of formulas by stores.

Definition 11. *The satisfaction of a formula π by a store σ (denoted $\sigma \models \pi$) is defined, by induction on the structure of π, as follows:*

- $\sigma \models \mathtt{b}$ *iff* $\sigma^B(\mathtt{b}) = \textbf{true}$
- $\sigma \models \mathtt{P}[\mathtt{k}_1]\ldots\mathtt{P}[\mathtt{k}_z].\mathtt{F}[\mathtt{r}]$ *iff* $(\lceil \mathtt{k}_1 \rceil, \ldots, \lceil \mathtt{k}_z \rceil, \lceil \mathtt{r} \rceil) \in Dom(\sigma^P_z)$ *and* $\sigma^P_z(\lceil \mathtt{k}_1 \rceil, \ldots, \lceil \mathtt{k}_z \rceil, \lceil \mathtt{r} \rceil) = \textbf{true}$
- $\sigma \models \neg\pi$ *iff* $\sigma \not\models \pi$
- $\sigma \models \pi_1 \wedge \pi_2$ *iff* $\sigma \models \pi_1$ *and* $\sigma \models \pi_2$
- $\sigma \models \pi_1 \vee \pi_2$ *iff* $\sigma \models \pi_1$ *or* $\sigma \models \pi_2$
- $\sigma \models Æ_1 \mathtt{i}_1, \ldots, Æ_z \mathtt{i}_z \,\textbf{.}\, \pi$ *iff* $Æ_1 k_1 \in [1, \sigma^n_1] \ldots Æ_z k_z \in [1, \sigma^n_z] \,\textbf{.}\, \sigma \downharpoonleft (k_1, \ldots, k_z) \models \pi[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_z \mapsto 1]$

The definition of satisfaction of Boolean formulas and the logical operators are standard. Parametric formulas, denoted $\mathtt{P}[\mathtt{k}_1]\ldots\mathtt{P}[\mathtt{k}_z].\mathtt{F}[\mathtt{r}]$, are satisfied if and only if the indices $\mathtt{k}_1, \ldots, \mathtt{k}_z, \mathtt{r}$ are in bounds, and the element at the specified location is **true**. Quantified formulas are satisfied by σ if and only if appropriate (depending on the quantifiers) projections of σ satisfy the formula obtained by substituting 1 for the quantified variables in π. We present the semantics of a *PGCL*$^+$ program as a *Kripke structure*.

Kripke Semantics. Let gc be any *PGCL*$^+$ guarded command and $\mathsf{k} \in \mathbb{N}^d$. We denote the set of stores σ such that $\sigma^n = \mathsf{k}$, as $Store(\mathsf{gc}(\mathsf{k}))$. Note that $Store(\mathsf{gc}(\mathsf{k}))$ is finite. Let *Init* be any formula and $\mathsf{AP} = \mathsf{USF}$ be the set of atomic propositions. Intuitively, a Kripke structure $M(\mathsf{gc}(\mathsf{k}), Init)$ over AP is induced by executing $\mathsf{gc}(\mathsf{k})$ starting from any store $\sigma \in Store(\mathsf{gc}(\mathsf{k}))$ that satisfies *Init*.

Definition 12. *Let Init* $\in \mathsf{USF}$ *be any formula. Formally,* $M(\mathsf{gc}(\mathsf{k}), Init)$ *is a four tuple* $(\mathcal{S}, I, \mathcal{T}, \mathcal{L})$*, where:*

- $\mathcal{S} = Store(\mathsf{gc}(\mathsf{k}))$ *is a set of states;*
- $I = \{\sigma | \sigma \models Init\}$ *is a set of initial states;*
- $\mathcal{T} = \{(\sigma, \sigma') \mid \{\sigma\}\mathsf{gc}(\mathsf{k})\{\sigma'\}\}$ *is a transition relation given by the operational semantics of PGCL*$^+$*; and*
- $\mathcal{L} : \mathcal{S} \to 2^{\mathsf{AP}}$ *is the function that labels each state with the set of propositions true in that state; formally,*

$$\forall \sigma \in \mathcal{S} \centerdot \mathcal{L}(\sigma) = \{\varphi \in \mathsf{AP} \mid \sigma \models \varphi\}$$

If ϕ is a PTSL formula, then $M, \sigma \models \phi$ means that ϕ holds at state σ in the Kripke structure M. We use an inductive definition of $\models$ [20]. Informally, an atomic proposition π holds at σ iff $\sigma \models \pi$; $\mathbf{A}\ \phi$ holds at σ if ϕ holds on all possible (infinite) paths starting from σ. TLPF formulas hold on paths. A TLF formula ϕ holds on a path Π iff it holds at the first state of Π; $\mathbf{X}\ \phi$ holds on a path Π iff ϕ holds on the suffix of Π starting at second state of Π; $\phi_1\ \mathbf{U}\ \phi_2$ holds on Π if ϕ_1 holds on suffixes of Π until ϕ_2 begins to hold. The definitions for $\neg$, $\wedge$ and $\vee$ are standard.

Simulation. For Kripke structures M_1 and M_2, we write $M_1 \preceq M_2$ to mean that M_1

is simulated by M_2. We use the standard definition of simulation [20] (presented in the appendix). Since satisfaction of ACTL* formulas is preserved by simulation [20], and PTSL is a subset of ACTL*, we claim that PTSL formulas are also preserved by simulation. The claim is expressed formally by Fact 3 in the appendix.

4.3.5 Small Model Theorems

In this section, we present two new small model theorems. While the statement of the theorems appear superficially similar to those in Chapter 3, they apply to *PGCL*$^+$ programs and the extended PTSL logic developed in this chapter. Both theorems relate the behavior of a *PGCL*$^+$ program with nested parametric arrays with arbitrarily many rows at each depth to its behavior with a single row at each level of nesting. The following definition extends the previous definition of exhibits to a tuple $\mathsf{k} \in \mathbb{N}^d$.

Definition 13 (Exhibits)**.** *A Kripke structure* $M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$ *exhibits a formula* φ *iff there is a reachable state* σ *of* $M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$ *such that* $\sigma \models \varphi$.

The first theorem applies to safety properties.

Theorem 9 (Small Model Safety 1)**.** *Let* $\mathsf{gc}(\mathsf{k})$ *be any instantiated guarded command in PGCL*$^+$. *Let* $\varphi \in \mathsf{GSF}$ *be any generic state formula in* PTSL, *and Init* $\in \mathsf{USF}$ *be any universal state formula in* PTSL. *Then* $M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$ *exhibits* φ *iff* $M(\mathsf{gc}(\mathtt{1}^d), \mathit{Init})$ *exhibits* φ.

The second theorem is more general, and relates Kripke structures via simulation.

Theorem 10 (Small Model Simulation)**.** *Let* $\mathsf{gc}(\mathsf{k})$ *be any instantiated guarded command in PGCL*$^+$. *Let Init* $\in \mathsf{GSF}$ *be any generic state formula in* PTSL. *Then* $M(\mathsf{gc}(\mathsf{k}), \mathit{Init}) \preceq M(\mathsf{gc}(\mathtt{1}^d), \mathit{Init})$ *and* $M(\mathsf{gc}(\mathtt{1}^d), \mathit{Init}) \preceq M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$.

Since, simulation preserves PTSL specifications, we obtain the following immediate corollary to Theorem 10.

Corollary 11 (Small Model Safety 2)**.** *Let* gc(k) *be any instantiated guarded command in PGCL$^+$. Let* $\varphi \in$ USF *be any universal state formula in* PTSL*, and Init* $\in$ GSF *be any generic state formula in* PTSL. *Then* $M(\texttt{gc(k)}, Init)$ *exhibits* φ *iff* $M(\texttt{gc}(\texttt{1}^d), Init)$ *exhibits* φ.

Note that Corollary 11 is the dual of Theorem 9 obtained by swapping the types of φ and *Init*. The proofs of Theorems 9 and 10 involve mutual induction over both the structure of commands, and the depth of the parametric array P. This is due to the recursive nature of *PGCL*$^+$, where commands at level z refer to paramaterized commands at level z, which in turn refer to commands at level $z+1$. We defer these proofs to the appendix.

4.4 Case Studies

We present two case studies – ShadowVisor and Xen – to illustrate our approach. In addition to these two examples, we believe that our approach is, in general, applicable to all paging systems that are strictly hierarchical. This includes paging modes of x86 [48] and ARM [5], two of the most widely used architectures in practice. Note that super-pages and super-sections in the ARM architecture require 16 adjacent entries to be identical in the page tables. However, this requirement can be eliminated by a shadow paging implementation that splits the super-pages and super-sections in the guest page tables into small pages and sections within the shadow page tables, and handles the synchronization appropriately.

```
shadow_page_fault_original ≡
  for $i_1$ do
    $PDT[i_1].F[gPRESENT] \wedge PDT[i_1].F[gPSE] \wedge PDT[i_1].F[gADDR] < MEM\_LIMIT$ ?
      $PDT[i_1].F[sPDE] := PDT[i_1].F[gPDE]$;
    for $i_2$ do
      $PDT[i_1].F[gPRESENT] \wedge PDT[i_1].PT[i_2].F[gPTE\_PRESENT] \wedge$
      $PDT[i_1].PT[i_2].F[gPTE\_ADDR] < MEM\_LIMIT$ ?
        $PDT[i_1].PT[i_2].F[sPTE] := PDT[i_1].PT[i_2].F[gPTE]$;
```

Figure 4.7: ShadowVisor's vulnerable shadow page fault handler.

4.4.1 ShadowVisor

Recall our model of ShadowVisor from Section 4.3.2 and the expression of ShadowVisor's initial condition and security properties as PTSL formulas from Section 3.3.3.

ShadowVisor's separation property states that the physical addresses accessible by the guest OS must be less than the lowest address of hypervisor protected memory, denoted MEM_LIMIT. This requires two distinct conditions depending on the table containing the mapping since pages mapped in PDTs are of size MPS_PDT and pages mapped in PTs are of size MPS_PT. Given a page frame of size s with starting address a, a guest OS can access any address in the range $[a, a+s]$. Hence, subtracting the maximum page size prevents pages from overlapping the hypervisor's protected memory. Note that we are making the reasonable assumption that $\texttt{MEM\_LIMIT} > \texttt{MPS\_PDT}$ and $\texttt{MEM\_LIMIT} > \texttt{MPS\_PT}$ to avoid underflow.

In ShadowVisor's original shadow page fault handler, shown in Figure 4.7, the conditionals allowed page directory and page table entries to start at addresses up to MEM_LIMIT. As a result, ShadowVisor running shadow_page_fault_original has a serious vulnerability where separation is violated by an adversary that non-deterministically chooses an address a such that $a + \texttt{MPS\_PDT} \geq \texttt{MEM\_LIMIT}$ or $a + \texttt{MPS\_PT} \geq \texttt{MEM\_LIMIT}$. This vulnerability exists in ShadowVisor's design and C source code implementation. We were able

PT-Size	Time(s)	Vars	Clauses
1	0.14	1828	3685
10	7.08	94163	201732
20	79.6	362795	783022
30	*	*	*

Table 4.1: ShadowVisor verification with increasing PT size. * means out of 1GB memory limit; Vars, Clauses = # of CNF variables and clauses generated by CBMC.

to fix the vulnerability by adding appropriate checks and verify that the resulting model is indeed secure. We present our verification of *PGCL*$^+$ models below.

Both the vulnerable and repaired ShadowVisor programs are expressible as a *PGCL*$^+$ program, the initial state is expressible in USF, and the negation of the address separation property is expressible in GSF. Therefore, Theorem 9 applies and we need only verify the system with one table at each depth with one entry per table (i.e., a parameter of (1,1)).

Effectiveness of Small Model Theorems. For a concrete evaluation of the effectiveness of our small model theorems, we verified ShadowVisor with increasing sizes of page tables at both levels. More specifically, we created models of ShadowVisor in C (note that a guarded command in *PGCL*$^+$ is expressible in C) for various PT sizes (the sizes at both PT levels were kept equal).

We then verified two properties using CBMC[1], a state-of-the-art model checker for C:

- the initial state of the system ensures separation;
- if the system started in a state that ensures separation, executing any of the four guarded commands in the ShadowVisor model preserves separation.

By induction, verifying these two properties guarantees that ShadowVisor ensures separation perpetually. Our results are shown in Table 4.1. Note that verification for size 1

[1]`www.cprover.org/cbmc`

(which is sound and complete due to our small model theorem) is quick, while it blows up for even page tables of size 30 (an unrealistically small number, implying that brute-force verification of ShadowVisor is intractable).

4.4.2 Xen

Next, we analyze address separation in a model of the Xen hypervisor, built from the source code of Xen version 3.0.3. Xen manages multiple virtual machines, each running a guest OS instance with multiple processes (i.e., contexts). Xen maintains a separate sPT for each context, and uses context caching (cf. Sec. 4.2).

We model Xen's context cache using a nested parametric array of depth 4. At the top level, row $\mathtt{P}_1[\mathtt{i}_\mathtt{i}]$ (denoted $\mathtt{VM}[\mathtt{i}_1]$ below) contains an entry for a particular VM's guest. At the next level, the array $\mathtt{P}_1[\mathtt{i}_1].\mathtt{P}_2$ (denoted $\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}$ below) contains an entry for each context of the $\mathtt{i}_1$-th guest. Next, the array $\mathtt{P}_1[\mathtt{i}_1].\mathtt{P}_2[\mathtt{i}_2].\mathtt{P}_3$ (denoted $\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}$) represents the PDT of the $\mathtt{i}_2$-th context of the $\mathtt{i}_1$-th guest OS. Finally, the array $\mathtt{P}_1[\mathtt{i}_1].\mathtt{P}_2[\mathtt{i}_2].\mathtt{P}_3[\mathtt{i}_3].\mathtt{P}_4$ (denoted $\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{PT}$) is the PT of the $\mathtt{i}_3$-th page directory table entry of the $\mathtt{i}_2$-th context of the $\mathtt{i}_1$-th guest.

Security Property. Recall that our basic separation property requires that the physical addresses mapped by the shadow page directory and shadow page table are less than a constant MEM_LIMIT minus a max page size. This constant represents the physical memory address at which hypervisor protected memory begins. We consider a natural extension of this separation property for a context caching system with multiple VMs that states that all VMs and contexts should be separate from VMM protected memory. This security property is stated in the following USF state formula:

$$\begin{aligned}
&\varphi_{sep} \triangleq \forall \mathtt{i}_1,\mathtt{i}_2,\mathtt{i}_3,\mathtt{i}_4 \,\textbf{.}\\
&\quad(\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{F}[\mathtt{sPRESENT}] \wedge \mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{F}[\mathtt{sPSE}] \Rightarrow\\
&\quad(\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{F}[\mathtt{sADDR}] < \mathtt{MEM\_LIMIT} - \mathtt{MPS\_PDT}))\wedge\\
&\quad(\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{F}[\mathtt{sPRESENT}] \wedge \neg\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{F}[\mathtt{sPSE}] \Rightarrow\\
&\quad(\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{PT}[\mathtt{i}_4].\mathtt{F}[\mathtt{sADDR}] < \mathtt{MEM\_LIMIT} - \mathtt{MPS\_PT}))
\end{aligned}$$

Initial State. We model Xen as starting in an initial state where all entries of all of the sPDT and sPT are marked as not present. This is expressed by the following USF formula:

$$\begin{aligned}
\mathit{Init} \triangleq \forall \mathtt{i}_1,\mathtt{i}_2,\mathtt{i}_3,\mathtt{i}_4 \,\textbf{.}\quad &\neg\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{F}[\mathtt{sPRESENT}]\wedge\\
&\neg\mathtt{VM}[\mathtt{i}_1].\mathtt{Ctx}[\mathtt{i}_2].\mathtt{PDT}[\mathtt{i}_3].\mathtt{PT}[\mathtt{i}_4].\mathtt{F}[\mathtt{sPRESENT}]
\end{aligned}$$

We define the Xen address translation system using context caching in *PGCL*$^+$ as follows:

```
XenAddressTrans ≡     shadow_page_fault
                    || shadow_invalidate_page
                    || context_caching_new_context
                    || Xen_adversary
```

The commands `shadow_page_fault` and `shadow_invalidate_page` generalize their counterparts for ShadowVisor over multiple VMs and contexts, and are omitted. The following *PGCL*$^+$ guarded command implements `context_caching_new_context` .

```
context_caching_new_context ≡
 for $i_1$ do
  for $i_2$ do
   for $i_3$ do
     $*$ ? VM[$i_1$].Ctx[$i_2$].PDT[$i_3$].F[sPDE] := 0;
```

PT-Size	Time(s)	Vars	Clauses
1	0.76	5774	13634
3	4.44	34480	81666
6	24.91	122214	289674
9	*	*	*

Table 4.2: Xen verification with increasing PT size. * means out of 1GB memory limit; Vars, Clauses = # of CNF variables and clauses generated by CBMC.

Note that to model VM and process scheduling soundly, we assume non-deterministic context switching. Hence, we extend ShadowVisor's `shadow_new_context` to non-deterministically clear contexts.

Finally, we consider an adversary model where the the attacker has control over an unbounded but finite number of VMs, each with a unbounded but finite number of contexts. This adversary is therefore expressed as follows:

```
Xen_adversary ≡
 for $i_1$ do
  for $i_2$ do
   for $i_3$ do
    VM[$i_1$].Ctx[$i_2$].PDT[$i_3$].F[gPDE] := $*$;
    for $i_4$ do
     VM[$i_1$].Ctx[$i_2$].PDT[$i_3$].PT[$i_4$].F[gPTE] := $*$;
```

Our Xen model is clearly expressible in *PGCL*$^+$, its initial state is expressible in USF, and the negation of the address separation property is expressible in GSF. Therefore, Theorem 9 applies and we need only verify the system with one table at each depth with one entry per table (i.e., a system parameter of (1,1,1,1)).

Effectiveness of Small Model Theorems. As in the case of ShadowVisor we verified the

Xen model with increasing (but equal) sizes of page tables at both levels, and 2 VMs and 2 contexts per VM. We verified the same two properties as for ShadowVisor to inductively prove that Xen ensures separation perpetually. Our results are shown in Table 4.2. Note again that verification for size 1 (which is sound and complete due to our small model theorem) is quick, while it blows up for even page tables of size 9 (an unrealistically small number, implying that brute-force verification of Xen is also intractable).

4.5 Conclusion

Verifying separation properties of address translation mechanisms of operating systems, hypervisors, and virtual machine monitors in the presence of adversaries is an important challenge toward developing secure systems. A significant factor behind the complexity of this challenge is that the data structures over which the translation mechanisms operate have both unbounded size and unbounded nesting depth. We developed a parametric verification technique to address this challenge. Our approach involves a new modeling language and specification mechanism to model and verify such parametric systems. We applied this methodology to verify that the designs of two hypervisors – ShadowVisor and Xen – correctly enforce the expected security properties in the presence of adversaries. Next, we extend our approach to operate directly on system implementations, and relax the restrictions of row independence and hierarchical row uniformity.

Chapter 5

The Havoc Adversary Abstraction

5.1 Introduction

A significant source of model checking complexity is adversary-controlled data structures (e.g., kernel page tables) and "accessor" functions that operate over them (e.g., for reading and writing kernel page table entries). Since model checkers explore the complete state space of a program, an increase in the number of variables and functions can potentially lead to an exponentially larger analysis cost – the state space explosion problem.

A crucial insight is that many accessor functions return values read from adversary-controlled data structures and not constrained by input validation checks. An example is a hypervisor function that reads from a guest page table to return the physical address corresponding to a virtual address. Since adversary-controlled data-structures (e.g., the guest page table), by definition of our adversary model, contain arbitrary values when an accessor function is invoked, the return value of the accessor is also arbitrary. Therefore, it is possible to replace a call to an accessor function with a non-deterministic assignment. The resulting code is smaller, simpler, and still verifiable by state-of-the-art software model checkers (all of which support non-deterministic assignments).

Based on this insight, we develop an abstraction technique consisting of two main steps:

Step 1: Detection. For every function f of a system, determine if it returns a non-deterministic value; we call such a function *havoc*.

Step 2: Replacement. If f is found to be a havoc function, replace every call $x := f()$ by the non-deterministic assignment $x = *$.

We develop an automated technique to perform havoc function detection. Specifically, we formalize the problem of deciding whether a function f is a havoc function as a validity problem for a Quantified Boolean formula (QBF). The formula is of the form $\forall o \centerdot \exists i \centerdot \phi_f(i,o)$ where $\phi_f(i,o)$ captures the relation between the inputs i and outputs o of f. The form of quantification ensures that the formula is valid iff for all possible values of output variables o there exist values for input variables i such that the function will produce the outputs o. In other words, since the adversary controls the inputs to the function, the return value of the function is truly non-deterministic. We formalize and prove this claim (see Theorem 12 in Section 5.2).

Once Step 1 succeeds to prove that a function f is havoc, the next step of our abstraction is to replace every call $x := f()$ by the non-deterministic assignment $x = *$. One of our key results is (see Theorem 13 in Section 5.2) that our abstraction is *sound*—no attacks are missed by the abstraction. In addition, our QBF-based formulation of the detection problem is an efficient technique for proving a form of completeness - we term local completeness - directly at the source level.

We implement havoc function detection via a tool chain to:

1. Automatically construct a QBF formula corresponding to $\forall o \centerdot \exists i \centerdot \phi_f(i,o)$ from C source code for f.

2. Use a state-of-the-art QBF solver, e.g., Skizzo, QuBe, GhostQ, and Quantor, to solve the QBF formula obtained above.

Our empirical evaluation demonstrates the effectiveness of this abstraction on real software: we identified two vulnerabilities in the C code of ShadowVisor (a prototype hypervisor), and successfully model check the code using the CBMC model checker after fixing the vulnerabilities. These vulnerabilities are related to those identified in the previous chapter. However, in this chapter they were discovered directly at the source code level rather than the design level. As a second case study, we model check two security properties of the SecVisor security hypervisor's implementation. Without the abstraction, CBMC either times out or exhausts system resources in its failed attempts to model check these systems. After applying the abstraction, CBMC takes less than 3 seconds on five-year-old hardware to model check each system.

The rest of this chapter is organized as follows. In Section 5.2, we describe our methodology, followed by a case study verification of ShadowVisor and SecVisor in Section 5.3. We conclude in Section 5.4.

5.2 Havoc-Function Abstraction

We present our abstraction methodology – which we refer to as *havoc-function abstraction* – in detail. Figure 5.1 illustrates the kind of systems and adversaries we consider. The figure depicts a kernel that exposes an interface, a CSI-adversary, and an adversary-controlled data structure. When we define a CSI-adversary, we abstract away software in an adversary-controlled layer (e.g., user space) and endow the adversary with a set of capabilities (e.g., the ability to make system calls) that account for any program the adversary could execute. However, abstracting away software in adversary-controlled layers alone

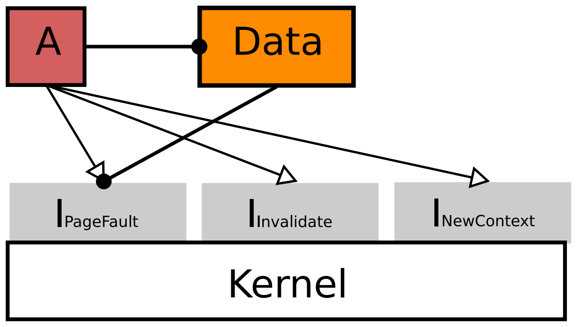


Figure 5.1: A CSI-adversary operates on data that is read by a kernel.

(i.e., ignoring everything above the system call interface) does not necessarily make model checking tractable. Havoc-function abstraction aims to reduce the state space further by eliminating adversary-controlled data structures and functions inside the kernel itself.

Example. We describe havoc-function abstraction over a running example of a hypervisor system that synchronizes an adversary-controlled GPT with a hypervisor-controlled SPT. The hypervisor's synchronization routine synchronizes the SPT with the GPT by indexing into the GPT, reading an adversary-controlled entry, and writing the entry into an entry in the SPT. Havoc-function abstraction enables us to replace the GPT and the code that reads from it by a non-deterministic assignment. Note that our small model abstractions typically do not apply to hypervisor or adversary programs that manipulate the GPT because these programs do not obey the necessary data-flow restrictions (i.e., row uniformity and row hierarchically) required for the small model theorems applicability.

In many cases, this replacement results in a cascade effect where operations that use a non-deterministic value as an operand are further reduced if their result is completely non-deterministic. The cascade effect also works in the "opposite" direction to abstract code that occurs before the data-structure access. For example, the computation of an index into a GPT can be abstracted if all values read from an index result in a non-deterministic value. The result of this cascade is a reduction of the resulting search space. In the case of ShadowVisor, model checking the abstract system is feasible while model checking the concrete system is intractable. We report experimental results in Section 5.3.

5.2.1 Problem Statement

We now present havoc-function abstraction. Let f be the target function, i be its input variables, and o be its output variables. If f is written in C, then i is the set of its input variables, and any global variable it reads from, while o is its return variable (if any) and all global variables it writes to. Without loss of generality, we assume a single input variable i and a single output variable o.

Definition 14 (Function Semantics)**.** *The semantics of a function f with input variable i and output variable o, denoted by $Sem(f)$, is a formula with $\{i,o\}$ as its free variables, such that following holds: there exists a satisfying assignment A to $Sem(f)$ iff there is an execution of f where the initial value of i is $A(i)$, and the final value of o is $A(o)$.*

In our experiments, we use the CBMC tool to extract $Sem(f)$ from the C code of f. We now define a havoc function.

Definition 15 (Havoc Function)**.** *A function f, with input variable i and output variable o, is said to be a havoc function iff for every value v_o of o there exists a value v_i of i and an execution of f such that the initial value of i is v_i, and the final value of o is v_o.*

We reduce the problem of deciding if a function f is havoc to deciding the validity of a logical formula. The following theorem expresses our reduction.

Theorem 12. *(Havoc Function Detection as Validity) A function f, with input variable i and output variable o, is a havoc function iff the following havoc-verification-condition formula $HVC(f)$ is valid:*

$$HVC(f) \equiv \forall o \,.\, \exists i \,.\, Sem(f)$$

Proof. Follows directly from Definition 14 and Definition 15. □

In our implementation, we extract $Sem(f)$ from the C code of f as a Boolean formula using the CBMC tool. After adding the required quantifiers, $HVC(f)$ is a QBF. Thus we reduce the problem of deciding if a function f is havoc to deciding QBF validity. This reduction allows us to exploit recent progress in the development of efficient QBF solvers.

5.2.2 Soundness

Once we prove that a function f is havoc, the next step of havoc-function abstraction is to replace every call $x := f()$ by the non-deterministic assignment $x = *$. We now prove the soundness of havoc-function abstraction.

Soundness of Abstraction. We consider an abstraction α as a transformation that converts a concrete program p to an abstract program $\alpha(p)$. The abstraction is *sound* for safety properties if for any program p: if a safety property φ holds on $\alpha(p)$ then it also holds on p. Equivalently, it is known that an abstraction α is sound for safety properties if for any program p, each finite execution of p is also an execution of $\alpha(p)$. We now prove the soundness of havoc-function abstraction.

Theorem 13. *Soundness Havoc-function abstraction is sound.*

Proof. (Sketch) Let p be a program and $\alpha(p)$ be the result of applying havoc-function abstraction on it. Without loss of generality, assume that p contains a single havoc function f called once. Thus, $\alpha(p)$ is the same as p, except that the call $x := f()$ is replaced by the non-deterministic assignment $x = *$. Also, let f have a single input variable i and a single output variable o.

Let e be a finite execution of p. Consider a sub-execution e' of e where a call to f is being executed. Suppose e' ends with a value v_o being assigned to o. Replace the entire

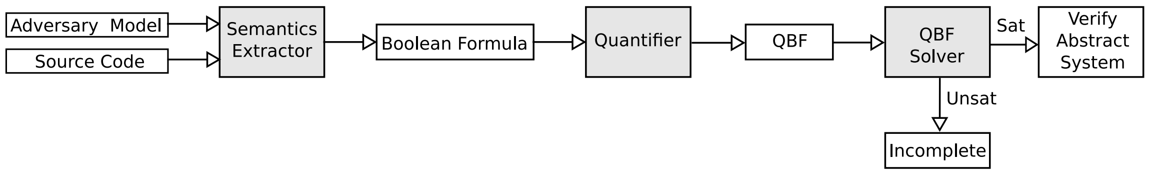


Figure 5.2: Overview of abstraction process.

sub-execution e' by the direct assignment of v_o to o to obtain the new finite execution $\alpha(e)$. Clearly, $\alpha(e)$ is an execution of $\alpha(p)$. This proves that α is sound.

$\square$

We now describe the overall havoc-function abstraction process in more detail, including automatic construction of QBFs from C source code, and solving QBFs to check for havoc functions.

5.2.3 Details

An overview of the entire havoc-function abstraction process followed by model checking of the abstracted code is shown in Figure 5.2. It consists of five steps, which we describe now in more detail.

Step 1: Specify adversary model. First, we specify a CSI-adversary as a set of functions which the adversary calls an arbitrary number of times with arbitrary parameter values. To specify data structures that are under adversary control, we either encapsulate these data structures in simple accessor functions or explicitly include a set of adversary-controlled data structures in the adversary model.

In our running example, we specify a malicious guest OS as constrained to the interface exposed to software running at the hardware privilege level of user code. Note that we rely on hardware-enforced privilege-level separation to prevent the adversary from circumventing the system call interface. We feel this is an advantageous problem separation as it enables a compositional analysis whereby the hardware and software are verified separately and verification results are composed to achieve a system-wide security property that covers both the hardware and software.

Step 2: Convert code region to Boolean formulas. Next, we use a semantics extractor (e.g., CBMC) to automatically convert source code regions that touch adversary-controlled data structures to a Boolean formula representation. The semantics extractor is composed of two components: a program annotator and a Boolean formula generator. The program annotator identifies and tags output variables and lazily initializes adversary-controlled data. The Boolean formula generator converts C code to Boolean formulas. These Boolean formulas encode the semantics of code regions.

Code regions can be of any granularity. For concreteness of presentation, we operate at function granularity, but our techniques apply to other levels of granularity as well. We select functions by extracting a function call graph by starting at the adversary interface and walking the call graph to identify functions that read from adversary-controlled data structures. For each such function, we use the program annotator to identify and tag output variables.

We implemented our program annotator using OCAML and the CIL source-to-source transformation framework. For Boolean formula generation, we use the CBMC tool [16] to convert C code to a Boolean formula in DIMACS format [23]. In some cases, CBMC makes assumptions about the system memory model, in particular, the initialization of variables. In cases where these assumptions do not match our requirements, we initialize variables according to our requirements. For example, CBMC initializes all global variables to zero. We instead initialize adversary-controlled global variables to a non-deterministic value.

Step 3: Quantify Boolean formulas. In the previous step, our approach produces a set of Boolean formulas where each formula, denoted $\varphi_f(i,o)$, defines the semantics of a function f (see Definition 14) with input variable i and output variable o. From each formula $\varphi_f(i,o)$, we construct the formula $HVC_f = \forall o\,.\,\exists i\,.\,\varphi_f(i,o)$, and then check the validity of HVC_f using a QBF solver.

We developed an automated quantification tool that takes as input $\varphi_f(i,o)$ in DIMACS format and generates $\forall o\,.\,\exists i\,.\,\varphi_f(i,o)$ in QDIMACS format. DIMACS and QDIMACS are standard formats for SAT and QBF solvers, respectively. Using them gives us flexibility in our choice of solvers.

Step 4: Solve QBF, abstract code. Next, we pass the QBFs output from Step 3 to a QBF solver. If the QBF $\forall o\,.\,\exists i\,.\,\varphi_f(i,o)$ is valid, then f is a havoc function. In this case, we proceed with the next step of havoc-function abstraction and replace all calls $x := f()$ by the non-deterministic assignment $x = *$.

Note that Steps 2, 3 and 4 were performed for each function in the source code by traversing the call-graph in a bottom-up manner. This enables us to leverage a cascade of havocs, where a function g that calls function f is determined to be havoc after all calls $x := f()$ in the body of g have been replaced by $x = *$.

The QBF validity problem is PSPACE-complete. However, modern QBF solvers employ a variety of efficient decision procedures to quickly decide validity. Our implementation interfaces with any QBF solver that supports QDIMACS, including sKizzo [72], QuBe [63], and GhostQ [51]. All the solvers take a QDIMACS-formatted input and return either “valid” or “invalid”. In most cases we’ve encountered, the solvers immediately return an answer and the different solvers have always agreed on the answer returned. In a few cases, a solver has taken longer (e.g., several minutes) than the others to solve a QBF, but the correct answer was eventually returned.

Step 5: Model check abstract system. Finally, we pass the abstracted system to a software model checker. If the model checker halts and returns that the target security property is satisfied, then the concrete system is secure against the interface constrained adversary defined in Step 1. If instead it halts and returns that the target security property is unsatisfied, then it outputs a counter-example – an assignment of values and a program trace that lead to the insecure state. Since our approach is sound, we are guaranteed that if no counter-example is returned then the system is indeed secure.

In our implementation, we employ CBMC to verify the abstract system. CBMC's support for bit-vectors and bit-wise operations makes it well-suited to verify the types of functions typically found in operating system kernel code. CBMC also supports the casts between integers and pointers that are necessary for address separation.

5.3 Case Studies

In this section, we present empirical evaluation of our approach on two hypervisors – ShadowVisor and SecVisor.

5.3.1 ShadowVisor

Our first case-study is the verification of ShadowVisor, a fully-functional shadow-paging hypervisor we built for the x86 platform. This case study demonstrates that our approach enables software model checking of realistic systems.

System Overview. ShadowVisor's goal is to protect its memory, which contains its code and data, from a potentially malicious guest OS. To accomplish this goal, ShadowVisor virtualizes the guest's view of memory through the use of shadow paging. In shadow paging, the guest's page tables are remapped using a set of underlying shadow page-tables that the guest is unaware of. Only the shadow page tables are used by the hardware memory management unit, making them the authoritative page tables.

When a guest OS modifies its page tables, ShadowVisor interposes and updates the shadow page tables after performing necessary security checks. To ensure address space separation, ShadowVisor performs a number of simple security checks to verify that the physical addresses mapped by the guest page tables are within an allowed range. These checks are implemented as conditionals that compare physical memory addresses against an integer constant signifying the largest physical memory address that a guest is allowed to access. If designed and implemented correctly, ShadowVisor's shadow paging mechanism should ensure that the guest OS cannot write to protected memory, thereby ensuring the integrity of ShadowVisor's code and data.

Complicating matters is the fact that the guest page tables and the shadow page tables are implemented as multi-level page tables. Multi-level page tables virtualize large source address spaces (e.g., 32 or 64-bit address spaces) without the overhead of maintaining one page table entry for each source page number. They are implemented using a tree of linked tables where lower level (indexed by less-significant bits of the virtual address) tables are excluded if no addresses in the relevant ranges are present.

In the two level paging approach used by ShadowVisor, there is a top level Page Directory Table and a set of Page Tables. Virtual addresses are split into three fields and used to index into the different tables to obtain the corresponding physical page frame. PDT entries can contain either a page frame number (indicated by the Page Size Extension, or PSE, flag being set), an address of a page table, or neither (indicated by the PRESENT bit being clear).

ShadowVisor is implemented in about 2000 lines of C code (374 of which is the actual shadow paging logic). Its small code size makes it an ideal target for software model checking. However, its extensive use of large and complex data structures such as guest and shadow page tables make it challenging to verify. In this case study, we demonstrate that the abstractions we developed make model checking ShadowVisor's source code feasible. Model checking the system without applying abstraction is simply infeasible since the size of the state space increases exponentially with the number of page table entries.

Adversary Model. ShadowVisor's adversary model is that of a malicious guest OS. The CSI-adversary abstraction gives us a natural way to express a malicious guest OS as an interface containing the hypercall interface and the guest page tables. In particular, we define the malicious guest OS adversary as having access to the three functions in the hypercall interface, $new\_context$, $invalidate$, and $page\_fault$, and the guest page table data structures including g_pdt and g_pt.

Security Property. ShadowVisor's goal is to separate a malicious guest OS from the hypervisor's protected memory. The protected memory region starts near the top of memory as defined by the constant $PHYMEM\_LIMIT$. We specify ShadowVisor's security property using the assertion language that is used by CBMC with the addition of implication and a universal quantifier. The assertion states if a Page Directory Table entry is present and directly mapped to a memory address (i.e., has the PSE flag set) then it should map to an address that is less than the physical memory limit. This is specified as follows:

$$\begin{aligned}\forall i.(s\_pdt[i]\&PAGE\_PRESENT)\&\&(s\_pdt[i]\&PAGE\_PSE)) \Rightarrow \\ (((u32)(s\_pdt[i])\&(\sim((u32)PAGE\_SIZE\_4K-1)))+ \\ PAGE\_SIZE\_4M) < PHYMEM\_LIMIT)\end{aligned} \tag{5.1}$$

We apply the small model abstraction of Chapters 3 and 4 to simplify this formula further. This enables us to consider an s_pdt array with a single element. Thus, our security property reduces to the following formula:

$$\begin{aligned}(s\_pdt[0]\&PAGE\_PRESENT)\&\&(s\_pdt[0]\&PAGE\_PSE)) \Rightarrow \\ (((u32)(s\_pdt[0])\&(\sim((u32)PAGE\_SIZE\_4K-1)))+ \\ PAGE\_SIZE\_4M) < PHYMEM\_LIMIT)\end{aligned} \tag{5.2}$$

Abstraction

We describe the steps of our abstraction and model checking process and show the results of applying our abstractions. During the verification of ShadowVisor, we re-discovered the two previously identified "page-overlap" vulnerabilities that allow a malicious guest OS to violate memory separation. This time, they were discovered directly at the source code level without the need for a model of the system's design.

Level of Automation. For the havoc-function abstraction process, human input is only required to specify the adversary model. For the verification process, human input is only required to specify the adversary model, initial condition, and security property. Since our adversary models are simply sets of procedure and data structure names, it is possible to automate the process of generating adversary models. However, one would have to then verify the system against all possible combinations of functions and data structures, many of which may not represent "reasonable" adversary models. Since CSI-adversary models are concise and easy to specify, we manually specify them. After providing these inputs, the tool runs automatically without any user intervention. We show intermediate steps below only to demonstrate the process – during a normal verification, they are invisible to the user.

Havoc-Function Abstraction. By employing the havoc-function abstraction, we are able to abstract ShadowVisor's *get_guestentry* function. This function takes as input a guest virtual address, *gva*, the guest's CR3 control register containing the guest physical memory address of the start of the guest PDT, *gCR*3, and two pointers to pointers to thirty-two bit values. The function sets these pointers to point to the guest PDT and PTE as indexed by the *gva*. Since the contents of the guest PDT and PT are non-deterministic, we expect the resulting reads to be non-deterministic and for the abstraction to eliminate unnecessary code.

```c
void get_guestentry(u32 gva, u32 gCR3, u32 **pdt_entry, u32 **pt_entry){
  u32 index_pdt, index_pt, g_pdt_entry;
  npt_t g_pt;

  index_pdt= (gva >> 22);
  index_pt  = ((gva & (u32)0x003FFFFF) >> 12);

  *pdt_entry = *pt_entry = (u32 *)0;
  g_pdt_entry = g_pdt[index_pdt];
  *pdt_entry = (u32 *)& g_pdt[index_pdt];

  if( !(g_pdt_entry & PAGE_PRESENT) )
    return;

  if(g_pdt_entry & PAGE_PSE)
    return;

  g_pt = (npt_t)(u32)pae_get_addr_from_pde(g_pdt_entry);

  *pt_entry = (u32 *)&g_pt[index_pt];
  return;
}
```

After passing the above code to our abstraction tool, the QBF solver returns that the thirty-two bit values, set by the function after indexing into adversary-controlled data structures, are non-deterministic. Our tool then abstracts the function and generates the following abstract code. The entire abstraction process took just a few seconds to complete.

```
void get_guestentry(u32 gva, u32 gCR3, u32 **pdt_entry, u32 **pt_entry){
  *pt_entry = (u32 *)0;

  *pdt_entry = (u32 *)& nondet_u32();

  if( !(**pdt_entry & PAGE_PRESENT) )
    return;

  if(**pdt_entry & PAGE_PSE)
    return;

  *pt_entry = (u32 *)& nondet_u32();
  return;
}
```

Note that the adversary-controlled guest data structures have been abstracted away and replaced by a function that returns non-deterministic 32-bit values (i.e., $nondet\_u32()$). The result is a significantly simpler, sound abstraction that can be model checked more efficiently than the concrete program.

Small Model Abstraction. We utilize the small model theorems to reduce the size of the shadow PDT to a single entry. That entry may point to a single PT with a single entry. The abstraction, which we apply manually, allows us to remove loops that iterate over table entries and operate instead on a single value. For example, consider the following code that allocates a page table by indexing into an array holding all the page tables and setting each entry of one page table to zero.

```
u32 shadow_alloc_pt(u32 gva){
  u32 index_pdt;
  index_pdt= (gva >> 22);

  for (int i=0; i < 1024; i++) {
    *((u32 *)((index_pdt * PAGE_SIZE_4K) + (u32)shadow_p_tables)+i) = (u32) 0;
  }

  return ( ((index_pdt * PAGE_SIZE_4K) + (u32)shadow_p_tables) );
}
```

After abstracting the code, we are left with a single assignment to the one remaining page table's single entry.

```
u32 shadow_alloc_pt(){
  *((u32 *)shadow__p_tables = (u32) 0;
  return  (u32)shadow_p_tables;
}
```

We apply this abstraction at any point in ShadowVisor that operates on the shadow page directory and shadow page tables. The abstraction's reduction of source code is substantial – it eliminates every loop in ShadowVisor's shadow paging functions. While the loops always perform a bounded number of iterations (e.g. 1024), the elimination of loops removes the central biggest hurdle that a model checker must handle. Without the abstraction, CBMC would unroll loops for a finite number of iterations. In the above example, if the number of iterations was less than 1024, then the result would be an incomplete verification that is suitable for only bug finding, not for verification. We ran a number of experiments where we directed CBMC to completely unroll all loops, but the resulting formula generation exhausted the memory of the machines. Even if it was able to generate the formula, exhaustively searching the state space would be infeasible.

Model Checking Abstract System

We model checked the abstracted C code using CBMC. We discovered two vulnerabilities in the implementation of the shadow-paging code as shown below.

Page Overlap Vulnerabilities.

```
    if( ((u32)(gPDE) & (~((u32)PAGE_SIZE_4K - 1))) < PHYMEM_LIMIT){
      s_pd_t[index_pdt] = gPDE;
    }

                ...

    if( ((u32)(gPTE) & (~((u32)PAGE_SIZE_4K - 1))) < PHYMEM_LIMIT){
      s_pt[index_pt] = gPTE;
    }
```

ShadowVisor's shadow paging logic keeps the shadow page tables synchronized with the guest page tables during the lifetime of the guest execution. This synchronization results in two different types of updates to the shadow page tables depending upon the guest page table contents: (a) updating a PDT entry (in case of a 4MB physical memory mapping), and (b) updating a page table entry (in case of a 4KB physical memory mapping). Either of these updates are performed after employing the guest physical memory limit checks as shown above.

If *PHYMEM_LIMIT* is a multiple of 4MB, these checks ensure that the guest can never access physical memory beyond what it is allocated. However, if *PHYMEM_LIMIT* is not a multiple of 4MB (let say *PHYMEM_LIMIT*=2MB) and if the adversary makes a 4MB mapping at the PDT entry corresponding to *PHYMEM_LIMIT* (in our example entry 0) then the first check shown above will pass and the shadow page table will be updated. However, since its a 4MB mapping, the adversary can now access anywhere from 0-4MB which is beyond *PHYMEM_LIMIT* (2MB in our example). A similar vulnerability is found with a 4K guest page mapping and the second check as shown above, if *PHYMEM_LIMIT* is not a multiple of 4K.

Vulnerability Repair and Successful Model Checking. We repaired the first vulnerability by adding *PAGE_SIZE_4M* to the left hand side of the conditional. Similarly, we repaired the second vulnerability by adding *PAGE_SIZE_4K* to the left hand side of the second conditional. These two constants denote the maximum page size that can be mapped in the two scenarios. By adding the maximum page size to the page table and page directory entries, the conditionals check that the adversary is unable to create a page that overlaps with the protected memory range.

We ran CBMC on the repaired abstract system and it returned that the security property was satisfied after approximately 3 seconds of runtime on a several-year-old laptop. This result is in contrast to attempts to verify the system without the optimizations, which ran out of memory during the process of generating formulas. These formulas need to encode all possible 1024 entries of the shadow PDT. Moreover, for each entry of the shadow PDT, the formula encodes a shadow PT which has 1024 additional entries. The total formula must encode more than one million entries each with thirty-two bits allowing for $2^{33,554,432}$ possible values. In contrast, our abstraction techniques completely abstracted away the guest PDT and PTs, reduced the size of the shadow PDT to a single entry, and eliminated all but a single shadow PT with a single entry. The resulting abstract system only contains two, thirty-two bit entries between the shadow PDT and PT.

5.3.2 SecVisor

We perform our second case study on the SecVisor [70] security hypervisor. SecVisor was built independently of ShadowVisor by different authors, but uses a similar shadow paging approach to memory virtualization. SecVisor's design and implementation differ significantly from ShadowVisor. In particular, SecVisor uses a three-level paging scheme that adds an additional level of complexity and aims to satisfy different security properties (lifetime kernel code integrity) than ShadowVisor. Despite this additional complexity and significant differences in design and implementations, we demonstrate that our approach as well as our abstraction technique and tools are flexible and enable efficient software-model checking of security properties at the source level.

System Overview. SecVisor was previously described in Section 3.2. Recall that SecVisor virtualizes memory using shadow paging and sets the memory protection bits in the shadow page tables to prevent integrity violations of authorized code. SecVisor also performs bounds checks to ensure that its own code and data are not modifiable by an adversary.

The SecVisor code base is split between initialization and runtime code with a total of 4092 lines of code. SecVisor's shadow paging code, which is responsible for performing address separation and other runtime security checks is the focus of our verification and is implemented in about 1200 lines of C code.

Adversary. SecVisor's adversary is expressible as a CSI-adversary that is constrained to SecVisor's hypercall interface and has complete control over the guest OS kernel's code and data including the kernel page tables. This model endows the adversary with the ability to synchronize the KPT with the shadow page tables (SPT) and cause transitions between user mode and kernel mode.

Security Property. SecVisor's security properties were previously described in Section 3.2. In addition to those "application-level" security properties, SecVisor must protect its code and data from the adversary and ensure that the adversary is unable to circumvent its interposition on kernel and user mode transitions. SecVisor protects itself using a combination of address separation and bounds checks. In particular, we verify that the adversary cannot map into SecVisor's memory with a KPT entry. We specify this property for the KPT as follows (a similar property is specified for entries in the kernel page directory (KPD)).

$$\begin{gathered}\forall i.!((kpt[i] >= visor\_relocate\_address)\&\& \\ (kpt[i] < (visor\_relocate\_address + 0x2000000)))\end{gathered} \tag{5.3}$$

To prevent circumvention of SecVisor's kernel to user mode and user to kernel mode handlers, we verify that the Global Descriptor Table (GDT), a CPU data structure that contains various code and data segment descriptions (type, base memory address, limit, and privilege level) is always mapped above the guest OS user space's virtual memory mapping (which starts at $0xc0000000$ for the Linux OS). We specify this property as follows.

$$(linux\_vmcb- > gdtr.base >> 28) == 0xc) \tag{5.4}$$

Abstraction

Next, we describe the application of our abstractions to SecVisor's source code.

CSI-adversary Abstraction. We applied the CSI-adversary abstraction to abstract SecVisor's guest OS kernel page table (KPT) walk function, *kernel_pt_walk*. This function indexes up to three levels of the guest OS kernel's page tables and returns entries that are under the control of the adversary. The KPT walker is similar in design to ShadowVisor's *get_guest_entry* code but includes an additional level of page tables to efficiently map the kernel's 64-bit address space. We include a snippet of the function below which is approximately 70 lines of C code. After preprocessing, many of the calls are replaced with inline macros that perform complex bit shifts, masks, and bit-wise logical operations.

```
u32 kernel_pt_walker (u32 vaddr)
    ...
    /* get fields from virtual addr */
    pdp_index = pae_get_pdpt_index(vaddr);
```

```
  pd_index = pae_get_pdt_index(vaddr);
  pt_index = pae_get_pt_index(vaddr);
  offset = pae_get_offset_4K_page(vaddr);

  //tmp is phy addr of page dir
  tmp = pae_get_addr_from_32bit_cr3(kcr3);
  kpdp = (pdpt_t)__gpa2hva((u32)tmp);
  pdp_entry = kpdp[pdp_index];
  tmp = pae_get_addr_from_pdpe(pdp_entry);
  kpd = (pdt_t)__gpa2hva((u32)tmp);
  pd_entry = kpd[pd_index];

  // if 0, then 4 KB page else 2MB
  if ( (pd_entry & _PAGE_PSE) == 0 ) {
    /* get addr of page table from entry */
    tmp = pae_get_addr_from_pde(pd_entry);
    kpt = (pt_t)__gpa2hva((u32)tmp);
    pt_entry  = kpt[pt_index];
    /* find physical page base addr from page table entry */
    paddr = (u64)pae_get_addr_from_pte(pt_entry) + offset;
  }
  else { /* 2MB page */
    offset = pae_get_offset_big(vaddr);
    paddr = pae_get_addr_from_pde_big(pd_entry);
    paddr += (u64)offset;
  }
  ...
  return (u32) paddr;
```

We applied our adversary abstraction tool to the *kernel_pt_walk* code while specifying that $(u32)paddr$ (i.e, the lower 32-bits of the *paddr* variable) could take any possible 32-bit value. The tool returned $TRUE$ in around 3 seconds, signifying that all 70 lines of code of the function could be abstracted to a single return of the function $nondet\_u32()$, a nondeterministic 32-bit value. The function is called at eight distinct locations in SecVisor's code and can be abstracted at each location, to significantly reduce the complexity of each calling function.

Parametricity Abstraction. We applied the parametricity abstraction to SecVisor's shadow paging code in much the same way as in the ShadowVisor case study. Since SecVisor's source code operates over three-level page tables, there were three levels of nested loops (in contrast, to ShadowVisor's two levels) that iterated over all the entries of the guest OS kernel page tables and the shadow page tables performing initialization, synchronization, and security checks. The result of the abstraction was an elimination of these loops and the reduction of each paging level to a single entry in the respective tables (i.e., PDPT, PDT, PT).

Model Checking Abstract System

We model checked the abstracted C code using CBMC and found that it satisfied both of the checked security properties. CBMC took approximately 3 seconds to model check the system. Both abstractions were necessary to model check the system. With just the parametricity abstraction alone, we were able to check that the adversary is unable to map into SecVisor's memory with a KPT entry, but the model checker did not terminate after thirty minutes of attempting to simplify the program's representation as a SAT formula. To test the scaling trends at work with just the parametricity abstraction, we bounded the size of the KPT in the *kernel_pt_walker* function (to alleviate the complexity stemming from this function). We observed runtimes of a few seconds for a kernel page table with less than 10 entries and exponentially increasing runtimes up to around 5 minutes for trivially small page tables with around 90 entries. This scaling test was performed on just a single level of the three-level page table hierarchy.

Next, to show that the adversary abstraction alone is not sufficient, we enabled the adversary abstraction and abstracted away the *kernel_pt_walker* function, but did not apply the parametricity result. Since the page table synchronization code loops over page tables with 1024 entries, the model checker was not able to build the SAT formulas that represented the program semantics.

Finally, we enabled both abstractions and were able to check both properties in just a few seconds of runtime. These experiments demonstrate the importance of employing abstraction techniques that reduce the complexity of both adversary-controlled and trusted data structures.

5.4 Conclusion

Despite its promise, scaling software model checking to verify security properties of systems code remains an open challenge. We presented an approach – called havoc-function abstraction – to address this challenge in the context of verifying security of hypervisors. The key insight behind this approach is that many functions in systems code operate on adversary-controlled data structures. In essence, these functions, which we call havoc functions, return non-deterministic values. We developed a fully automated solution, based on QBF solving, to detect havoc functions and abstract them away in a sound manner. Empirical evaluation on two real-life hypervisors demonstrate the effectiveness of our approach.

Chapter 6

Towards Refinement

6.1 Introduction

In this chapter, we show that small model theorems similar to those in Chapter 4 apply to a detailed model of ShadowVisor's implementation. In particular, we prove a refinement theorem between ShadowVisor's implementation defined in a subset of the C language (denoted *MiniCee*) and an extension to $PGCL^{+}$, $PGCL^{++}$, that allows previously forbidden data flow. This requires a number of steps.

First, the syntax and semantics of our source-level modeling language, *MiniCee*, must be formally defined. Next, new programming constructs must be added to $PGCL^{+}$ to allow previously forbidden data flows that occur in ShadowVisor's implementation. The SMTs must be updated to account for new data flows and programming constructs. A refinement mapping between ShadowVisor's high-level $PGCL^{++}$ model and more detailed *MiniCee* model must be developed. We state and sketch the proof of a refinement theorem that relates the abstract and concrete models by simulation. We address each of these challenges in turn.

We begin by defining the *MiniCee* language, a subset of the ANSI-C standard. *MiniCee* is expressive enough to model our target system, ShadowVisor, while being simple enough to enable concise manual proofs. We reduce the complexity of the language and proofs in the language by abstracting away program constructs that are unused in ShadowVisor such as pointers and inline function calls that would enable recursion.

PGCL^{++} by adding write-only variables that enable previously disallowed data flow from tables lower in the hierarchy to tables higher in the hierarchy. This extension is necessary to model the behavior of the lower-level system and successfully prove a refinement theorem. For example, consider a multi-level paging system that writes to an entry in a higher-level table depending on the value of the flag in a lower-level table. This system is not expressible in *PGCL*$^{+}$ because there is a data flow that violates hierarchical row independence. Indeed, such a system may not be expressible in any language that is amenable to Small Model Theorems with optimal cutoffs. If data is allowed to flow both up and down the hierarchy, then a security property may be violated by a data flow that includes more than just one entry at each level of the data structure. In *PGCL*$^{++}$, we allow a limited form of upward data flow that is sufficient to model systems of interest while being amenable to SMTs with optimal cutoffs. The key idea is allowing upward flow only into write-only variables. We bake this restriction syntactically into the definition of *PGCL*$^{++}$ and prove new SMTs that are analogous to those in Chapter 4.

To establish our refinement result, we fully specify a number of details missing from the ShadowVisor models of previous chapters. In particular, we describe each column of the parametrized arrays and define the store in full detail. We define a refinement mapping that relates states in the ShadowVisor model defined in the *MiniCee* language and states in the model defined in *PGCL*$^{++}$. Our primary technical result is a simulation theorem that preserves PTSL properties between a concrete and abstract model of ShadowVisor.

Organization. This chapter is organized as follows. In Section 6.2, we extend *PGCL*$^+$ to *PGCL*$^{++}$. We define our specification formalism in Chapter 6.3. We extend our SMTs to apply to *PGCL*$^{++}$ programs in Section 6.4. Section 6.5 defines *MiniCee*. We state our refinement theorem and prove it in Chapter 6.6. We conclude in Section 6.7.

6.2 Definition of *PGCL*$^{++}$

We define the syntax and semantics of *PGCL*$^{++}$. *PGCL*$^{++}$ is a strict extension of *PGCL*$^+$ that includes write-only variables to enable a new class of data flows required for our refinement proof.

6.2.1 *PGCL*$^{++}$ Syntax

The syntax of *PGCL*$^{++}$ is similar to *PGCL*$^+$. The primary difference is a new class of write-only variables and a write-only variable assignment operator. All variables in *PGCL*$^{++}$ are Boolean. As in *PGCL*$^+$, *PGCL*$^{++}$ includes nested parametric arrays to a finite depth d. Each row of an array at depth d is a record with a single field F, a finite array of Booleans of size q_d. Each row of an array at depth z ($1 \leq z < d$) is a structure with two fields: F, a finite array of Booleans of size q_z, and P an array at depth $z+1$. Let 1 and 0 be, respectively, the representations of the truth values **true** and **false**. Let B be a set of Boolean variables, W be a set of write-only Boolean variables, $\mathtt{i}_1,\ldots,\mathtt{i}_d$ be variables used to index into $\mathtt{P}_1,\ldots,\mathtt{P}_d$, respectively, and $\mathtt{n}_1,\ldots,\mathtt{n}_d$ be variables used to store the number of rows of $\mathtt{P}_1,\ldots,\mathtt{P}_d$, respectively.

The syntax of *PGCL*$^{++}$ is shown in Figure 6.1. New constructs are highlighted in bold. As in *PGCL*$^+$, *PGCL*$^{++}$ supports natural numbers, Boolean variables, propositional expressions over Boolean variables and F elements, guarded commands that update Boolean vari-

Natural Numerals	$\mathtt{K}$			
Boolean Variables	$\mathtt{B}$			
Write-only Boolean Variables	$\mathtt{W}$			
Parametric Index Variables	$\mathtt{i}_1,\ldots,\mathtt{i}_d$			
Parameter Variables	$\mathtt{n}_1,\ldots,\mathtt{n}_d$			
Expressions	$\mathtt{E}$	$::=$	$1 \mid 0 \mid * \mid \mathtt{B} \mid \mathtt{E}\vee\mathtt{E} \mid \mathtt{E}\wedge\mathtt{E} \mid \neg\mathtt{E}$	
Parameterized Expressions ($1 \leq z \leq d$)	$\widehat{\mathtt{E}}_z$	$::=$	$\mathtt{E} \mid \mathtt{P}[\mathtt{i}_1]\ldots\mathtt{P}[\mathtt{i}_z].\mathtt{F}[\mathtt{K}] \mid \widehat{\mathtt{E}}_z\vee\widehat{\mathtt{E}}_z \mid \widehat{\mathtt{E}}_z\wedge\widehat{\mathtt{E}}_z$	
		$\mid$	$\neg\widehat{\mathtt{E}}_z$	
Instantiated Guarded Commands	$\mathtt{G}$	$::=$	$\mathtt{GC}(\mathtt{K}^d)$	
Guarded Commands	$\mathtt{GC}$	$::=$	$\mathtt{E}\ ?\ \mathtt{C}_1 : \mathtt{C}_1$	Simple guarded command
		$\mid$	$\mathtt{GC} \parallel \mathtt{GC}$	Parallel composition
Commands (depth $1 \leq z \leq d$)	$\mathtt{C}_z$	$::=$	$\mathtt{B} := \mathtt{E}$ (if $z = 1$)	Assignment
		$\mid$	$\mathtt{for}\ \mathtt{i}_z\ \mathtt{do}\ \widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z : \widehat{\mathtt{C}}_z$	Parametric for
		$\mid$	$\mathtt{C}_z;\mathtt{C}_z$	Sequencing
		$\mid$	$\mathtt{skip}$	Skip
Parameterized Commands ($1 \leq z \leq d$)	$\widehat{\mathtt{C}}_z$	$::=$	$\mathtt{P}[\mathtt{i}_1]\ldots\mathtt{P}[\mathtt{i}_z].\mathtt{F}[\mathtt{K}] := \widehat{\mathtt{E}}_z$	Array assignment
		$\mid$	$\mathtt{W} := \widehat{\mathtt{E}}_z$	**Write-only variable assignment**
		$\mid$	$\widehat{\mathtt{C}}_z;\widehat{\mathtt{C}}_z$	Sequencing
		$\mid$	$\mathtt{C}_{z+1}$ (if $z < d$)	Nesting

Figure 6.1: *PGCL*$^{++}$ Syntax, z denotes depth.

ables and F elements, and parallel composition of guarded commands. *PGCL*$^{++}$ extends *PGCL*$^{+}$ by including write-only variables and a write-only variable assignment operator.

We define a parallel interleaving semantics where guarded commands are executed atomically. In more detail, a guarded command e ? c_1 : c_2 executes c_1 or c_2 depending on whether e evaluates to true or false. The parallel composition of two guarded commands executes by non-deterministically picking one of the commands to execute. The sequential composition of two commands executes the first command followed by the second command. Note that commands at depth $z+1$ are nested within those at depth z.

Language Design. As in *PGCL*$^{+}$, values assigned to an element of an F array at depth z can depend only on: (i) other elements of the same F array; (ii) elements of parent F arrays along the nesting hierarchy (to ensure hierarchical row uniformity); and (iii) Boolean variables. Values assigned to Boolean variables depend only on other Boolean variables.

In *PGCL*$^{++}$, we add a write-only assignment operator that enables "upward" data flow from F and Boolean variables B to a class of write-only Boolean variables, W. The write-only variables can be seen as a part of the array F, however for simplicity we separate their definitions. The addition of write-only variables and a write-only assignment operator allows us to model systems that were not able to be expressed in *PGCL*$^+$. For example, a multi-level paging system that writes to an entry in a higher-level table depending on the value of the flag in a lower-level table. This system is not expressible in *PGCL*$^+$ because there is a data flow that violates hierarchical row independence. Since *PGCL*$^{++}$ is a strict extension of *PGCL*$^+$, we are able to model all the previous systems and new systems of interest.

6.2.2 *PGCL*$^{++}$ Semantics

We now present the operational semantics of *PGCL*$^{++}$ as a relation on stores. The semantics are a strict extension of *PGCL*$^+$ with an extended store that includes a mapping of Write-only Boolean variables to Boolean values. Formally, in *PGCL*$^{++}$, a store σ is a tuple $(\sigma^B, \sigma^W, \sigma^n, \sigma^P)$ such that:

- $\sigma^B : \mathtt{B} \to \mathbb{B}$ maps Boolean variables to $\mathbb{B}$;
- $\sigma^W : \mathtt{W} \to \mathbb{B}$ maps Write-only Boolean variables to $\mathbb{B}$;
- $\sigma^n \in \mathbb{N}^d$ is a tuple of values of the parameter variables;
- σ^P is a tuple of functions defined as follows:

$$\forall z \in [1,d] \centerdot \sigma^P_z : \otimes(\sigma^n_{1,z}, q_z) \to \mathbb{B}$$

The rules for evaluating *PGCL*$^{++}$ expressions under stores are analogous to those in *PGCL*$^+$ and are defined inductively over the structure of *PGCL*$^{++}$ expressions, and shown in Figure 6.2.

To define the semantics of *PGCL*$^{++}$, we first present the updated notion of store projection. The following definitions are straight-forward extensions of those presented in

$$\frac{}{\langle 1,\sigma\rangle \to \mathbf{true}} \qquad \frac{}{\langle 0,\sigma\rangle \to \mathbf{false}} \qquad \frac{}{\langle *,\sigma\rangle \to \mathbf{true}} \qquad \frac{}{\langle *,\sigma\rangle \to \mathbf{false}} \qquad \frac{\mathtt{b} \in dom(\sigma^B)}{\langle \mathtt{b},\sigma\rangle \to \sigma^B(\mathtt{b})}$$

$$\frac{\mathtt{w} \in dom(\sigma^W)}{\langle \mathtt{w},\sigma\rangle \to \sigma^W(\mathtt{w})} \qquad \frac{\langle \mathsf{e},\sigma\rangle \to t}{\langle \neg\mathsf{e},\sigma\rangle \to [\neg]t} \qquad \frac{(\lceil \mathtt{k}_1 \rceil,\ldots,\lceil \mathtt{k}_z \rceil,\lceil \mathtt{r} \rceil) \in Dom(\sigma^P_z)}{\langle \mathtt{P}[\mathtt{k}_1]\ldots\mathtt{P}[\mathtt{k}_z].\mathtt{F}[\mathtt{r}],\sigma\rangle \to \sigma^P_z(\lceil \mathtt{k}_1 \rceil,\ldots,\lceil \mathtt{k}_z \rceil,\lceil \mathtt{r} \rceil)}$$

$$\frac{\langle \mathsf{e},\sigma\rangle \to t \qquad \langle \mathsf{e}',\sigma\rangle \to t'}{\langle \mathsf{e} \vee \mathsf{e}',\sigma\rangle \to t[\vee]t'} \qquad \frac{\langle \mathsf{e},\sigma\rangle \to t \qquad \langle \mathsf{e}',\sigma\rangle \to t'}{\langle \mathsf{e} \wedge \mathsf{e}',\sigma\rangle \to t[\wedge]t'}$$

Figure 6.2: Rules for expression evaluation in $PGCL^{++}$.

previous chapters. We repeat them here for purposes of presentation.

Definition 16 (Store Projection)**.** *Let* $\sigma = (\sigma^B, \sigma^W, \sigma^n, \sigma^P)$ *be any store and* $1 \leq z \leq d$*. For* $\mathsf{k} = (k_1, \ldots, k_z) \in \otimes(\sigma^n_1, \ldots, \sigma^n_z)$ *we write* $\sigma \downharpoonright \mathsf{k}$ *to mean the store* $(\sigma^B, \sigma^W, \sigma^m, \sigma^Q)$ *such that:*

$$\sigma^m = \sigma^n[1 \mapsto 1][2 \mapsto 1]\ldots[z \mapsto 1] \tag{1}$$

$$\forall y \in [1,z] \centerdot \forall X \in Dom(\sigma^Q_y) \centerdot \sigma^Q_y(X) = \sigma^P_y(X[1 \mapsto k_1][2 \mapsto k_2]\ldots[y \mapsto k_y]) \tag{2}$$

$$\forall y \in [z+1,d] \centerdot \forall X \in Dom(\sigma^Q_y) \centerdot \sigma^Q_y(X) = \sigma^P_y(X[1 \mapsto k_1][2 \mapsto k_2]\ldots[z \mapsto k_z]) \tag{3}$$

Note: $\forall z \in [1,d] \centerdot \forall \mathsf{k} \in \otimes(\sigma^n_1, \ldots, \sigma^n_z) \centerdot \sigma \downharpoonright \mathsf{k} = (\sigma \downharpoonright \mathsf{k}) \downharpoonright \mathtt{1}^z$.

$\sigma \downharpoonright \mathsf{k}$ retains σ^B and σ^W, changes the first z elements of σ^n to 1, and leaves the remaining elements unchanged, and projects away all but the k_y-th row of the parametric array at depth y for $1 \leq y \leq z$. Note that since projection retains σ^B and σ^W, it does not affect the evaluation of expressions that do not refer to elements of $\mathtt{P}$.

$$\frac{\sigma^n = (\lceil \mathtt{k}_1 \rceil, \ldots, \lceil \mathtt{k}_d \rceil) \qquad \{\sigma\}\ \mathtt{gc}\ \{\sigma'\}}{\{\sigma\}\ \mathtt{gc}(\mathtt{k}_1, \ldots, \mathtt{k}_d)\ \{\sigma'\}}\text{Parameter Instantiation}$$

$$\frac{\{\sigma\}\ \mathtt{c}\ \{\sigma''\} \qquad \{\sigma''\}\ \mathtt{c}'\ \{\sigma'\}}{\{\sigma\}\ \mathtt{c};\mathtt{c}'\ \{\sigma'\}}\text{Sequential} \qquad\qquad \frac{}{\{\sigma\}\ \mathtt{skip}\ \{\sigma\}}\text{Skip}$$

$$\frac{\begin{array}{c}\sigma^n_{1,z} = (1^{z-1}, N) \qquad \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2 \in (\widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z : \widehat{\mathtt{C}}_z)[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_{z-1} \mapsto 1] \\ \forall y \in [1, N] \,.\, \{\sigma \downarrow (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downarrow (1^{z-1}, y)\}\end{array}}{\{\sigma\}\ \mathtt{for}\ \mathtt{i}_z\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2\ \{\sigma'\}}\text{Unroll}$$

$$\frac{\langle \mathtt{e}, \sigma \rangle \to \mathbf{true} \wedge \{\sigma\}\ \mathtt{c}_1\ \{\sigma'\} \bigvee \langle \mathtt{e}, \sigma \rangle \to \mathbf{false} \wedge \{\sigma\}\ \mathtt{c}_2\ \{\sigma'\}}{\{\sigma\}\ \mathtt{e}\ ?\ \mathtt{c}_1 : \mathtt{c}_2\ \{\sigma'\}}\text{GC}$$

$$\frac{\{\sigma\}\ \mathtt{gc}\ \{\sigma'\} \vee \{\sigma\}\ \mathtt{gc}'\ \{\sigma'\}}{\{\sigma\}\ \mathtt{gc} \parallel \mathtt{gc}'\ \{\sigma'\}}\text{Parallel} \qquad\qquad \frac{\langle \mathtt{e}, \sigma \rangle \to t}{\{\sigma\}\ \mathtt{b} := \mathtt{e}\ \{\sigma[\sigma^B \mapsto \sigma^B[\mathtt{b} \mapsto t]]\}}\text{Assign}$$

$$\frac{\widehat{\mathtt{e}} \in \widehat{\mathtt{E}}_z \qquad \langle \widehat{\mathtt{e}}, \sigma \rangle \to t}{\{\sigma\}\ \mathtt{w} := \widehat{\mathtt{e}}\ \{\sigma[\sigma^W \mapsto \sigma^W[\mathtt{w} \mapsto t]]\}}\text{Write-only Assign}$$

$$\frac{\widehat{\mathtt{e}} \in \widehat{\mathtt{E}}_z \qquad \langle \widehat{\mathtt{e}}, \sigma \rangle \to t \qquad (\lceil \mathtt{k}_1 \rceil, \ldots, \lceil \mathtt{k}_z \rceil, \lceil \mathtt{r} \rceil) \in \mathit{Dom}(\sigma^P_z)}{\{\sigma\}\ \mathtt{P}[\mathtt{k}_1] \ldots \mathtt{P}[\mathtt{k}_z].\mathtt{F}[\mathtt{r}] := \widehat{\mathtt{e}}\ \{\sigma[\sigma^P \mapsto \sigma^P[\sigma^P_z \mapsto [\sigma^P_z[((\lceil \mathtt{k}_1 \rceil, \ldots, \lceil \mathtt{k}_z \rceil, \lceil \mathtt{r} \rceil) \mapsto t]]]]\}}\text{Parameterized Array Assign}$$

Figure 6.3: Rules for *PGCL*$^{++}$ commands

Store Transformation. For any *PGCL*$^{++}$ command $\mathtt{c}$ and stores σ and σ', we write $\{\sigma\}\ \mathtt{c}\ \{\sigma'\}$ to mean that σ is transformed to σ' by the execution of $\mathtt{c}$. We define $\{\sigma\}\ \mathtt{c}\ \{\sigma'\}$ via induction on the structure of $\mathtt{c}$, as shown in Figure 6.3.

The only rule not already in *PGCL*$^{+}$ is the "Write-only Assign" rule that states that a σ is transformed to σ' by assigning either a Boolean variable, an element of $\mathtt{F}$, or the result of a Boolean expression to a write-only variable. Note that write-only variables are not included in the right-hand size of any commands or allowed in expressions. This is our definition of write-only. We define Loop Variable Instantiation as in *PGCL*$^{+}$.

Definition 17 (Loop Variable Instantiation)**.** *Let σ and σ' be two stores such that $\sigma^n = \sigma'^n$. Let $1 \le z \le d$ and $\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2 \in \widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z$ be a guarded command. Then for any $1 \le y \le z$ and $k_y \in [1, \sigma^n_y]$, we write $\{\sigma\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)(\mathtt{i}_y \gg k_y)\ \{\sigma'\}$ to mean:*

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \mathsf{i}_y = k_y \Rightarrow \{\sigma \downharpoonright \mathsf{i}\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_y \mapsto \mathtt{1}]\ \{\sigma' \downharpoonright \mathsf{i}\} \bigwedge \mathsf{i}_y \ne k_y \Rightarrow \sigma \downharpoonright \mathsf{i} = \sigma' \downharpoonright \mathsf{i}$$

Thus, $\{\sigma\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)(\mathtt{i}_y \gg k_y)\ \{\sigma'\}$ means that σ' is obtained from σ by first replacing $\mathtt{i}_y$ with k_y in $\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2$, and then executing the resulting guarded command.

6.2.3 ShadowVisor Code in *PGCL*$^{++}$

Figure 6.4 shows our ShadowVisor model in *PGCL*$^{++}$. Since all *PGCL*$^{++}$ variables are Boolean, we write $\mathtt{x} < \mathtt{C}$ to mean the binary comparison between a finite-valued variable $\mathtt{x}$ and a constant $\mathtt{C}$.

Data Structures. Our ShadowVisor model uses a 2-level PT scheme. The guest and shadow Page Directory Table (gPDT and sPDT) and the guest and shadow Page Tables (gPTs and sPTs) are modeled using the 2-level *PGCL*$^{++}$ parametric array.

Let PDT be the top-level array P. Elements $\mathtt{PDT[i_1].F[gPRESENT]}$ and $\mathtt{PDT[i_1].F[gPSE]}$

are the present and page size extension flags for the $\mathtt{i_1}$-th gPDT entry, while $\mathtt{PDT[i_1].F[gADDR]}$ is the destination address contained in the $\mathtt{i_1}$-th gPDT entry. Elements sPRESENT, sPSE, and sADDR are defined analogously for sPD entries. Below, we refer to the present and page size extension flags of the $\mathtt{i_1}$ entry of the gPDT and sPDT as $\mathtt{PDT[i_1].F[gFlags]}$ and $\mathtt{PDT[i_1].F[sFlags]}$, respectively.

Let $\mathtt{PDT[i_1].PT}$ be the array $\mathtt{P[i_1].P}$. Elements gPTE_PRESENT and gPTE_ADDR of $\mathtt{PDT[i_1].PT[i_2].F}$ are the present flag and destination address contained in the $\mathtt{i_2}$-th entry of the PT pointed to by the $\mathtt{i_1}$-th gPDT entry. Elements sPTE_PRESENT and sPTE_ADDR of $\mathtt{PDT[i_1].PT[i_2].F}$ are similarly defined for the sPT. Terms gPDE refers to the set of elements corresponding to a gPDT entry (i.e., gPRESENT, gPSE, and gADDR). Terms gPTE, sPDE and sPTE are defined similarly for the gPT, sPDT, and sPT, respectively.

Interface. ShadowVisor's interface is a parallel composition of four guarded commands $\mathtt{shadow\_page\_fault_A}$, $\mathtt{shadow\_invalidate\_page_A}$, $\mathtt{shadow\_new\_context_A}$, and $\mathtt{adversary_A}$. The commands operate as in previous chapters. Command $\mathtt{shadow\_page\_fault_A}$ synchronizes sPDT and sPT with gPDT and gPT. Command $\mathtt{shadow\_invalidate\_page_A}$ invalidates entries in the sPD and sPT. Command $\mathtt{shadow\_new\_context_A}$ initializes a new context by clearing all the entries of the sPD. Finally, command $\mathtt{adversary_A}$ models the attacker by arbitrarily modifying every gPDT and gPT entry.

6.3 Specification Formalism

Temporal logic specifications are expressed in PTSL as defined in Chapter 4. The semantics of a *PGCL*$^{++}$ program are defined as in Chapter 4.

```
adversary_A ≡
  for i_1 do
    PDT[i_1].F[ADV] := *;
    PDT[i_1].F[gPDE] := *;
    for i_2 do
      PDT[i_1].PT[i_2].F[ADV] := *;
      PDT[i_1].PT[i_2].F[gPTE] := *;

shadow_page_fault_A ≡
  for i_1 do
    PDT[i_1].F[ADV] ∧ PDT[i_1].F[gPRESENT] ∧ PDT[i_1].F[gPSE]∧
    PDT[i_1].F[gADDR] < PMEM_LIMIT − SIZE_4M ?
      PDT[i_1].F[sPDE] := PDT[i_1].F[gPDE];
    for i_2 do
      PDT[i_1].F[ADV] ∧ PDT[i_1].F[gPRESENT] ∧ PDT[i_1].PT[i_2].F[gPTE_PRESENT]∧
      PDT[i_1].PT[i_2].F[gPTE_ADDR] < PMEM_LIMIT − SIZE_4K ?
        PDT[i_1].PT[i_2].F[sPTE] := PDT[i_1].PT[i_2].F[gPTE];
        PDT[i_1].F[sFlags] := PDT[i_1].F[gFlags];

shadow_invalidate_page_A ≡
  for i_1 do
    (PDT[i_1].F[sPRESENT] ∧ ¬PDT[i_1].F[gPRESENT])∨
    (PDT[i_1].F[sPRESENT] ∧ PDT[i_1].F[gPRESENT]∧
    (PDT[i_1].F[sPSE] ∨ PDT[i_1].F[gPSE])) ?
      PDT[i_1].F[sPDE] := 0;
  for i_1 do
    PDT[i_1].F[sPRESENT] ∧ PDT[i_1].F[gPRESENT]∧
    ¬PDT[i_1].F[gPSE] ∧ ¬PDT[i_1].F[sPSE] ?
      for i_2 do
        PDT[i_1].PT[i_2].F[sPTE] := 0;

shadow_new_context_A ≡
  for i_1 do
    PDT[i_1].F[sPDE] := 0;
```

Figure 6.4: ShadowVisor model in *PGCL*$^{++}$.

6.4 Small Model Theorems

The following small model theorems are identical to those for *PGCL*$^+$, except they apply to *PGCL*$^{++}$ programs. Both theorems relate the behavior of a *PGCL*$^{++}$ program when P has arbitrarily many rows to its behavior when P has a single row. The first theorem applies to safety properties.

Definition 18 (Exhibits)**.** *A Kripke structure* $M(\mathtt{gc(k)}, Init)$ *exhibits a formula* φ *iff there is a reachable state* σ *of* $M(\mathtt{gc(k)}, Init)$ *such that* $\sigma \models \varphi$.

Theorem 14 (Small Model Safety 1)**.** *Let* $\mathtt{gc(k)}$ *be any instantiated guarded command in PGCL*$^{++}$. *Let* $\varphi \in \mathsf{GSF}$ *be any generic state formula, and Init* $\in \mathsf{USF}$ *be any universal state formula. Then* $M(\mathtt{gc(k)}, Init)$ *exhibits* φ *iff* $M(\mathtt{gc(1}^d\mathtt{)}, Init)$ *exhibits* φ.

The second theorem relates Kripke structures via simulation.

Theorem 15 (Small Model Simulation)**.** *Let* $\mathtt{gc(k)}$ *be any instantiated guarded command in PGCL*$^{++}$. *Let Init* $\in \mathsf{GSF}$ *be any generic state formula. Then* $M(\mathtt{gc(k)}, Init) \preceq M(\mathtt{gc(1}^d\mathtt{)}, Init)$ *and* $M(\mathtt{gc(1}^d\mathtt{)}, Init) \preceq M(\mathtt{gc(k)}, Init)$.

Since, simulation preserves PTSL specifications, we obtain the following immediate corollary to Theorem 15.

Corollary 16 (Small Model Safety 2)**.** *Let* $\mathtt{gc(k)}$ *be any instantiated guarded command in PGCL*$^{++}$. *Let* $\varphi \in \mathsf{USF}$ *be any universal state formula, and Init* $\in \mathsf{GSF}$ *be any generic state formula. Then* $M(\mathtt{gc(k)}, Init)$ *exhibits* φ *iff* $M(\mathtt{gc(1}^d\mathtt{)}, Init)$ *exhibits* φ.

6.5 *MiniCee* Definition

We define *MiniCee*, our language for specifying concrete programs.

6.5.1 *MiniCee* Syntax

MiniCee is a simple imperative language whose syntax and semantics are based on ANSI-C. A primary abstraction employed in *MiniCee* is the use of multi-dimensional arrays to model memory. The arrays-as-memory abstraction enables us to ignore pointers and issues related to aliasing. To implement this abstraction, we include pointers in the syntax of *MiniCee*, but abstract pointer operations in the semantics by converting them to semantic nops.

The syntax of *MiniCee* is shown in Figure 6.5. The language supports ASCII characters, natural numerals, finite precision unsigned integer variables (bit vectors), semantically-meaningless pointer variables, and standard arithmetic, bitwise, logical, and relational operators over bit vectors. Since all variables are unsigned 32bit vectors, we ignore types and treat the `u32` as sugar which is excluded from our syntax. Like *PGCL*$^{++}$, *MiniCee* includes nested parametric arrays to a finite depth d. Each row of an array at depth d is a record with a single field $\mathtt{F}$, a finite array of Booleans of size q_d. Each row of an array at depth z ($1 \leq z < d$) is a structure with two fields: $\mathtt{F}$, a finite array of Booleans of size q_z, and $\mathtt{P}$ an array at depth $z+1$. Let 1 and 0 be, respectively, the representations of the truth values **true** and **false**. Let $\mathtt{I}$ be a set of Integer variables, $\mathtt{R}$ be a set of pointer variables, $\mathtt{i}_1, \ldots, \mathtt{i}_d$ be variables used to index into $\mathtt{P}_1, \ldots, \mathtt{P}_d$, respectively, and $\mathtt{n}_1, \ldots, \mathtt{n}_d$ be variables used to store the number of rows of $\mathtt{P}_1, \ldots, \mathtt{P}_d$, respectively.

Unlike *PGCL*$^{++}$, *MiniCee* allows the parameterized array to be updated in a non-row-uniform and non-row-hierarchical manner. As a result, our Small Model Theorems do not directly apply to programs written in *MiniCee*. Instead, we apply our SMTs to *MiniCee* programs through a refinement argument.

ASCII Characters	S			
Natural Numerals	K			
Integer Variables	I			
Pointer Variables	R			
Parametric Index Variables	$\mathtt{i}_1,\ldots,\mathtt{i}_d$			
Parameter Variables	$\mathtt{n}_1,\ldots,\mathtt{n}_d$			
Arithmetic Operators	aop	::=	$+ \mid - \mid \% \mid * \mid /$	
Binary Bitwise Operators	bop	::=	$\& \mid \mathtt{or} \mid \sim \mid << \mid >>$	
Unary Bitwise Operators	ubop	::=	$\sim$	
Binary Logical Operators	lop	::=	$\&\& \mid \vee$	
Unary Logical Operators	ulop	::=	!	
Pointer Operators	pop	::=	$*$	
Relational Operators	rop	::=	$> \mid < \mid \geq \mid \leq \mid != \mid ==$	
Integer Expressions	IE	::=	$\mathtt{K} \mid * \mid \mathtt{I} \mid \mathtt{IE\ aop\ IE} \mid \mathtt{IE\ bop\ IE}$	
		$\mid$	$\mathtt{IE\ lop\ IE} \mid \mathtt{IE\ rop\ IE} \mid \mathtt{ulop\ IE}$	
		$\mid$	$\mathtt{ubop\ IE} \mid \mathtt{P}[\mathtt{i}_1]\ldots\mathtt{P}[\mathtt{i}_z].\mathtt{F}[\mathtt{K}]$	
Program	Prg	::=	FnList Intr	
Function Defn. List	FnList	::=	$\mathtt{FnList\ Fn} \mid \mathtt{Fn}$	
Function Defn.	Fn	::=	$\mathtt{u32\ S}^{+}(\mathtt{FP})\{\mathtt{C}\}$	
Fun. Parameters	FP	::=	$\mathtt{I,FP} \mid \mathtt{I}$	
Interface	Intr	::=	IFn	
		$\mid$	$\mathtt{Intr} \parallel \mathtt{Intr}$	Parallel composition
Instantiated Function	IFn	::=	$\mathtt{C}(*)$	
Commands	C	::=	$\mathtt{P}[\mathtt{i}_1]\ldots\mathtt{P}[\mathtt{i}_z].\mathtt{F}[\mathtt{K}] = \mathtt{IE}$	Assignment (Array)
		$\mid$	$\mathtt{I} = \mathtt{IE}$	Assignment (Integer)
		$\mid$	$\mathtt{R} = \mathtt{IE}$	Assignment (Pointer)
		$\mid$	$\mathtt{if(IE)}\{\mathtt{C}\}\mathtt{else}\{\mathtt{C}\}$	Conditional
		$\mid$	return IE	Return
		$\mid$	C;C	Sequencing

Figure 6.5: *MiniCee* Syntax

$$\frac{\langle \mathtt{e},\sigma\rangle \to \mathbf{true} \land \{\sigma\}\ \mathtt{c}_1\ \{\sigma'\} \bigvee \langle \mathtt{e},\sigma\rangle \to \mathbf{false} \land \{\sigma\}\ \mathtt{c}_2\ \{\sigma'\}}{\{\sigma\}\ \mathtt{if(e)\{c_1\}else\{c_2\}}\ \{\sigma'\}}\text{COND}$$

$$\frac{\langle \mathtt{i},\sigma\rangle \to t}{\{\sigma\}\ \mathtt{i} = \mathtt{i}\ \{\sigma[\sigma^I \mapsto \sigma^I[\mathtt{b} \mapsto t]]\}}\text{Assign (Integer)} \qquad \frac{\langle \mathtt{i},\sigma\rangle \to t}{\{\sigma\}\ \mathtt{p} = \mathtt{i}\ \{\sigma\}}\text{Pointer Assign (Nop)}$$

$$\frac{\langle \mathtt{e},\sigma\rangle \to t}{\{\sigma\}\ \mathtt{return\ e}\ \{\sigma\}}\text{Return (Nop)}$$

$$\frac{\langle \mathtt{e},\sigma\rangle \to t \qquad (\lceil \mathtt{k}_1 \rceil,\ldots,\lceil \mathtt{k}_z \rceil) \in Dom(\sigma^P_z)}{\{\sigma\}\ \mathtt{P}[\mathtt{k}_1]\ldots\mathtt{P}[\mathtt{k}_z] = \mathtt{e}\ \{\sigma[\sigma^P \mapsto \sigma^P[\sigma^P_z \mapsto [\sigma^P_z[(\lceil \mathtt{k}_1 \rceil,\ldots,\lceil \mathtt{k}_z \rceil) \mapsto t]]]]\}}\text{Parameterized Array Assign}$$

Figure 6.6: Rules for *MiniCee* commands

6.5.2 *MiniCee* Semantics

We now present the operational semantics of *MiniCee* as a relation on stores. Let $\mathbb{B}$ be the truth values $\{\mathbf{true},\mathbf{false}\}$. Let $\mathbb{N}$ denote the set of natural numbers. For two natural numbers j and k such that $j \le k$, we write $[j,k]$ to mean the set of numbers in the closed range from j to k. We write $Dom(f)$ to mean the domain of a function f; $(\mathtt{t},\mathtt{t}')$ denotes the concatenation of tuples $\mathtt{t}$ and $\mathtt{t}'$; $\mathtt{t}_{i,j}$ is the subtuple of $\mathtt{t}$ from the i^{th} to the j^{th} elements, and $\mathtt{t}_i$ means $\mathtt{t}_{i,i}$. Given a tuple of natural numbers $\mathtt{t} = (t_1,\ldots,t_z)$, we write $\otimes(\mathtt{t})$ to denote the set of tuples $[1,t_1] \times \cdots \times [1,t_z]$. We write $\mathtt{B}^k$ to represent the set of all bit (Boolean) vectors of length k. Recall that, for $1 \le z \le d$, q_z is the size of the array $\mathtt{F}$ at depth z. Then, a store σ is a tuple $(\sigma^I,\sigma^n,\sigma^P)$ such that:

- $\sigma^I : \mathtt{I} \to \mathtt{B}^{32}$ maps Integer variables to $\mathbb{B}^{32}$;
- $\sigma^n \in \mathbb{N}^d$ is a tuple of values of the parameter variables;
- σ^P is a tuple of functions defined as follows:

$$\forall z \in [1,d]\,.\,\sigma^P_z : \otimes(\sigma^n_{1,z},q_z) \to \mathbb{B}$$

The rules for evaluating *MiniCee* expressions under stores are defined according to the ANSI-C standard and are omitted.

Store Transformation. We overload the $\mapsto$ operator as in previous chapters. For any $z \in \mathbb{N}$, $\mathtt{1}^z$ denotes the tuple of z `1`'s. As before, for any *MiniCee* command `c` and stores σ and σ', we write $\{\sigma\}$ `c` $\{\sigma'\}$ to mean that σ is transformed to σ' by the execution of `c`. We define $\{\sigma\}$ `c` $\{\sigma'\}$ via induction on the structure of `c`, as shown in Figure 6.6. Straightforward rules such as sequential, parallel, and parameter instantiation have been omitted. See previous chapters for examples of similar rules.

6.5.3 Concrete Code

ShadowVisor implemented in *MiniCee* uses a 2-level PT scheme. The guest page table entries (gPDE and gPTE) have been abstracted using the havoc abstraction and the shadow Page Directory Table and shadow Page Tables (`s_pdt` and `s_pt`) are modeled using the 2-level *MiniCee* parametric array.

The ShadowVisor implementation in *MiniCee* consists of the following three functions: `shadow_new_context`, `shadow_invalidate_page`, and `shadow_page_fault` and the `adversary` interface. To keep the refinement proof focused on the relevant details, we elide a number of details. For example, we ignore typing since all types are unsigned thirty-two bit vectors. Function calls have been inlined, which in this case is possible since all functions are non-recursive. The for loop in *shadow_new_context* is syntactic sugar for an unrolled loop. Unrolling is feasible since the loop executes for a finite number of iterations. The `elseif` construct is syntactic sugar for a nested conditional. Direct assignments to `s_pt` are treated as pointer assignments and hence are semantic NOPS while assignments to its elements are considered array assignments. This differentiation is the key to our arrays-as-memory abstraction.

We ignore declarations in the syntax and declare variables as follows. Let s_pdt be the top-level array P.F and let s_pt be the array P[i_1].P.F. Let n_1 be 1024 and n_2 be 1024. Let gPDE and gPTE be integer variables. Let i_pdt and i_pt be i_1 and i_2, respectively. The $0x$ prefix signifies a hexadecimal value. Let PFERR_WR_MASK, PFERR_PRESENT, CANCEL, INJECT be uniquely defined natural numbers. The particular values are unimportant. Let PRESENT and PSE be the PRESENT and PSE flags. Let PMEMLIMIT be the physical memory limit beyond which hypervisor code is stored and SIZE_4k be 2^{12}.

The ShadowVisor Adversary is modeled in Figure 6.7 as a parallel composition of functions calls with non-deterministic parameters. ShadowVisor's Page Fault Handler is modeled in Figure 6.8. This handler extracts indexes from the passed control register, cr2, and checks the PRESENT and PSE flags and the physical memory limit before synchronizing the guest page table with the shadow page table. ShadowVisor's Page Invalidation Handler is modeled in Figure 6.9. This handler extracts indexes from cr2, checks the PRESENT and PSE flags, and invalidates pages by clearing all bits to zero. ShadowVisor's New Context Load Handler, shown in Figure 6.10, clears all entries of the shadow page directory table.

```
    shadow_new_context(*) ||
    shadow_invalidate_page(*) ||
    shadow_page_fault(*, *)
```

Figure 6.7: ShadowVisor Adversary

6.6 Towards Refinement

We state a refinement theorem and sketch the key details of the proof. Informally, our refinement theorem relates each step in the execution of the concrete system with a step of execution in the abstract system. Formally, the theorem is a weak simulation where

```
u32 shadow_page_fault(u32 cr2, u32 error_code){
  u32 *s_pt;
  u32 gPDE = *;
  u32 gPTE = *;

  u32 i_pdt = (cr2 >> 22);
  u32 i_pt  = ((cr2 & 0x003FFFFF) >> 12);

  if((s_pdt[i_pdt] & PRESENT) && !(s_pdt[i_pdt] & PSE)){
    s_pt = s_pdt[i_pdt];
  }

  if(!(error_code & PFERR_PRESENT)){
    if( ((gPDE & PRESENT) && (gPDE & PSE)) ||
        ((gPDE & PRESENT) && (!(gPDE & PSE)) && (gPTE & PRESENT))) {

      if(gPDE & PSE){
        if((gPDE & (~(SIZE_4K-1))) + SIZE_4M < PMEMLIMIT){
          s_pdt[i_pdt] = gPDE;
        }

      } else {
        s_pdt[i_pdt] = ((s_pdt[i_pdt] & (~(SIZE_4K-1))) &
                       (~(SIZE_4K-1))) | (gPDE & (SIZE_4K-1));

        if((gPTE & (~(SIZE_4k-1))) + SIZE_4K < PMEMLIMIT){
          s_pt[i_pt] = gPTE;
        }

      }
      return CANCEL;
    } else {
      return INJECT;
    }
  }else if (error_code & PFERR_WR_MASK){
    return INJECT;
  }else{
    return INJECT;
  }
}
```

Figure 6.8: ShadowVisor Page Fault Handler

```
u32 shadow_invalidate_page(u32 address){
  u32 *s_pt;
  u32 gPDE = *;
  u32 gPTE = *;

  u32 i_pdt = (address >> 22);
  u32 i_pt  = ((address & 0x003FFFFF) >> 12);

  if( (s_pdt[i_pdt] & PRESENT) && !(s_pdt[i_pdt] & PSE)) {
    s_pt = s_pdt[i_pdt];
  }

  if( !( s_pdt[i_pdt] & _PAGE_PRESENT) )
    return 0;

  if( !(gPDE & _PAGE_PRESENT) ){
     s_pdt[i_pdt] = 0;
  }else{
    if( ((gPDE & PSE) && !( s_pdt[i_pdt] & PSE)) ||
        (!(gPDE & PSE) && ( s_pdt[i_pdt] & PSE)) ){
      s_pdt[i_pdt] = 0;
    }else{
      if(s_pt){
        s_pt[i_pt] = 0;
      }else{
        s_pdt[i_pdt] = 0;
      }
    }
  }
  return 0;
}
```

Figure 6.9: ShadowVisor SPT Invalidation

```
u32 shadow_new_context(u32 guest_CR3){

  for (u32 i= 0; i < 1024; i++) {
    s_pdt[i] = 0;
  }

  return s_pdt;
}
```

Figure 6.10: ShadowVisor New Context Load

guarded commands are the atomic unit of execution in the abstract language and functions are the atomic unit of execution in the concrete language [20]. Note that it is reasonable to consider functions and guarded commands as atomic execution blocks because the system implementation disables interrupts when the hypervisor executes, making functions non-interruptable. The theorem requires two relationships be established: 1) the whole-array operations of the abstract program are related to the adversary-driven one-row-at-a-time execution of the concrete system, and 2) the stores of the abstract system are related to the stores of the concrete system.

We develop the first relationship by formalizing the following intuition: each function in the concrete code (excluding shadow_new_context) updates, at most, one row in the page directory table and, at most, one row in a page table. This behavior can be simulated by a trace of the abstract system where the adversary chooses the corresponding row and sets the adversary choice column, *ADV*, to true while setting all other rows of the adversary choice column to false. The key being that we have augmented the abstract system with a special indicator column, *ADV*, whereby the adversary selects the particular row to be updated by the whole-array operation rather than updating all rows at once.

The remaining discrepancy between the concrete and abstract programs, which is addressed with the abstraction function defined below, is that the stores of the abstract and concrete systems are named differently. Despite the apparent differences, there exists an exact bit-to-bit correspondence whereby a concrete state can be mapped into an abstract state. For example, the indices in the concrete programs are computed from the adversary input, while the indices in the abstract program are enumerated as a part of whole-array operations. Nonetheless, we show that the same indices are computed. A parallel argument can be made with respect to the arrays used in the concrete and abstract programs. We formalize this reasoning in the following theorem.

Theorem 17. *(Refinement) Let $A = adversary_A;shadow\_page\_fault_A$ and let $C = adversary;shadow\_page\_fault$ as defined in Figures 6.7, 6.8, and 6.4. Let $Init \in \mathsf{GSF}$ be any generic state formula of the form φ where $\varphi \in \mathsf{GSF}$. Let $M_A = M(A, Init)$ and $M_C = M(C, Init)$ be the transition systems induced by executing A and C, respectively. Then $M_C \preceq M_A$.*

Proof. Let $M_C = (\mathcal{S}_C, I_C, \mathcal{T}_C, \mathcal{L}_C)$ and $M_A = (\mathcal{S}_A, I_A, \mathcal{T}_A, \mathcal{L}_A)$ be Kripke structures over sets of atomic propositions AP_C and AP_A induced by executing C and A such that $\mathsf{AP}_A \subseteq \mathsf{AP}_C$.

We propose the following abstraction function H and show that it is a satisfies the conditions of simulation:

$$H(\sigma_C, \sigma_A) \Leftrightarrow$$

$$\left\{ \begin{array}{lcl}
\forall X \in \otimes(\sigma_1^n) \centerdot \sigma_C^{U32}(gPDE)(PRESENT) & = & \sigma_A^P(X)(gPRESENT) \\
\wedge\ \forall X \in \otimes(\sigma_1^n) \centerdot \sigma_C^{U32}(gPDE)(PSE) & = & \sigma_A^P(X)(gPSE) \\
\wedge\ \forall X \in \otimes(\sigma_1^n) \centerdot \sigma_C^{U32}(gPDE)(ADDR) & = & \sigma_A^P(X)(gADDR) \\
\wedge\ \forall X \in \otimes(\sigma_1^n) \centerdot \sigma_C^{spdt}(X)(PRESENT) & = & \sigma_A^P(X)(sPRESENT) \\
\wedge\ \forall X \in \otimes(\sigma_1^n) \centerdot \sigma_C^{spdt}(X)(PSE) & = & \sigma_A^P(X)(sPSE) \\
\wedge\ \forall X \in \otimes(\sigma_1^n) \centerdot \sigma_C^{spdt}(X)(ADDR) & = & \sigma_A^P(X)(sADDR) \\
\wedge\ \forall X \in \otimes(\sigma_{1,2}^n) \centerdot \sigma_C^{U32}(gPTE)(PRESENT) & = & \sigma_A^P(X)(gPTE\_PRESENT) \\
\wedge\ \forall X \in \otimes(\sigma_{1,2}^n) \centerdot \sigma_C^{U32}(gPTE)(ADDR) & = & \sigma_A^P(X)(gPTE\_ADDR) \\
\wedge\ \forall X \in \otimes(\sigma_{1,2}^n) \centerdot \sigma_C^{spt}(X)(PRESENT) & = & \sigma_A^P(X)(sPTE\_PRESENT) \\
\wedge\ \forall X \in \otimes(\sigma_{1,2}^n) \centerdot \sigma_C^{spt}(X)(ADDR) & = & \sigma_A^P(X)(sPTE\_ADDR)
\end{array} \right\}$$

Recall the conditions **C1–C3** in Definition 6 in Chapter 3 for simulation.

C1 holds because the atomic propositions are formulas that do not mention the deleted columns ADV and hence are equal;

C2 analogous to **C1**;

C3 Consider the set of traces of M_C, denoted $T(M_C)$. Let $L2C$ be a function that maps line numbers to their respective commands. Let $\sigma_C \in \mathcal{S}_C$ and $\sigma_A \in \mathcal{S}_A$. Let $t_C \in T(M_C)$ be an arbitrary trace such that $\{\sigma_C\}\ L2C(t_C)\ \{\sigma'_C\}$.

To show:

$$\forall t_C \in T(M_C) \,\centerdot\, \exists t_A \in T(M_A) \,\centerdot\, H(\sigma_A, \sigma_C) \bigwedge \{\sigma_A\}\ L2C(t_A)\ \{\sigma'_A\} \bigwedge H(s'_A, s'_C)$$

The following abstractions reduce the number of concrete traces that need be considered: (1) *MiniCee* abstracts away pointer details by converting pointer operations to NOPS, (2) our adversary model (nondeterministic choice) is unchanged by return values and our initial and security properties do not refer to return values allowing us to treat returns as NOPS, and (3) the index variables and the gPDE and gPTE variables are not mentioned in the initial or security properties and hence are not relevant for simulation. After applying these abstractions, we need only consider those traces that modify key system data structures. The following proof ignores stutter steps.

Let [...k] of a program P be the sequence of commands that are executed up to line k. It is sufficient to consider only distinct sequences including [...19], [...24], and [...27]:

Case [...19]. Let $\texttt{PMEMLIMIT} \in \texttt{K}$ and $\texttt{SIZE\_4M} \in \texttt{K}$ be constants.

Consider $\{\sigma_C\}$ *adversary*;$L2C$([...19]) $\{\sigma'_C\}$, by Figure 6.6, we know that $\exists \texttt{t1}, \texttt{t2} \in$

$U32$ such that:

$$\langle \texttt{cr2}, \sigma'_C \rangle \to \texttt{t1} \wedge$$
$$\langle \texttt{gPDE}, \sigma'_C \rangle \to \texttt{t2} \wedge$$
$$\langle \texttt{i}_{\texttt{pdt}}, \sigma'_C \rangle \to (\texttt{t1} >> 22) \wedge$$
$$\langle \texttt{i}_{\texttt{pt}}, \sigma'_C \rangle \to ((\texttt{t2} \ \& \ \texttt{0x003FFFFF}) >> 12) \wedge$$
$$\langle \texttt{t2} \ \& \ \texttt{PRESENT}, \sigma'_C \rangle \to \top \wedge \langle \texttt{t2} \ \& \ \texttt{PSE}, \sigma'_C \rangle \to \top \wedge$$
$$\langle ((\texttt{t2} \ \& \ (\sim (\texttt{SIZE\_4K} - 1))) + \texttt{SIZE\_4M} < \texttt{PMEMLIMIT}), \sigma'_C \rangle \to \top$$

And as a result:

$$\sigma'_C = \sigma[\sigma^{spdt} \mapsto \sigma^{spdt}[(\texttt{t1} >> 22) \mapsto \texttt{t2}]] \tag{6.1}$$
$$[\sigma^{U32} \mapsto \sigma^{U32}[\texttt{cr2} \mapsto \texttt{t1}, \texttt{gPDE} \mapsto \texttt{t2}, \tag{6.2}$$
$$\texttt{i}_{\texttt{pdt}} \mapsto (\texttt{t1} >> 22), \texttt{i}_{\texttt{pt}} \mapsto ((\texttt{t2} \ \& \ \texttt{0x003FFFFF}) >> 12)]] \tag{6.3}$$

Consider $t_A \in adversary_A; shadow\_page\_fault_A(1024, 1024)$ where $\{\sigma_A\}$ $adversary_A$ $\{\sigma''_A\}$ sets the columns of PDT as follows:

$$\langle \texttt{PDT}[(\texttt{t1} >> 22)].\texttt{F}[\texttt{ADV}], \sigma''_A \rangle \to \top \wedge$$
$$\langle \texttt{PDT}[(\texttt{t1} >> 22)].\texttt{F}[\texttt{gPDE}], \sigma''_A \rangle \to \texttt{t2} \wedge$$
$$\forall i \in [1, 1024] \,.\, i \neq (t1 >> 22) \Rightarrow \langle \texttt{PDT[i].F[ADV]}, \sigma''_A \rangle \to \perp \wedge$$
$$\forall (i, j) \in \otimes(\sigma^n_{A(1,2)}) \,.\, \langle \texttt{PDT[i].PT[j].F[ADV]}, \sigma''_A \rangle \to \perp \wedge$$
$$\langle \texttt{PDT}[(\texttt{t1} >> 22)].\texttt{F}[\texttt{gPRESENT}], \sigma''_A \rangle \to \top \wedge$$
$$\langle \texttt{PDT}[(\texttt{t1} >> 22)].\texttt{F}[\texttt{gPSE}], \sigma''_A \rangle \to \top \wedge$$
$$\langle (\texttt{PDT}[(\texttt{t1} >> 22)].\texttt{F}[\texttt{gADDR}] < \texttt{PMEMLIMIT} - \texttt{SIZE\_4M}), \sigma''_A \rangle \to \top$$

Next, consider $\{\sigma''_A\}$ $shadow\_page\_fault_A$ $\{\sigma'_A\}$ where by Figure 6.3 we arrive at:

$$\sigma'_A = \sigma[\sigma^P \mapsto \sigma^P[(\mathtt{t1} >> 22) \mapsto \mathtt{t2}]] \quad (6.4)$$

$$[\sigma^{U32} \mapsto \sigma^{U32}[\mathtt{cr2} \mapsto \mathtt{t1}, \mathtt{gPDE} \mapsto \mathtt{t2}]] \quad (6.5)$$

and hence $H(s'_A, s'_C)$.

Case [...24]. This case follows analogously to the previous.

Case [...27]. We construct t_A from $adversary; shadow\_page\_fault$ as follows:

Let $i_1 = (cr2 >> 22)$ and $i_2 = ((cr2\&0x003FFFFF) >> 12)$. Consider the trace of $adversary$ where $PDT[i_1].F[gPDE] := *$, $PDT[i_1].PT[i_2].F[gPTE] := *$, and for all $i_1! = (cr2 >> 22)$ and for all $i_2! = ((cr2\&0x003FFFFF) >> 12)$ the values are left unchanged (i.e., $*$ evaluates to the current value). By Figure 6.3, $H(s'_A, s'_C)$.

□

6.7 Conclusion

We show that Small Model Theorems apply to a detailed model of ShadowVisor's implementation by proving a refinement theorem between ShadowVisor defined in *MiniCee* and ShadowVisor defined in $PGCL^{++}$. We formally defined the syntax and semantics of *MiniCee* and added new programming constructs to $PGCL^{+}$ to allow previously forbidden data flows. We updated the SMTs to account for new data flows and programming constructs. A refinement mapping between ShadowVisor's high-level $PGCL^{++}$ model and more detailed *MiniCee* model was developed. We stated a refinement theorem and sketched a proof.

Chapter 7

Related Work

We describe related work in parametric verification for correctness, parametric verification for security, model checking for security, bug finding using model checking, operating system verification, QBF solving, summarization, and refinement.

7.1 Parametric Verification for Correctness

Parametric verification has been applied successfully to a wide variety of problems, including cache coherence [29, 27, 77], bus arbitration [31], and resource allocation [26].

The general parametric model checking problem is undecidable [4, 73]. However, restricted versions of the problem, typically tailored to cache coherence protocols, yield decision procedures [38, 28]. These decision procedures are more efficient [30, 25, 27] when the problem is restricted to a greater degree.

We consider a family of data-independent systems [78] for which efficient decision procedures exist [55, 54]. These procedures enable verification of *all* finite parameter instantiations by considering only a *finite* number of such instantiations. However, all these approaches are either not expressive enough to model reference monitors, or are less efficient than our technique. In particular, the forms of whole-array (i.e., `for`) operations supported by PGCL are critical for modeling and verifying the security of reference monitor-based systems that operate on unbounded data structures. Existing formalisms for parameterized verification of data-independent systems either do not allow whole-array operations [54], or restrict them to a reset or copy operation that updates array elements to fixed values [55]. Neither case can model our adversary. Our whole-array operations allow atomic updates across the array, a necessary feature for modeling reference monitors that is missing in Emerson and Kahlon [25].

Pnueli et al [62], Arons et al., [6], and Fang et al. [33] investigate finite bounded-data systems which support stratified arrays that map to Booleans, a notion similar to our hierarchical arrays. However, they consider a restricted logic that allows for safety properties and a limited form of liveness properties, referred to as “response properties.” In contrast, we consider both safety and more expressive liveness properties that can include both next state and until operators in addition to the forall operator. Moreover, the cutoffs of their small model theorems are a function of the type signatures, number of quantified index variables, and other factors. When instantiated with the same types used in our languages, their small model theorems have larger cutoffs than our own. By focusing on the specific case of address translation systems and address separation properties, we are able to arrive at smaller cutoffs.

7.2 Parametric Verification for Security

Lowe et al. [57] study parametric verification of authentication properties of network protocols. Roscoe and Broadfoot [65] apply data independence techniques to model check security protocols. Durgin et al. [24] show that small model theorems do not exist for a general class of security protocols. Millen [58] presents a family of protocols such that for any $k \in \mathbb{N}$, a member of the family has a cutoff greater than k. We present the first small model theorems for system security.

7.3 Model Checking System Security

Several projects have looked at applying model checking for verifying security of software. Guttman et al. [41] employ model checking to verify information-flow properties of the SELinux system. In addition, Lie et al. verify XOM [56], a hardware-based approach for tamper-resistance and copy-resistance, using the Murφ model checker. Mitchell et al. [60, 59] use Murφ to verify the correctness of security protocol specifications. The Murφ tool has its own modeling language. Our approach to parametric verification is amenable to verification via other model checkers, such as SPIN [47], TLA+ [53] and SMV [1]. Rather than model checking a particular system, the focus of this thesis is on development of a general framework for direct verification of C source code using a software model checker and specialized abstractions that apply to a broad class of systems.

7.4 Bug Finding

A number of projects use software model checking and static analysis to find errors in source code, without a specific attacker model. Some of these projects [42, 18, 79] target a general class of bugs. Others focus on specific types of errors, e.g., Kidd et al. [49] detect atomic set serializability violations, while Emmi et al. [32] verify correctness of reference counting implementation. All these approaches require abstraction, e.g., random isolation [49] or predicate abstraction [32], to handle source code, and are unsound and/or

[1] `http://www.cs.cmu.edu/~modelcheck/smv.html`

incomplete. In contrast, our focus is on bug detection and verification in the presence of a CSI-adversary with precisely defined capabilities.

7.5 Operating System Verification

Prior work [61, 66, 71, 44, 40, 37, 13, 69] has explored the problem of verifying the design of secure systems. These works are similar to our own in spirit, but differ in the methods applied. They suggest an approach where properties are manually proven using a logic and without an explicit adversary model. We focus on model checking source code with an explicit adversary model.

A number of groups – Walker et al. [76] were one of the first – have used theorem proving to verify security properties of OS implementations. For example, Heitmeyer et al. [45] use PVS to verify correctness of a data separation kernel, while Klein et al. [50] use Isabelle to prove functional correctness properties of the L4 microkernel. In contrast, we use model checking to automatically verify semantic security properties of systems that enforce protections with unbounded data structures.

Recently, there has been a flurry of work using theorem proving to verify hypervisors. Barthe et al. [11] formalized an idealized model of a hypervisor in the Coq proof assistant and established formal guarantees of isolation. Alkassar et al. [2, 1] and Baumann et al. [12] annotated the C code of a hypervisor, including shadow paging code, with Hoare-like annotations at a total cost of two person-years of effort and then utilized the VCC verifier to prove correctness properties. In contrast, we use software model checking and sound abstractions to automatically check security properties of systems, including hypervisors, that enforce address space separation with unbounded data structures.

7.6 QBF Solving

Developing efficient QBF solvers is an active area of research. Even though QBF validity is PSPACE-complete, a number of efficient tools have been developed in recent years. Notable tools are sKizzo [72], QuBE [63], and GhostQ [51]. QBFs solvers have been applied to solve a variety of problems, including bounded model checking [22]. However, to our knowledge, we are the first to employ QBF solvers to improve the scalability of verification by detecting havoc functions.

7.7 Program Summarization

Havoc abstraction is a form of program summarization, an active area of research in software verification. Inter-procedural model checkers, such as BEBOP [8], perform function summarization. In addition, several projects, such as LoopFrog [52], have looked at summarization of loops. Our focus is on detecting if a function can return arbitrary values, rather than computing general summaries.

Chapter 8

Conclusions

Systems software forms the foundation of security for platforms including desktop, cloud, and mobile. Despite their ubiquity, these security-critical systems regularly suffer from serious vulnerabilities. In this thesis, we introduce and formally define the problem of *semantic security verification*: verifying that every execution of a program running in parallel with a system-specific adversary satisfies a security property.

We develop an approach to semantic security verification that utilizes model checking to enable automated verifiable security guarantees for a wide range of systems software. Central to the effectiveness of our framework are novel abstractions that significantly improve the scalability of model checking secure systems. The abstractions exploit structure common to secure systems and the non-determinism inherent in adversary models to reduce the complexity of verification.

We develop three abstractions: *CSI-adversary* abstraction, *Small Model Analysis*, and *Havoc* abstraction. We prove soundness theorems to demonstrate that no attacks are missed by the abstractions. We prove a completeness theorem that provides the theoretical basis for our zero false positive rate. Finally, we sketch a proof of a refinement theorem that carries our results to the source level.

We perform case studies on hypervisors designed to enforce a variety of security properties in the presence of adversary-controlled guest operating systems. Our framework accomplishes a variety of previously intractable results: we identified unknown vulnerabilities in two research hypervisors and successfully model checked their code after fixing the vulnerabilities. We also successfully model check the design of two complex Xen variants.

Appendix A

Proofs of Small Model Theorems

A.1 Proofs

This appendix contains lemmas used in the small model theorems and their associated proofs.

A.1.1 Introductory Lemmas

First we prove that projection does not affect the evaluation of expressions, as expressed by the following lemma.

Lemma 18. *Let* $\mathsf{e} \in \mathsf{E}$, $t \in \mathbb{B}$, *and* σ *be any store. Then:*

$$\langle \mathsf{e}, \sigma \rangle \to t \Leftrightarrow \forall i \in [1, \sigma^n] \centerdot \langle \mathsf{e}, \sigma \downharpoonleft i \rangle \to t$$

$$\Leftrightarrow \exists i \in [1, \sigma^n] \centerdot \langle \mathsf{e}, \sigma \downharpoonleft i \rangle \to t$$

Proof. Follows from the fact that e depends only on Boolean variables, and the following observation:

$$\forall i \in [1, \sigma^n] \,.\, \sigma^B = (\sigma \downharpoonright i)^B$$

$\square$

We now state a fact, and prove a lemma about the “Unroll” rule.

Fact 2. *Suppose* $\{\sigma\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)(\mathtt{i} \gg \mathtt{j})\ \{\sigma'\}$*. Then the only possible difference between* σ^P *and* σ'^P *is in the* $\lceil \mathtt{j} \rceil$*-th row. Also, since* $\mathtt{c} \in \widehat{\mathtt{C}}$*, we have* $\sigma^B = \sigma'^B$*. Thus, the only possible difference between* σ *and* σ' *is in the* $\lceil \mathtt{j} \rceil$*-th row of* σ'^P*.*

Lemma 19. *Suppose that* $\{\sigma\}\ \mathtt{for}\ i : \mathtt{P_{n,q}}\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2\ \{\sigma'\}$*, and* $\sigma^n = N$*. Let* $\sigma_1, \ldots, \sigma_{N+1}$ *be any set of stores satisfying the premise of the “Unroll” rule of Figure 3.5. Let* $j, k \in \mathbb{N}$ *be any two natural numbers such that* $1 \leq j \leq k \leq N+1$*. Then:*

$$\forall l \in [1, N] \,.\, (l < j \vee l \geq k) \Rightarrow (\sigma_j \downharpoonright l = \sigma_k \downharpoonright l)$$

In other words, the only possible difference between σ_j *and* σ_k *are in rows* j *through* $k-1$ *of* σ_j^P *and* σ_k^P*.*

Proof. By induction on $k - j$.

Case 1. Let $k - j = 0$, i.e., $j = k$. In this case, the claim holds trivially, since

$$\forall l \in [1, N] \,.\, \sigma_j \downharpoonright l = \sigma_k \downharpoonright l$$

Case 2. Suppose the claim holds for $k - j = x$. Suppose $k - j = x + 1$, i.e, $(k-1) - j = x$. Therefore, by the inductive hypothesis:

$$\forall l \in [1, N] \,.\, l < j \vee l \geq k-1 \Rightarrow \sigma_j \downharpoonright l = \sigma_{k-1} \downharpoonright l$$

In other words, the only possible difference between σ_j and σ_{k-1} are in rows j through $k-2$ of σ_j^P and σ_{k-1}^P. Now, from the premise of the "Unroll" rule, we know that $\{\sigma_{k-1}\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)(\mathtt{i} \gg \mathtt{k}')\ \{\sigma_k\}$, where $\lceil \mathtt{k}' \rceil = k-1$. Therefore, by Fact 2, the only possible difference between σ_{k-1} and σ_k is in the $(k-1)$-th row of σ_{k-1}^P and σ_k^P. Thus, the only possible difference between σ_j and σ_k are in rows j through $k-1$ of σ_j^P and σ_k^P, which is what we want to prove. □

A.1.2 Store Projection Lemmas

We now present a series of lemmas that explore the relationship between store projection and different types of formulas. These lemmas will be used later to prove our small model theorems. The first lemma states that a store σ satisfies a basic proposition π iff every projection of σ satisfies π.

Lemma 20. *Let* $\pi \in \mathsf{BP}$ *and* σ *be any store. Then:*

$$\sigma \models \pi \Leftrightarrow \forall i \in [1,\sigma^n] \centerdot \sigma \downharpoonleft i \models \pi \Leftrightarrow \exists i \in [1,\sigma^n] \centerdot \sigma \downharpoonleft i \models \pi$$

Proof. Follows from the fact that π depends only on Boolean variables, and the following observation:

$$\forall i \in [1,\sigma^n] \centerdot \sigma^B = (\sigma \downharpoonleft i)^B$$

□

The next lemma states that a store σ satisfies an universally quantified formula π iff every projection of σ satisfies π.

Lemma 21. *Let* $\pi = \forall \mathtt{i} \centerdot \pi'$, $\pi' \in \mathsf{PP}(\mathtt{i})$ *and* σ *be any store. Then:*

$$\sigma \models \pi \Leftrightarrow \forall i \in [1,\sigma^n] \centerdot \sigma \downharpoonleft i \models \pi$$

Proof. By Definition 3, we know that:

$$\sigma \models \pi \Leftrightarrow \forall i \in [1, \sigma^n] \,.\, \sigma \downharpoonleft i \models \pi'[\mathtt{i} \mapsto 1]$$

$$\triangleright \text{ since } ((\sigma \downharpoonleft i) \downharpoonleft 1) = (\sigma \downharpoonleft i)$$

$$\Leftrightarrow \forall i \in [1, \sigma^n] \,.\, (\sigma \downharpoonleft i) \downharpoonleft 1 \models \pi'[\mathtt{i} \mapsto 1]$$

$$\triangleright \text{ let } j \text{ be a fresh variable}$$

$$\Leftrightarrow \forall i \in [1, \sigma^n] \,.\, \forall j \in \{1\} \,.\, (\sigma \downharpoonleft i) \downharpoonleft j \models \pi'[\mathtt{i} \mapsto 1]$$

$$\triangleright \text{ since } (\sigma \downharpoonleft i)^n = 1$$

$$\Leftrightarrow \forall i \in [1, \sigma^n] \,.\, \forall j \in [1, (\sigma \downharpoonleft i)^n] \,.\, (\sigma \downharpoonleft i) \downharpoonleft j \models \pi'[\mathtt{i} \mapsto 1]$$

$$\triangleright \text{ let } \mathtt{j} \text{ be a fresh variable}$$

$$\Leftrightarrow \forall i \in [1, \sigma^n] \,.\, \forall j \in [1, (\sigma \downharpoonleft i)^n] \,.\, (\sigma \downharpoonleft i) \downharpoonleft j \models \pi'[\mathtt{i} \mapsto \mathtt{j}][\mathtt{j} \mapsto 1]$$

$$\triangleright \text{ by Definition 3}$$

$$\Leftrightarrow \forall i \in [1, \sigma^n] \,.\, \sigma \downharpoonleft i \models \forall \mathtt{j} \,.\, (\pi'[\mathtt{i} \mapsto \mathtt{j}])$$

By alpha renaming, we know that $\forall \mathtt{j} \,.\, (\pi'[\mathtt{i} \mapsto \mathtt{j}])$ is equivalent to π. This give us our desired result. □

The next lemma states that a store σ satisfies an existentially quantified formula π iff some projection of σ satisfies π.

Lemma 22. *Let* $\pi = \exists \mathtt{i} \,.\, \pi'$ *and* $\pi' \in \mathsf{PP}(\mathtt{i})$ *and* σ *be any store. Then:*

$$\sigma \models \pi \Leftrightarrow \exists i \in [1, \sigma^n] \,.\, \sigma \downharpoonleft i \models \pi$$

Proof. By Definition 3, we know that:

$$\sigma \models \pi \Leftrightarrow \exists i \in [1, \sigma^n] \,\textbf{.}\, \sigma \downharpoonleft i \models \pi'[\mathtt{i} \mapsto 1]$$

$$\rhd \text{ since } ((\sigma \downharpoonleft i) \downharpoonleft 1) = (\sigma \downharpoonleft i)$$

$$\Leftrightarrow \exists i \in [1, \sigma^n] \,\textbf{.}\, (\sigma \downharpoonleft i) \downharpoonleft 1 \models \pi'[\mathtt{i} \mapsto 1]$$

$$\rhd \text{ let } j \text{ be a fresh variable}$$

$$\Leftrightarrow \exists i \in [1, \sigma^n] \,\textbf{.}\, \exists j \in [1, 1] \,\textbf{.}\, (\sigma \downharpoonleft i) \downharpoonleft j \models \pi'[\mathtt{i} \mapsto 1]$$

$$\rhd \text{ since } (\sigma \downharpoonleft i)^n = 1$$

$$\Leftrightarrow \exists i \in [1, \sigma^n] \,\textbf{.}\, \exists j \in [1, (\sigma \downharpoonleft i)^n] \,\textbf{.}\, (\sigma \downharpoonleft i) \downharpoonleft j \models \pi'[\mathtt{i} \mapsto 1]$$

$$\rhd \text{ let } \mathtt{j} \text{ be a fresh variable}$$

$$\Leftrightarrow \exists i \in [1, \sigma^n] \,\textbf{.}\, \exists j \in [1, (\sigma \downharpoonleft i)^n] \,\textbf{.}\, (\sigma \downharpoonleft i) \downharpoonleft j \models \pi'[\mathtt{i} \mapsto \mathtt{j}][\mathtt{j} \mapsto 1]$$

$$\rhd \text{ by Definition 3}$$

$$\Leftrightarrow \exists i \in [1, \sigma^n] \,\textbf{.}\, \sigma \downharpoonleft i \models \exists \mathtt{j} \,\textbf{.}\, (\pi'[\mathtt{i} \mapsto \mathtt{j}])$$

By alpha renaming, we know that $\exists \mathtt{j} \,\textbf{.}\, (\pi'[\mathtt{i} \mapsto \mathtt{j}])$ is equivalent to π. This give us our desired result. □

The next lemma states that a store σ satisfies an existential state formula φ iff some projection of σ satisfies φ.

Lemma 23. *Let* $\varphi \in \mathsf{ESF}$ *and* σ *be any store. Then:*

$$\sigma \models \varphi \Leftrightarrow \exists i \in [1, \sigma^n] \,\textbf{.}\, \sigma \downharpoonleft i \models \varphi$$

Proof. From Figure 3.6, we consider three sub-cases:

Case 1. $\varphi \in \mathsf{BP}$. Our result follows from Lemma 29.

Case 2. $\varphi = \exists \mathtt{i} \,\mathtt{.}\, \pi$ and $\pi \in \mathsf{PP}(\mathtt{i})$. Our result follows from Lemma 30.

Case 3. $\varphi = \pi \wedge \exists \mathtt{i} \,\mathtt{.}\, \pi'$ such that $\pi \in \mathsf{BP}$, and $\pi' \in \mathsf{PP}(\mathtt{i})$. For the forward implication, by Definition 3:

$$\sigma \models \varphi \Rightarrow \sigma \models \pi \bigwedge \sigma \models \exists \mathtt{i} \,\mathtt{.}\, \pi'$$

$\triangleright$ by Lemma 29 and Lemma 30

$$\Rightarrow \forall j \in [1, \sigma^n] \,\mathtt{.}\, \sigma \downharpoonleft j \models \pi \bigwedge \exists j \in [1, \sigma^n] \,\mathtt{.}\, \sigma \downharpoonleft j \models \exists \mathtt{i} \,\mathtt{.}\, \pi'$$

$\triangleright$ playing around with $\wedge$, $\forall$, and $\exists$

$$\Rightarrow \exists j \in [1, \sigma^n] \,\mathtt{.}\, \sigma \downharpoonleft j \models \pi \wedge \sigma \downharpoonleft j \models \exists \mathtt{i} \,\mathtt{.}\, \pi'$$

$\triangleright$ again by Definition 3

$$\Rightarrow \exists j \in [1, \sigma^n] \,\mathtt{.}\, \sigma \downharpoonleft j \models \pi \wedge \exists \mathtt{i} \,\mathtt{.}\, \pi'$$

By alpha renaming j to i, we get our desired result. For the reverse implication:

$$\exists i \in [1, \sigma^n] \,\mathtt{.}\, \sigma \downharpoonleft i \models \pi \wedge \exists \mathtt{i} \,\mathtt{.}\, \pi'$$

$\triangleright$ by Definition 3

$$\Rightarrow \exists i \in [1, \sigma^n] \,\mathtt{.}\, \sigma \downharpoonleft i \models \pi \wedge \sigma \downharpoonleft i \models \exists \mathtt{i} \,\mathtt{.}\, \pi'$$

$\triangleright$ playing around with $\vee$ and $\exists$

$$\Rightarrow \exists i \in [1, \sigma^n] \,\mathtt{.}\, \sigma \downharpoonleft i \models \pi \bigwedge \exists i \in [1, \sigma^n] \,\mathtt{.}\, \sigma \downharpoonleft i \models \exists \mathtt{i} \,\mathtt{.}\, \pi'$$

$\triangleright$ by Lemma 29 and Lemma 30

$$\Rightarrow \sigma \models \pi \bigwedge \sigma \models \exists \mathtt{i} \centerdot \pi'$$

$\triangleright$ again by Definition 3

$$\Rightarrow \sigma \models \varphi$$

This completes our proof. $\square$

A.1.3 Store Projection and Command Lemmas

The following lemma states that if a store σ is transformed to σ' by executing a command $\mathtt{c}$, then every projection of σ is transformed to the corresponding projection of σ' by executing $\mathtt{c}$.

Lemma 24 (Command Projection)**.** *For any stores σ, σ' and any command $\mathtt{c} \in \mathtt{C}$:*

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \forall i \in [1, \sigma^n] \centerdot \{\sigma \downharpoonleft i\}\ \mathtt{c}\ \{\sigma' \downharpoonleft i\}$$

Proof. By induction on the structure of $\mathtt{c}$. We consider three subcases.

Case 1. $\mathtt{c} \triangleq \mathtt{b} := \mathtt{e}$. Suppose $\langle \mathtt{e}, \sigma \rangle \to t$. By Figure 3.5:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \sigma' = \sigma[\sigma^B \mapsto \sigma^B[\mathtt{b} \mapsto t]] \Rightarrow$$

$\triangleright$ by Definition 1 and Lemma 28, $\forall i \in [1, \sigma^n]$

$$\sigma' \downharpoonleft i = (\sigma \downharpoonleft i)[(\sigma \downharpoonleft i)^B \mapsto (\sigma \downharpoonleft i)^B[\mathtt{b} \mapsto t]] \wedge \langle \mathtt{e}, \sigma \downharpoonleft i \rangle \to t$$

$\triangleright$ again by Figure 3.5

$$\Rightarrow \forall i \in [1, \sigma^n] \centerdot \{\sigma \downharpoonleft i\}\ \mathtt{c}\ \{\sigma' \downharpoonleft i\}$$

Case 2. $\mathtt{c} \triangleq \mathtt{for\ i}\ : \mathtt{P_{n,q}}\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2$. Let $\sigma^n = N$. By Figure 3.5:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \exists \sigma_1, \ldots, \sigma_{N+1} \boldsymbol{.}$$

$$\sigma = \sigma_1 \wedge \sigma' = \sigma_{N+1} \wedge \forall \lceil \mathtt{j} \rceil \in [1,N] \boldsymbol{.} \{\sigma_{\lceil \mathtt{j} \rceil}\}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2(\mathtt{i} \gg \mathtt{j})\ \{\sigma_{\lceil \mathtt{j} \rceil + 1}\}$$

$\triangleright$ First, by Definition 2

$$\forall \lceil \mathtt{j} \rceil \in [1,N] \boldsymbol{.} \{\sigma_{\lceil \mathtt{j} \rceil} \downharpoonleft \lceil \mathtt{j} \rceil\}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2[\mathtt{i} \mapsto \mathtt{1}]\ \{\sigma_{\lceil \mathtt{j} \rceil + 1} \downharpoonleft \lceil \mathtt{j} \rceil\}$$

$\triangleright$ Second, by Lemma 19

$$\forall \lceil \mathtt{j} \rceil \in [1,N] \boldsymbol{.} ((\sigma \downharpoonleft \lceil \mathtt{j} \rceil = \sigma_{\lceil \mathtt{j} \rceil} \downharpoonleft \lceil \mathtt{j} \rceil) \wedge (\sigma' \downharpoonleft \lceil \mathtt{j} \rceil = \sigma_{\lceil \mathtt{j} \rceil + 1} \downharpoonleft \lceil \mathtt{j} \rceil))$$

$\triangleright$ Combining two previous facts

$$\forall \lceil \mathtt{j} \rceil \in [1,N] \boldsymbol{.} \{\sigma \downharpoonleft \lceil \mathtt{j} \rceil\}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2[\mathtt{i} \mapsto \mathtt{1}]\ \{\sigma' \downharpoonleft \lceil \mathtt{j} \rceil\}$$

$\triangleright$ Again by Figure 3.5

$$\forall \lceil \mathtt{j} \rceil \in [1,N] \boldsymbol{.} \{\sigma \downharpoonleft \lceil \mathtt{j} \rceil\}\ \mathtt{c}\ \{\sigma' \downharpoonleft \lceil \mathtt{j} \rceil\}$$

Case 3. $\mathtt{c} \triangleq \mathtt{c}_1; \mathtt{c}_2$. By Figure 3.5:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \exists \sigma'' \boldsymbol{.} \{\sigma\}\ \mathtt{c}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \mathtt{c}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\exists \sigma'' \boldsymbol{.} \forall i \in [1, \sigma^n] \boldsymbol{.} \{\sigma \downharpoonleft i\}\ \mathtt{c}_1\ \{\sigma'' \downharpoonleft i\} \wedge$$

$$\forall i \in [1, \sigma^n] \boldsymbol{.} \{\sigma'' \downharpoonleft i\}\ \mathtt{c}_2\ \{\sigma' \downharpoonleft i\} \Rightarrow$$

$\triangleright$ Swapping $\forall$ and $\exists$

$$\forall i \in [1,\sigma^n] \centerdot \exists \sigma'' \centerdot \{\sigma \downharpoonright i\}\ \mathsf{c}_1\ \{\sigma'' \downharpoonright i\} \wedge \{\sigma'' \downharpoonright i\}\ \mathsf{c}_2\ \{\sigma' \downharpoonright i\}$$

$$\triangleright \text{ Again by Figure 3.5}$$

$$\Rightarrow \forall i \in [1,\sigma^n] \centerdot \{\sigma \downharpoonright i\}\ \mathsf{c}\ \{\sigma' \downharpoonright i\}$$

This completes the proof. □

The next lemma states that if a store σ is transformed to σ′ by executing a guarded command gc, then every projection of σ is transformed to the corresponding projection of σ′ by executing gc.

Lemma 25 (Guarded Command Projection)**.** *For any stores* σ, σ' *and any guarded command* $\mathsf{gc} \in \mathtt{GC}$*:*

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \forall i \in [1,\sigma^n] \centerdot \{\sigma \downharpoonright i\}\ \mathsf{gc}\ \{\sigma' \downharpoonright i\}$$

Proof. We consider two cases.

Case 1. $\mathsf{gc} \triangleq \mathsf{e}\ ?\ \mathsf{c}_1 : \mathsf{c}_2$. By Figure 3.5:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma\rangle \rightarrow \textbf{true} \wedge \{\sigma\}\ \mathsf{c}\ \{\sigma'\} \Rightarrow$$

$$\triangleright \text{ By Lemma 28 and Lemma 33}$$

$$\forall i \in [1,\sigma^n] \centerdot \langle \mathsf{e}, \sigma \downharpoonright i\rangle \rightarrow \textbf{true} \wedge \forall i \in [1,\sigma^n] \centerdot \{\sigma \downharpoonright i\}\ \mathsf{c}\ \{\sigma' \downharpoonright i\}$$

$$\triangleright \text{ Since } \wedge \text{ distributes over } \forall$$

$$\Rightarrow \forall i \in [1,\sigma^n] \centerdot \langle \mathsf{e}, \sigma \downharpoonright i\rangle \rightarrow \textbf{true} \wedge \{\sigma \downharpoonright i\}\ \mathsf{c}\ \{\sigma' \downharpoonright i\}$$

$$\triangleright \text{ Again by Figure 3.5}$$

$$\Rightarrow \forall i \in [1,\sigma^n] \centerdot \{\sigma \downharpoonright i\}\ \mathsf{gc}\ \{\sigma' \downharpoonright i\}$$

Case 2. $\mathsf{gc} \triangleq \mathsf{gc}_1 \parallel \mathsf{gc}_2$. By Figure 3.5:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \{\sigma\}\ \mathsf{gc}_1\ \{\sigma'\} \vee \{\sigma\}\ \mathsf{gc}_2\ \{\sigma'\} \Rightarrow$$

$$\triangleright \text{ By inductive hypothesis}$$

$$\forall i \in [1,\sigma^n] \centerdot \{\sigma \downharpoonright i\}\ \mathsf{gc}_1\ \{\sigma' \downharpoonright i\} \bigvee$$

$$\forall i \in [1,\sigma^n] \centerdot \{\sigma \downharpoonright i\}\ \mathsf{gc}_2\ \{\sigma' \downharpoonright i\} \Rightarrow$$

$$\triangleright \text{ Playing around with } \vee \text{ and } \forall$$

$$\forall i \in [1,\sigma^n] \centerdot \{\sigma \downharpoonright i\}\ \mathsf{gc}_1\ \{\sigma' \downharpoonright i\} \vee \{\sigma \downharpoonright i\}\ \mathsf{gc}_2\ \{\sigma' \downharpoonright i\}$$

$$\triangleright \text{ Again by Figure 3.5}$$

$$\Rightarrow \forall i \in [1,\sigma^n] \centerdot \{\sigma \downharpoonright i\}\ \mathsf{gc}\ \{\sigma' \downharpoonright i\}$$

This completes the proof. □

A.1.4 Store Generalization Lemmas

We now present a series of lemmas that relate the execution semantics of PGCL to store generalization. The first lemma states that if a store σ is transformed to σ' by executing a command c, then every generalization of σ is transformed to the corresponding generalization of σ' by executing c.

Lemma 26 (Command Generalization)**.** *For any stores σ,σ' such that $\sigma^n = \sigma'^n = 1$, and any command $\mathsf{c} \in \mathtt{C}$:*

$$\{\sigma\}\ \mathsf{c}\ \{\sigma'\} \Rightarrow \forall k \in \mathbb{N} \centerdot \{\sigma \upharpoonright k\}\ \mathsf{c}\ \{\sigma' \upharpoonright k\}$$

Proof. By induction on the structure of $\mathtt{c}$. We consider three subcases.

Case 1. $\mathtt{c} \triangleq \mathtt{b} := \mathtt{e}$. Suppose $\langle \mathtt{e}, \sigma \rangle \to t$. By Figure 3.5:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \sigma' = \sigma[\sigma^B \mapsto \sigma^B[\mathtt{b} \mapsto t]] \Rightarrow$$

$$\triangleright \text{ by Definition 7 and Lemma 28, } \forall k \in \mathbb{N}$$

$$\sigma' \upharpoonright k = (\sigma \upharpoonright k)[(\sigma \upharpoonright k)^B \mapsto (\sigma \upharpoonright k)^B[\mathtt{b} \mapsto t]] \wedge \langle \mathtt{e}, \sigma \upharpoonright k \rangle \to t$$

$$\triangleright \text{ again by Figure 3.5}$$

$$\Rightarrow \forall k \in \mathbb{N} \,\boldsymbol{.}\, \{\sigma \upharpoonright k\}\ \mathtt{c}\ \{\sigma' \upharpoonright k\}$$

Case 2. $\mathtt{c} \triangleq \mathtt{for\ i}\ : \mathtt{P}_{\mathtt{n},\mathtt{q}}\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2$. By Figure 3.5:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \{\sigma\}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2(\mathtt{i} \gg \mathtt{1})\ \{\sigma'\}$$

Pick any $k \in \mathbb{N}$. For $j \in [1, k+1]$, define σ_j as follows:

$$\sigma_j^B = \sigma^B \wedge \sigma_j^n = k \bigwedge \forall i \in [1, j-1] \,\boldsymbol{.}\, \sigma_j^P(i) = \sigma'^P(1) \bigwedge$$

$$\forall i \in [j, k] \,\boldsymbol{.}\, \sigma_j^P(i) = \sigma^P(1)$$

$$\triangleright \text{ By Definition 7}$$

$$\Rightarrow \forall k \in \mathbb{N} \,\boldsymbol{.}\, \exists \sigma_1, \ldots, \sigma_{k+1} \,\boldsymbol{.}\, \sigma_1 = \sigma \upharpoonright k \bigwedge \sigma_{k+1} = \sigma' \upharpoonright k \bigwedge$$

$$\forall \lceil \mathtt{j} \rceil \in [1, k] \,\boldsymbol{.}\, \{\sigma_{\lceil \mathtt{j} \rceil}\}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2[\mathtt{i} \mapsto \mathtt{j}]\ \{\sigma_{\lceil \mathtt{j} \rceil + 1}\}$$

$$\triangleright \text{ Again by Figure 3.5}$$

$$\Rightarrow \forall k \in \mathbb{N} \,\boldsymbol{.}\, \{\sigma \upharpoonright k\}\ \mathtt{c}\ \{\sigma' \upharpoonright k\}$$

Case 3. $\mathsf{c} = \mathsf{c}_1;\mathsf{c}_2$. By Figure 3.5:

$$\{\sigma\}\ \mathsf{c}\ \{\sigma'\} \Rightarrow \exists\sigma''\,\boldsymbol{.}\,\{\sigma\}\ \mathsf{c}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \mathsf{c}_2\ \{\sigma'\} \Rightarrow$$

$$\triangleright \text{ By inductive hypothesis}$$

$$\exists\sigma''\,\boldsymbol{.}\,\forall k \in \mathbb{N}\,\boldsymbol{.}\,\{\sigma \upharpoonright k\}\ \mathsf{c}_1\ \{\sigma'' \upharpoonright k\} \wedge$$

$$\forall k \in \mathbb{N}\,\boldsymbol{.}\,\{\sigma'' \upharpoonright k\}\ \mathsf{c}_2\ \{\sigma' \upharpoonright k\} \Rightarrow$$

$$\triangleright \text{ Playing around with } \exists, \forall, \text{ and } \wedge$$

$$\forall k \in \mathbb{N}\,\boldsymbol{.}\,\exists\sigma''\,\boldsymbol{.}\,\{\sigma \upharpoonright k\}\ \mathsf{c}_1\ \{\sigma'' \upharpoonright k\} \wedge \{\sigma'' \upharpoonright k\}\ \mathsf{c}_2\ \{\sigma' \upharpoonright k\}$$

$$\triangleright \text{ again by Figure 3.5}$$

$$\Rightarrow \forall k \in \mathbb{N}\,\boldsymbol{.}\,\{\sigma \upharpoonright k\}\ \mathsf{c}\ \{\sigma' \upharpoonright k\}$$

This completes the proof. □

The next lemma states that if a store σ is transformed to σ' by executing a guarded command gc, then every generalization of σ is transformed to the corresponding generalization of σ' by executing gc.

Lemma 27 (Guarded Command Generalization)**.** *For any stores* σ, σ' *such that* $\sigma^n = \sigma'^n = 1$*, and any guarded command* $\mathsf{gc} \in \mathtt{GC}$*:*

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \forall k \in \mathbb{N}\,\boldsymbol{.}\,\{\sigma \upharpoonright k\}\ \mathsf{gc}\ \{\sigma' \upharpoonright k\}$$

Proof. We consider two cases.

Case 1. $\mathsf{gc} \triangleq \mathsf{e}\ ?\ \mathsf{c}_1 : \mathsf{c}_2$. By Figure 3.5:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma\rangle \to \textbf{true} \wedge \{\sigma\}\ \mathsf{c}\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By Lemma 28 and Lemma 36

$$\forall k \in \mathbb{N}\,.\,\langle \mathsf{e}, \sigma \upharpoonright k\rangle \to \textbf{true} \wedge \forall k \in \mathbb{N}\,.\,\{\sigma \upharpoonright k\}\ \mathsf{c}\ \{\sigma' \upharpoonright k\}$$

$\triangleright$ Since $\wedge$ distributes over $\forall$

$$\Rightarrow \forall k \in \mathbb{N}\,.\,\langle \mathsf{e}, \sigma \upharpoonright k\rangle \to \textbf{true} \wedge \{\sigma \upharpoonright k\}\ \mathsf{c}\ \{\sigma' \upharpoonright k\}$$

$\triangleright$ again by Figure 3.5

$$\Rightarrow \forall k \in \mathbb{N}\,.\,\{\sigma \upharpoonright k\}\ \mathsf{gc}\ \{\sigma' \upharpoonright k\}$$

Case 2. $\mathsf{gc} \triangleq \mathsf{gc}_1 \parallel \mathsf{gc}_2$. By Figure 3.5:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \{\sigma\}\ \mathsf{gc}_1\ \{\sigma'\} \vee \{\sigma\}\ \mathsf{gc}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\forall k \in \mathbb{N}\,.\,\{\sigma \upharpoonright k\}\ \mathsf{gc}_1\ \{\sigma' \upharpoonright k\} \bigvee$$

$$\forall k \in \mathbb{N}\,.\,\{\sigma \upharpoonright k\}\ \mathsf{gc}_2\ \{\sigma' \upharpoonright k\} \Rightarrow$$

$\triangleright$ Playing around with $\vee$ and $\forall$

$$\forall k \in \mathbb{N}\,.\,\{\sigma \upharpoonright k\}\ \mathsf{gc}_1\ \{\sigma' \upharpoonright k\} \vee \{\sigma \upharpoonright k\}\ \mathsf{gc}_2\ \{\sigma' \upharpoonright k\}$$

$\triangleright$ again by Figure 3.5

$$\Rightarrow \forall k \in \mathbb{N}\,.\,\{\sigma \upharpoonright k\}\ \mathsf{gc}\ \{\sigma' \upharpoonright k\}$$

This completes the proof. $\square$

A.1.5 Proofs of Lemmas Presented in Main Paper

In this section, we prove lemmas that were stated without proof in Section 3.3.5.

Proof of Lemma 4.

Proof. From Figure 3.6, we consider three sub-cases:

Case 1. $\varphi \in \mathsf{BP}$. Our result follows from Lemma 29.

Case 2. $\varphi = \forall \mathtt{i}\,\boldsymbol{.}\,\pi$ and $\pi \in \mathsf{PP}(\mathtt{i})$. Our result follows from Lemma 21.

Case 3. $\varphi = \pi \wedge \forall \mathtt{i}\,\boldsymbol{.}\,\pi'$ such that $\pi \in \mathsf{BP}$, and $\pi' \in \mathsf{PP}(\mathtt{i})$. In this case, by Definition 3:

$$\sigma \models \varphi \Leftrightarrow \sigma \models \pi \bigwedge \sigma \models \forall \mathtt{i}\,\boldsymbol{.}\,\pi'$$

$$\rhd \text{ by Lemma 29 and Lemma 21}$$

$$\Leftrightarrow \forall j \in [1, \sigma^n]\,\boldsymbol{.}\,\sigma \downharpoonright j \models \pi \bigwedge \forall j \in [1, \sigma^n]\,\boldsymbol{.}\,\sigma \downharpoonright j \models \forall \mathtt{i}\,\boldsymbol{.}\,\pi'$$

$$\rhd \text{ since } \forall \text{ distributes over } \wedge$$

$$\Leftrightarrow \forall j \in [1, \sigma^n]\,\boldsymbol{.}\,\sigma \downharpoonright j \models \pi \wedge \sigma \downharpoonright j \models \forall \mathtt{i}\,\boldsymbol{.}\,\pi'$$

$$\rhd \text{ again by Definition 3}$$

$$\Leftrightarrow \forall j \in [1, \sigma^n]\,\boldsymbol{.}\,\sigma \downharpoonright j \models \pi \wedge \forall \mathtt{i}\,\boldsymbol{.}\,\pi'$$

By alpha renaming of j to i, we get our desired result. $\square$

Proof of Lemma 5.

Proof. By considering the structure of φ. We consider three cases:

Case 1. $\varphi \in \mathsf{USF}$. By Lemma 4, we know that:

$$\sigma \models \varphi \Leftrightarrow \forall i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models \varphi \Rightarrow \exists i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models \varphi$$

Case 2. $\varphi \in \mathsf{ESF}$. By Lemma 23, we know that:

$$\sigma \models \varphi \Leftrightarrow \exists i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models \varphi \Rightarrow \exists i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models \varphi$$

Case 3. $\varphi \in \mathsf{USF} \wedge \mathsf{ESF}$. Without loss of generality, let $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where $\varphi_1 \in \mathsf{BP}$, $\varphi_2 \in \forall \mathtt{i} \textbf{.} \mathsf{PP}(\mathtt{i})$, and $\varphi_3 \in \exists \mathtt{i} \textbf{.} \mathsf{PP}(\mathtt{i})$. In this case, by Definition 3:

$$\sigma \models \varphi \Leftrightarrow \sigma \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \models \varphi_3 \Leftrightarrow$$

$\triangleright$ by Lemma 4 and Lemma 30

$$\forall i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models (\varphi_1 \wedge \varphi_2) \bigwedge \exists i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models \varphi_3 \Rightarrow$$

$\triangleright$ playing around with $\wedge$, $\forall$, and $\exists$

$$\exists i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \downharpoonleft i \models \varphi_3 \Rightarrow$$

$\triangleright$ again by Definition 3

$$\exists i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Rightarrow$$

$$\exists i \in [1, \sigma^n] \textbf{.} \sigma \downharpoonleft i \models \varphi$$

This completes the proof. □

Proof of Lemma 6.

Proof. The proof proceeds as follows. By Figure 3.5:

$$\{\sigma\}\ \mathtt{gc(k)}\ \{\sigma'\} \Rightarrow \sigma^n = \sigma'^n = \lceil \mathtt{k} \rceil \wedge \{\sigma\}\ \mathtt{gc}\ \{\sigma'\} \Rightarrow$$

$$\triangleright \text{ By Lemma 35}$$

$$\forall i \in [1, \sigma^n]\textbf{.}\ (\sigma \downharpoonleft i)^n = (\sigma' \downharpoonleft i)^n = 1 \wedge \{\sigma \downharpoonleft i\}\ \mathtt{gc}\ \{\sigma' \downharpoonleft i\}$$

$$\triangleright \text{ Again by Figure 3.5}$$

$$\Rightarrow \forall i \in [1, \sigma^n]\textbf{.}\ \{\sigma \downharpoonleft i\}\ \mathtt{gc(1)}\ \{\sigma' \downharpoonleft i\}$$

This completes the proof. □

Proof of Lemma 7.

Proof. By considering the structure of φ. We consider three cases:

Case 1. $\varphi \in \mathsf{USF}$. Follows directly from Lemma 4.

Case 2. $\varphi \in \mathsf{ESF}$. By Lemma 23, we know that:

$$\forall i \in [1, \sigma^n]\textbf{.}\ \sigma \downharpoonleft i \models \varphi \Rightarrow \exists i \in [1, \sigma^n]\textbf{.}\ \sigma \downharpoonleft i \models \varphi \Rightarrow \sigma \models \varphi$$

Case 3. $\varphi \in \mathsf{USF} \wedge \mathsf{ESF}$. Without loss of generality, let $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where $\varphi_1 \in \mathsf{BP}$, $\varphi_2 \in \forall \mathtt{i}\textbf{.}\mathsf{PP}(\mathtt{i})$, and $\varphi_3 \in \exists \mathtt{i}\textbf{.}\mathsf{PP}(\mathtt{i})$. In this case, by Definition 3:

$$\forall i \in [1, \sigma^n]\textbf{.}\ \sigma \downharpoonleft i \models \varphi \Leftrightarrow$$

$$\forall i \in [1, \sigma^n]\textbf{.}\ \sigma \downharpoonleft i \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Leftrightarrow$$

$$\forall i \in [1, \sigma^n]\textbf{.}\ \sigma \downharpoonleft i \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \downharpoonleft i \models \varphi_3 \Leftrightarrow$$

$$\triangleright \text{ playing around with } \wedge \text{ and } \forall$$

$$\forall i \in [1, \sigma^n] \centerdot \sigma \downharpoonleft i \models (\varphi_1 \wedge \varphi_2) \bigwedge \forall i \in [1, \sigma^n] \centerdot \sigma \downharpoonleft i \models \varphi_3 \Rightarrow$$

$$\triangleright \text{ weakening } \forall \text{ to } \exists$$

$$\forall i \in [1, \sigma^n] \centerdot \sigma \downharpoonleft i \models (\varphi_1 \wedge \varphi_2) \bigwedge \exists i \in [1, \sigma^n] \centerdot \sigma \downharpoonleft i \models \varphi_3 \Rightarrow$$

$$\triangleright \text{ by Lemma 4 and Lemma 30}$$

$$\sigma \models (\varphi_1 \wedge \varphi_2) \wedge \sigma \models \varphi_3 \Rightarrow \sigma \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Rightarrow \sigma \models \varphi$$

This completes the proof. □

Proof of Lemma 8.

Proof. The proof proceeds as follows. By Figure 3.5:

$$\{\sigma\}\ \mathtt{gc(1)}\ \{\sigma'\} \Rightarrow \sigma^n = \sigma'^n = 1 \wedge \{\sigma\}\ \mathtt{gc}\ \{\sigma'\} \Rightarrow$$

$$\triangleright \text{ By Lemma 38}$$

$$\forall \lceil \mathtt{k} \rceil \in \mathbb{N} \centerdot (\sigma \upharpoonleft \lceil \mathtt{k} \rceil)^n = (\sigma' \upharpoonleft \lceil \mathtt{k} \rceil)^n = \lceil \mathtt{k} \rceil \bigwedge$$

$$\{\sigma \upharpoonleft \lceil \mathtt{k} \rceil\}\ \mathtt{gc}\ \{\sigma' \upharpoonleft \lceil \mathtt{k} \rceil\}$$

$$\triangleright \text{ Again by Figure 3.5}$$

$$\Rightarrow \forall \lceil \mathtt{k} \rceil \in \mathbb{N} \centerdot \{\sigma \upharpoonleft \lceil \mathtt{k} \rceil\}\ \mathtt{gc(k)}\ \{\sigma' \upharpoonleft \lceil \mathtt{k} \rceil\}$$

This completes the proof. □

Appendix B

Proofs of Small Model Theorems for Hierarchical Data Structures

B.1 Proofs

This appendix contains proofs of our small model theorems and supporting lemmas. We begin with some introductory lemmas.

B.1.1 Introductory Lemmas

First we prove that projection does not affect the evaluation of expressions, as expressed by the following lemma.

Lemma 28. *Let* $\mathsf{e} \in \mathtt{E}$*,* $t \in \mathbb{B}$*,* $z \in [1,d]$*, and* σ *be any store. Then:*

$$\langle \mathsf{e}, \sigma \rangle \rightarrow t \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i} \rangle \rightarrow t$$

$$\Leftrightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i} \rangle \rightarrow t$$

Proof. Follows from the fact that e depends only on Boolean variables, and the following observation:

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma^B = (\sigma \downharpoonleft \mathsf{i})^B$$

□

B.1.2 Store Projection Lemmas

We now present a series of lemmas that explore the relationship between store projection and different types of formulas. These lemmas will be used later to prove our small model theorems. The first lemma states that a store σ satisfies a basic proposition π iff every projection of σ satisfies π.

Lemma 29. *Let $z \in [1,d]$, $\pi \in \mathsf{BP}$, and σ be any store. Then:*

$$\begin{aligned} \sigma \models \pi &\Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \pi \\ &\Leftrightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \pi \end{aligned}$$

Proof. Follows from the fact that π depends only on Boolean variables, and the following observation:

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma^B = (\sigma \downharpoonleft \mathsf{i})^B$$

□

The next lemma states that a store σ satisfies an existentially quantified formula π iff some projection of σ satisfies π.

Lemma 30. *Let $z \in [1,d]$, $\pi = \text{Æ}_1 \mathtt{i}_1 \dots \text{Æ}_z \mathtt{i}_z \centerdot \pi'$ and $\pi' \in \mathsf{PP}(\mathtt{i}_1, \dots, \mathtt{i}_z)$ and σ be any* *store. Then:*

$$\sigma \models \pi \Leftrightarrow \text{Æ}_1 i_1 \in [1, \sigma_1^n] \ldots \text{Æ}_z i_z \in [1, \sigma_z^n] \,\boldsymbol{.}\, \sigma \downharpoonleft (i_1, \ldots, i_z) \models \pi$$

Proof. By Definition 11, we know that:

$$\sigma \models \pi \Leftrightarrow \exists \text{Æ}_1 i_1 \in [1, \sigma_1^n] \ldots \text{Æ}_z i_z \in [1, \sigma_z^n] \boldsymbol{.}$$

$$\sigma \downharpoonleft (i_1, \ldots, i_z) \models \pi'[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_z \mapsto 1]$$

$$\triangleright \text{ let } \mathsf{i} = (i_1, \ldots, i_z)\text{; from Definition 10, since } ((\sigma \downharpoonleft \mathsf{i}) \downharpoonleft 1^z) = (\sigma \downharpoonleft \mathsf{i})$$

$$\Leftrightarrow \text{Æ}_1 i_1 \in [1, \sigma_1^n] \ldots \text{Æ}_z i_z \in [1, \sigma_z^n] \boldsymbol{.}$$

$$(\sigma \downharpoonleft \mathsf{i}) \downharpoonleft 1^z \models \pi'[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_z \mapsto 1]$$

$$\triangleright \text{ let } \mathsf{j} = (j_1, \ldots, j_z) \text{ be fresh variables; since } (\sigma \downharpoonleft i)_{1,z}^n = 1^z$$

$$\Leftrightarrow \text{Æ}_1 i_1 \in [1, \sigma_1^n] \ldots \text{Æ}_z i_z \in [1, \sigma_z^n] \boldsymbol{.}$$

$$\text{Æ}_1 j_1 \in [1, (\sigma \downharpoonleft \mathsf{i})_1^n] \ldots \text{Æ}_z j_z \in [1, (\sigma \downharpoonleft \mathsf{i})_z^n] \boldsymbol{.}$$

$$(\sigma \downharpoonleft \mathsf{i}) \downharpoonleft \mathsf{j} \models \pi'[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_z \mapsto 1]$$

$$\triangleright \text{ let } \mathtt{j}_1, \ldots, \mathtt{j}_z \text{ be fresh variables}$$

$$\Leftrightarrow \text{Æ}_1 i_1 \in [1, \sigma_1^n] \ldots \text{Æ}_z i_z \in [1, \sigma_z^n] \boldsymbol{.}$$

$$\text{Æ}_1 j_1 \in [1, (\sigma \downharpoonleft \mathsf{i})_1^n] \ldots \text{Æ}_z j_z \in [1, (\sigma \downharpoonleft \mathsf{i})_z^n] \boldsymbol{.}$$

$$(\sigma \downharpoonleft \mathsf{i}) \downharpoonleft \mathsf{j} \models \pi'[\mathtt{i}_1 \mapsto \mathtt{j}_1] \ldots [\mathtt{i}_z \mapsto \mathtt{j}_z][\mathtt{j}_1 \mapsto 1] \ldots [\mathtt{j}_z \mapsto 1]$$

$$\triangleright \text{ by Definition 11}$$

$$\Leftrightarrow Æ_1 i_1 \in [1,\sigma_1^n] \ldots Æ_z i_z \in [1,\sigma_z^n] \centerdot$$

$$\sigma \downharpoonleft \mathsf{i} \models Æ_1 \mathtt{j}_1 \ldots Æ_z \mathtt{j}_z \centerdot (\pi'[\mathtt{i}_1 \mapsto \mathtt{j}_1] \ldots [\mathtt{i}_z \mapsto \mathtt{j}_z])$$

By alpha renaming, we know that $Æ_1 \mathtt{j}_1 \ldots Æ_z \mathtt{j}_z \centerdot (\pi'[\mathtt{i}_1 \mapsto \mathtt{j}_1] \ldots [\mathtt{i}_z \mapsto \mathtt{j}_z])$ is equivalent to π. This gives us our desired result. □

The next lemma states that a store σ satisfies an existential state formula φ iff certain projection of σ satisfy φ.

Lemma 31. *Let* $\varphi \in \mathsf{ESF}$ *and* σ *be any store. Then:*

$$\sigma \models \varphi \Rightarrow \exists \mathsf{i} \in \otimes(\sigma_{1,d}^n) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi$$

Proof. From Figure 4.6, we consider three sub-cases:

Case 1. $\varphi \in \mathsf{BP}$. Our result follows from Lemma 29.

Case 2. $\varphi = Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi$ and $\pi \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. Our result follows from Lemma 30.

Case 3. $\varphi = \pi \wedge Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$ such that $\pi \in \mathsf{BP}$, and $\pi' \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. By Definition 11:

$$\sigma \models \varphi \Rightarrow \sigma \models \pi \bigwedge \sigma \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ by Lemma 29 and Lemma 30}$$

$$\Rightarrow \forall \mathsf{j} \in \otimes(\sigma_{1,z}^n) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \bigwedge$$

$$\exists \mathsf{j} \in \otimes(\sigma_{1,z}^n) \centerdot \sigma \downharpoonleft \mathsf{j} \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ playing around with } \wedge, \forall, \text{ and } \exists$$

$$\Rightarrow \exists \mathsf{j} \in \otimes(\sigma_{1,z}^n) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \wedge$$

$$\sigma \downharpoonleft \mathsf{j} \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ again by Definition 11}$$

$$\Rightarrow \exists \mathsf{j} \in \otimes(\sigma^{n}_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \wedge Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

By alpha renaming j to i, we get our desired result. □

Lemma 32. *Let* $\varphi \in \mathsf{ESF}$ *and* σ *be any store. Then:*

$$\forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi \Rightarrow \sigma \models \varphi$$

Proof. From Figure 4.6, we consider three sub-cases:

Case 1. $\varphi \in \mathsf{BP}$. Our result follows from Lemma 29.

Case 2. $\varphi = Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi$ and $\pi \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. Our result follows from Lemma 30.

Case 3. $\varphi = \pi \wedge Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$ such that $\pi \in \mathsf{BP}$, and $\pi' \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$.

$$\forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \pi \wedge Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ by Definition 11}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \pi \wedge$$

$$\sigma \downharpoonleft \mathsf{i} \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ playing around with } \wedge \text{ and } \forall$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \pi \bigwedge$$

$$\forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ by Lemma 29 and Lemma 30}$$

$$\Rightarrow \sigma \models \pi \bigwedge \sigma \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$\triangleright$ again by Definition 11

$$\Rightarrow \sigma \models \varphi$$

This completes our proof. $\square$

B.1.3 Store Projection and Command Lemmas

The following lemma states that if a store σ is transformed to σ' by executing a command c_1, then every projection of σ is transformed to the corresponding projection of σ' by executing c_1.

Lemma 33 (Command Projection)**.** *For $z \in [1,d]$, any stores σ, σ' such that $\sigma^n_{1,z-1} = \sigma'^n_{1,z-1} = 1^{z-1}$, and any command $\mathsf{c} \in \mathtt{C}_z[\mathtt{i}_1 \mapsto 1]\ldots[\mathtt{i}_{z-1} \mapsto 1]$:*

$$\{\sigma\}\ \mathsf{c}\ \{\sigma'\} \Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{c}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Proof. By induction on the structure of c. We consider four subcases.

Case 0. $\mathsf{c} \triangleq \mathtt{skip}$. Trivially since by Figure 4.5, $\sigma = \sigma'$.

Case 1. $\mathsf{c} \triangleq \mathsf{b} := \mathsf{e}$. Suppose $\langle \mathsf{e}, \sigma \rangle \to t$. By Figure 4.5:

$$\{\sigma\}\ \mathsf{c}\ \{\sigma'\} \Rightarrow \sigma' = \sigma[\sigma^B \mapsto \sigma^B[\mathsf{b} \mapsto t]] \Rightarrow$$

$\triangleright$ by Definition 10 and Lemma 28, $\forall \mathsf{i} \in \otimes(\sigma^n_{1,d})$

$$\sigma' \downharpoonleft \mathsf{i} = (\sigma \downharpoonleft \mathsf{i})[(\sigma \downharpoonleft \mathsf{i})^B \mapsto (\sigma \downharpoonleft \mathsf{i})^B[\mathsf{b} \mapsto t]] \wedge \langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i} \rangle \to t$$

$\triangleright$ again by Figure 4.5

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{c}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 2. $\mathtt{c} \triangleq \mathtt{for}\ \mathtt{i}_z\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2$.

Let $\sigma^n_z = N$. By Figure 4.5:

$$\forall y \in [1,N] \centerdot \{\sigma \downharpoonleft (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft (1^{z-1}, y)\}$$

$\triangleright$ By Lemma 34

$$\forall y \in [1,N] \centerdot \forall \mathsf{i} \in \otimes(1^z, \sigma^n_{z+1,d}) \centerdot$$

$$\{\sigma \downharpoonleft (1^{z-1}, y) \downharpoonleft \mathsf{i}\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft (1^{z-1}, y) \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Combining the two $\forall$ quantifiers

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot$$

$$\{\sigma \downharpoonleft \mathsf{i}\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Expanding out

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \forall y \in [1,1] \centerdot$$

$$\{\sigma \downharpoonleft \mathsf{i} \downharpoonleft (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft \mathsf{i} \downharpoonleft (1^{z-1}, y)\}$$

$\triangleright$ Again by Figure 4.5

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathtt{c}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 3. $\mathtt{c} \triangleq \mathtt{c}_1; \mathtt{c}_2$. By Figure 4.5:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \exists \sigma'' \centerdot \{\sigma\}\ \mathtt{c}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \mathtt{c}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\exists \sigma'' \centerdot \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{c}_1\ \{\sigma'' \downharpoonleft \mathsf{i}\} \wedge$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma'' \downharpoonleft \mathsf{i}\}\ \mathsf{c}_2\ \{\sigma' \downharpoonleft \mathsf{i}\} \Rightarrow$$

$$\rhd \text{ Swapping } \forall \text{ and } \exists$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \exists \sigma'' \centerdot$$

$$\{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{c}_1\ \{\sigma'' \downharpoonleft \mathsf{i}\} \wedge \{\sigma'' \downharpoonleft \mathsf{i}\}\ \mathsf{c}_2\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$$\rhd \text{ Again by Figure 4.5}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{c}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

This completes the proof. □

Lemma 34 (Parameterized Command Projection)**.** *For $z \in [1,d]$, any stores σ, σ' such that $\sigma^n_{1,z} = \sigma'^n_{1,z} = 1^z$, and any command $\widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}' \in (\widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z : \widehat{\mathtt{C}}_z)[\mathtt{i}_1 \mapsto 1] \dots [\mathtt{i}_z \mapsto 1]$:*

$$\{\sigma\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma'\} \Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Proof. First consider the case $\langle \widehat{\mathsf{e}}, \sigma \rangle \to \mathbf{true}$. Note that $\widehat{\mathsf{e}}$ does not contain any index variables and only refers to the parametric array at depth z. Since $\sigma^n_{1,z} = \sigma'^n_{1,z} = 1^z$, we claim that:

$$\langle \widehat{\mathsf{e}}, \sigma \rangle \to \mathbf{true} \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \langle \widehat{\mathsf{e}}, \sigma \downharpoonleft \mathsf{i} \rangle \to \mathbf{true}$$

$$\Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \langle \widehat{\mathsf{e}}, \sigma' \downharpoonleft \mathsf{i} \rangle \to \mathbf{true} \Leftrightarrow \langle \widehat{\mathsf{e}}, \sigma' \rangle \to \mathbf{true}$$

Now by Figure 4.5, $\{\sigma\}\ \widehat{\mathsf{c}}\ \{\sigma'\}$. We proceed by induction on the structure of $\widehat{\mathsf{c}}$. We consider three subcases.

Case 1. $\widehat{\mathsf{c}} \triangleq \mathtt{P}[1] \dots \mathtt{P}[1].\mathtt{F}[\mathtt{r}] := \widehat{\mathsf{e}}'$. Suppose $\langle \widehat{\mathsf{e}}', \sigma \rangle \to t$. Since $\widehat{\mathsf{e}}'$ also refers to the parametric array at depth z and $\sigma^n_{1,z} = \sigma'^n_{1,z} = 1^z$. By Figure 4.5:

$$\sigma' = \sigma[\sigma^P \mapsto \sigma^P[\sigma^P_z \mapsto [\sigma^P_z[(1^z, \lceil \mathtt{r} \rceil) \mapsto t]]]]$$

$$\rhd \text{ by Definition 10, } \forall \mathsf{i} \in \otimes(\sigma^n_{1,d})$$

$$\sigma' \downharpoonleft \mathsf{i} = \sigma \downharpoonleft \mathsf{i}[\sigma \downharpoonleft \mathsf{i}^P \mapsto \sigma \downharpoonleft \mathsf{i}^P[\sigma \downharpoonleft \mathsf{i}^P_z \mapsto [\sigma \downharpoonleft \mathsf{i}^P_z[(1^z, \lceil \mathtt{r} \rceil) \mapsto t]]]] \wedge$$

$$\langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i} \rangle \to t$$

$$\rhd \text{ again by Figure 4.5}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 2. $\widehat{\mathsf{c}} \triangleq \widehat{\mathsf{c}}_1; \widehat{\mathsf{c}}_2$. By Figure 4.5:

$$\{\sigma\}\ \widehat{\mathsf{c}}\ \{\sigma'\} \Rightarrow \exists \sigma'' \centerdot \{\sigma\}\ \widehat{\mathsf{c}}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \widehat{\mathsf{c}}_2\ \{\sigma'\} \Rightarrow$$

$$\rhd \text{ By inductive hypothesis}$$

$$\exists \sigma'' \centerdot \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}_1\ \{\sigma'' \downharpoonleft \mathsf{i}\} \wedge$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma'' \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}_2\ \{\sigma' \downharpoonleft \mathsf{i}\} \Rightarrow$$

$$\rhd \text{ Swapping } \forall \text{ and } \exists$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \exists \sigma'' \centerdot$$

$$\{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}_1\ \{\sigma'' \downharpoonleft \mathsf{i}\} \wedge \{\sigma'' \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}_2\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$$\rhd \text{ Again by Figure 4.5}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 3. $\widehat{\mathsf{c}} \triangleq \mathsf{c}$. Follows directly from Lemma 33.

Since $\forall \mathsf{i} \in \otimes(\sigma^n_{1,d})\langle \widehat{\mathsf{e}}, \sigma \downharpoonright \mathsf{i}\rangle \rightarrow \textbf{true}$ and $\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonright \mathsf{i}\}\ \widehat{\mathsf{c}}\ \{\sigma' \downharpoonright \mathsf{i}\}$, from Figure 4.5, we have our desired result $\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonright \mathsf{i}\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma' \downharpoonright \mathsf{i}\}$.

The proof for the case when $\langle \widehat{\mathsf{e}}, \sigma\rangle \rightarrow \textbf{false}$ is analogous. This completes the proof. $\square$

Note that the dependence between Lemma 33 and Lemma 34 is not circular since for Lemma 34 to be valid for z, Lemma 33 must be valid for $z+1$, and for $z = d$, Lemma 34 does not require Lemma 33 to be valid. Hence we can argue using mutual induction, starting from $z = d$, and prove that both lemmas are valid for all $z \in [1,d]$.

The next lemma states that if a store σ is transformed to σ' by executing a guarded command gc, then every projection of σ is transformed to the corresponding projection of σ' by executing gc.

Lemma 35 (Guarded Command Projection)**.** *For any stores* σ, σ'*, and any guarded command* $\mathsf{gc} \in \mathtt{GC}$*:*

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonright \mathsf{i}\}\ \mathsf{gc}\ \{\sigma' \downharpoonright \mathsf{i}\}$$

Proof. We consider two cases.

Case 1. $\mathsf{gc} \triangleq \mathsf{e}\ ?\ \mathsf{c}_1 : \mathsf{c}_2$. By Figure 4.5, we have two sub-cases:

Case 1.1. In this case:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma\rangle \rightarrow \textbf{true} \wedge \{\sigma\}\ \mathsf{c}_1\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By Lemma 28 and Lemma 33

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \langle \mathsf{e}, \sigma \downharpoonright \mathsf{i}\rangle \rightarrow \textbf{true} \wedge$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonright \mathsf{i}\}\ \mathsf{c}_1\ \{\sigma' \downharpoonright \mathsf{i}\}$$

$\triangleright$ Since $\wedge$ distributes over $\forall$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i} \rangle \rightarrow \textbf{true} \wedge \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{c}_1\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Again by Figure 4.5

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 1.2. In this case:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma \rangle \rightarrow \textbf{false} \wedge \{\sigma\}\ \mathsf{c}_2\ \{\sigma'\}$$

The proof is analogous to the previous sub-case.

Case 2. $\mathsf{gc} \triangleq \mathsf{gc}_1 \parallel \mathsf{gc}_2$. By Figure 4.5:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \{\sigma\}\ \mathsf{gc}_1\ \{\sigma'\} \vee \{\sigma\}\ \mathsf{gc}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}_1\ \{\sigma' \downharpoonleft \mathsf{i}\} \bigvee$$

$$\forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}_2\ \{\sigma' \downharpoonleft \mathsf{i}\} \Rightarrow$$

$\triangleright$ Playing around with $\vee$ and $\forall$

$$\forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot$$

$$\{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}_1\ \{\sigma' \downharpoonleft \mathsf{i}\} \vee \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}_2\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Again by Figure 4.5

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

This completes the proof. □

B.1.4 Store Generalization Lemmas

We now present a series of lemmas that relate the execution semantics of PGCL to store generalization. The first lemma states that if a store σ is transformed to σ' by executing a command $\mathtt{c}$, then every generalization of σ is transformed to the corresponding generalization of σ' by executing $\mathtt{c}$.

Lemma 36 (Command Generalization)**.** *For $z \in [1,d]$, any stores σ, σ' such that $\sigma^n = \sigma'^n = 1^d$, any $\mathsf{k} \in \mathbb{N}^d$ such that $\mathsf{k}_{1,z-1} = 1^{z-1}$, and any command $\mathtt{c} \in \mathtt{C}_z[\mathtt{i}_1 \mapsto 1]\ldots[\mathtt{i}_{z-1} \mapsto 1]$:*

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \mathtt{c}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Proof. By induction on the structure of $\mathtt{c}$. We consider four subcases.

Case 0. $\mathtt{c} \triangleq \mathtt{skip}$. Trivially since by Figure 4.5, $\sigma = \sigma'$.

Case 1. $\mathtt{c} \triangleq \mathtt{b} := \mathtt{e}$. Suppose $\langle \mathtt{e}, \sigma \rangle \to t$. By Figure 4.5:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \sigma' = \sigma[\sigma^B \mapsto \sigma^B[\mathtt{b} \mapsto t]] \Rightarrow$$

$$\triangleright \text{ by Definition 19 and Lemma 28}$$

$$\sigma' \upharpoonright \mathsf{k} = (\sigma \upharpoonright \mathsf{k})[(\sigma \upharpoonright \mathsf{k})^B \mapsto (\sigma \upharpoonright \mathsf{k})^B[\mathtt{b} \mapsto t]] \wedge \langle \mathtt{e}, \sigma \upharpoonright \mathsf{k} \rangle \to t$$

$$\triangleright \text{ again by Figure 4.5}$$

$$\Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \mathtt{c}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 2. $\mathtt{c} \triangleq \mathtt{for}\ \mathtt{i}_z\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2$. By Figure 4.5:

$$\forall y \in [1,1] \centerdot \{\sigma \downharpoonright (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonright (1^{z-1}, y)\}$$

$\triangleright$ Simplifying

$$\{\sigma\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma'\}$$

$\triangleright$ By Lemma 37

$$\{\sigma \upharpoonright \mathsf{k}[z \mapsto 1]\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \upharpoonright \mathsf{k}[z \mapsto 1]\}$$

$\triangleright$ Let $\mathsf{k}_z = N$. Expanding out

$$\forall y \in [1,N]\,\textbf{.}\,\{\sigma \upharpoonright \mathsf{k} \downharpoonright (1^{z-1},y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \upharpoonright \mathsf{k} \downharpoonright (1^{z-1},y)\}$$

$\triangleright$ Again by Figure 4.5

$$\Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 3. $\mathsf{c} = \mathsf{c}_1; \mathsf{c}_2$. By Figure 4.5:

$$\{\sigma\}\ \mathsf{c}\ \{\sigma'\} \Rightarrow \exists \sigma''\,\textbf{.}\,\{\sigma\}\ \mathsf{c}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \mathsf{c}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\exists \sigma''\,\textbf{.}\,\forall \mathsf{k} \in \mathbb{N}^d\,\textbf{.}\,\{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}_1\ \{\sigma'' \upharpoonright \mathsf{k}\} \wedge$$

$$\forall \mathsf{k} \in \mathbb{N}^d\,\textbf{.}\,\{\sigma'' \upharpoonright \mathsf{k}\}\ \mathsf{c}_2\ \{\sigma' \upharpoonright \mathsf{k}\} \Rightarrow$$

$\triangleright$ Playing around with $\exists$, $\forall$, and $\wedge$

$$\forall \mathsf{k} \in \mathbb{N}^d\,\textbf{.}\,\exists \sigma''\,\textbf{.}\,\{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}_1\ \{\sigma'' \upharpoonright \mathsf{k}\} \wedge \{\sigma'' \upharpoonright \mathsf{k}\}\ \mathsf{c}_2\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ again by Figure 4.5

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d\,\textbf{.}\,\{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

This completes the proof. $\square$

Lemma 37 (Parameterized Command Generalization)**.** *For $z \in [1,d]$, any stores σ, σ' such that $\sigma^n = \sigma'^n = 1^d$, any $\mathsf{k} \in \mathbb{N}^d$ such that $\mathsf{k}_{1,z} = 1^z$, and any command $\widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}' \in (\widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z : \widehat{\mathtt{C}}_z)[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_z \mapsto 1]$:*

$$\{\sigma\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma'\} \Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Proof. First consider the case $\langle \widehat{\mathsf{e}}, \sigma \rangle \to \mathbf{true}$. Note that $\widehat{\mathsf{e}}$ does not contain any index variables and only refers to the parametric array at depth z. Since $\sigma^n_{1,z} = \sigma'^n_{1,z} = 1^z$, we claim that:

$$\langle \widehat{\mathsf{e}}, \sigma \rangle \to \mathbf{true} \Leftrightarrow \langle \widehat{\mathsf{e}}, \sigma \upharpoonright \mathsf{k} \rangle \to \mathbf{true}$$

$$\Leftrightarrow \langle \widehat{\mathsf{e}}, \sigma' \upharpoonright \mathsf{k} \rangle \to \mathbf{true} \Leftrightarrow \langle \widehat{\mathsf{e}}, \sigma' \rangle \to \mathbf{true}$$

Now by Figure 4.5, $\{\sigma\}\ \widehat{\mathsf{c}}\ \{\sigma'\}$. We proceed by induction on the structure of $\widehat{\mathsf{c}}$. We consider three subcases.

Case 1. $\widehat{\mathsf{c}} \triangleq \mathtt{P[1]} \ldots \mathtt{P[1].F[r]} := \widehat{\mathsf{e}}'$. Suppose $\langle \widehat{\mathsf{e}}', \sigma \rangle \to t$. Since $\widehat{\mathsf{e}}'$ also refers to the parametric array at depth z and $\sigma^n_{1,z} = \sigma'^n_{1,z} = 1^z$. By Figure 4.5:

$$\sigma' = \sigma[\sigma^P \mapsto \sigma^P[\sigma^P_z \mapsto [\sigma^P_z[(1^z, \lceil \mathtt{r} \rceil) \mapsto t]]]]$$

$$\triangleright \text{ by Definition 19}$$

$$\sigma' \upharpoonright \mathsf{k} = \sigma \upharpoonright \mathsf{k}[\sigma \upharpoonright \mathsf{k}^P \mapsto \sigma \upharpoonright \mathsf{k}^P[\sigma \upharpoonright \mathsf{k}^P_z \mapsto [\sigma \upharpoonright \mathsf{k}^P_z[(1^z, \lceil \mathtt{r} \rceil) \mapsto t]]]] \wedge$$

$$\langle \mathsf{e}, \sigma \upharpoonright \mathsf{k} \rangle \to t$$

$$\triangleright \text{ again by Figure 4.5}$$

$$\Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 2. $\widehat{\mathsf{c}} \triangleq \widehat{\mathsf{c}}_1;\widehat{\mathsf{c}}_2$. By Figure 4.5:

$$\{\sigma\}\ \widehat{\mathsf{c}}\ \{\sigma'\} \Rightarrow \exists \sigma'' \centerdot \{\sigma\}\ \widehat{\mathsf{c}}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \widehat{\mathsf{c}}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\exists \sigma'' \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}_1\ \{\sigma'' \upharpoonright \mathsf{k}\} \wedge \{\sigma'' \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}_2\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ Again by Figure 4.5

$$\Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 3. $\widehat{\mathsf{c}} \triangleq \mathsf{c}$. Follows directly from Lemma 36.

Since $\langle \widehat{\mathsf{e}}, \sigma \upharpoonright \mathsf{k}\rangle \to$ **true** and $\{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}\ \{\sigma' \upharpoonright \mathsf{k}\}$, from Figure 4.5, we have our desired result $\{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma' \upharpoonright \mathsf{k}\}$.

The proof for the case when $\langle \widehat{\mathsf{e}}, \sigma\rangle \to$ **false** is analogous. This completes the proof. $\square$

Note that the dependence between Lemma 36 and Lemma 37 is not circular since for Lemma 37 to be valid for z, Lemma 36 must be valid for $z+1$, and for $z = d$, Lemma 37 does not require Lemma 36 to be valid. Hence we can argue using mutual induction, starting from $z = d$, and prove that both lemmas are valid for all $z \in [1, d]$.

The next lemma states that if a store σ is transformed to σ' by executing a guarded command gc, then every generalization of σ is transformed to the corresponding generalization of σ' by executing gc.

Lemma 38 (Guarded Command Generalization). *For any stores* σ, σ' *such that* $\sigma^n = \sigma'^n =$

1^d*, and any guarded command* $\mathsf{gc} \in \mathtt{GC}$*:*

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \,.\, \{\sigma \restriction \mathsf{k}\}\ \mathsf{gc}\ \{\sigma' \restriction \mathsf{k}\}$$

Proof. We consider two cases.

Case 1. $\mathsf{gc} \triangleq \mathsf{e}\ ?\ \mathsf{c}_1 : \mathsf{c}_2$. By Figure 4.5, we have two sub-cases:

Case 1.1. In this case:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma\rangle \to \textbf{true} \wedge \{\sigma\}\ \mathsf{c}_1\ \{\sigma'\} \Rightarrow$$

$$\rhd \text{ By Lemma 28 and Lemma 36}$$

$$\forall \mathsf{k} \in \mathbb{N}^d \,.\, \langle \mathsf{e}, \sigma \restriction \mathsf{k}\rangle \to \textbf{true} \wedge \forall \mathsf{k} \in \mathbb{N}^d \,.\, \{\sigma \restriction \mathsf{k}\}\ \mathsf{c}_1\ \{\sigma' \restriction \mathsf{k}\}$$

$$\rhd \text{ Since } \wedge \text{ distributes over } \forall$$

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \,.\, \langle \mathsf{e}, \sigma \restriction \mathsf{k}\rangle \to \textbf{true} \wedge \{\sigma \restriction \mathsf{k}\}\ \mathsf{c}_1\ \{\sigma' \restriction \mathsf{k}\}$$

$$\rhd \text{ again by Figure 4.5}$$

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \,.\, \{\sigma \restriction \mathsf{k}\}\ \mathsf{gc}\ \{\sigma' \restriction \mathsf{k}\}$$

Case 1.2. In this case:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma\rangle \to \textbf{false} \wedge \{\sigma\}\ \mathsf{c}_2\ \{\sigma'\}$$

The proof is analogous to the previous sub-case.

Case 2. $\mathsf{gc} \triangleq \mathsf{gc}_1 \parallel \mathsf{gc}_2$. By Figure 4.5:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \{\sigma\}\ \mathsf{gc}_1\ \{\sigma'\} \vee \{\sigma\}\ \mathsf{gc}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\forall \mathsf{k} \in \mathbb{N}^d \,.\, \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}_1\ \{\sigma' \upharpoonright \mathsf{k}\} \bigvee$$

$$\forall \mathsf{k} \in \mathbb{N}^d \,.\, \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}_2\ \{\sigma' \upharpoonright \mathsf{k}\} \Rightarrow$$

$\triangleright$ Playing around with $\vee$ and $\forall$

$$\forall \mathsf{k} \in \mathbb{N}^d \,.\, \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}_1\ \{\sigma' \upharpoonright \mathsf{k}\} \vee \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}_2\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ again by Figure 4.5

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \,.\, \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

This completes the proof. $\square$

B.1.5 Proofs of Small Model Theorems

In this section, we prove our small model theorems. We first present a set of supporting lemmas for the proof of Theorem 9. In some cases, the proof of the lemma is in the appendix. In addition, the proofs of these lemmas rely on other lemmas, which are in the appendix.

In the following proofs, for $z \in [1,d]$, and any tuple $\mathsf{i}^z = (i_1, \ldots, i_z)$, we write $Æ\mathsf{i}^z$ to mean $Æ_1 \imath_1, \ldots, Æ_z \imath_z$ where the tuple of quantifiers $(Æ_1, \ldots, Æ_z)$ (where for each z, Æ is either $\forall$ or $\exists$) is fixed and remains fixed across all instances of Æ in the same line of reasoning. This ensures consistency between all $Æ\mathsf{i}^z$ and $Æj^z$. Note that our results hold for any combination of quantifiers as long as they are consistent.

The first lemma states that if a store σ satisfies a generic state formula φ, then some projection of σ satisfies φ.

Lemma 39. *Let* $\varphi \in \mathsf{GSF}$*, and* σ *be any store. Then:*

$$\sigma \models \varphi \Rightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi$$

Proof. By considering the structure of φ. We consider three cases:

Case 1. $\varphi \in \mathsf{USF}$. By Lemma 40, we know that:

$$\sigma \models \varphi \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi \Rightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi$$

Case 2. $\varphi \in \mathsf{ESF}$. Follows directly from Lemma 31.

Case 3. $\varphi \in \mathsf{USF} \wedge \mathsf{ESF}$. Without loss of generality, let $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where $\varphi_1 \in \mathsf{BP}$, $\varphi_2 \in \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$, and $\varphi_3 \in Æ_1 \mathtt{i}_1, \ldots, Æ_z \mathtt{i}_z \centerdot \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. By Definition 11:

$$\sigma \models \varphi \Leftrightarrow \sigma \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \models \varphi_3 \Leftrightarrow$$

$\triangleright$ by Lemma 40 and Lemma 30

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi_3 \Rightarrow$$

$\triangleright$ playing around with $\wedge$, $\forall$, and $\exists$

$$\exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \downharpoonleft \mathsf{i} \models \varphi_3 \Rightarrow$$

$\triangleright$ again by Definition 11

$$\exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Rightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi$$

This completes the proof. □

The next lemma states that a store σ satisfies an universal state formula φ iff every projection of σ satisfies φ.

Lemma 40. *Let* $\varphi \in \mathsf{USF}$ *and* σ *be any store. Then:*

$$\sigma \models \varphi \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi$$

Proof. From Figure 4.6, we consider three sub-cases:

Case 1. $\varphi \in \mathsf{BP}$. Our result follows from Lemma 29.

Case 2. $\varphi = \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi$ and $\pi \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. Our result follows from Lemma 30.

Case 3. $\varphi = \pi \wedge \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$ such that $\pi \in \mathsf{BP}$, and $\pi' \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. In this case, by Definition 11:

$$\sigma \models \varphi \Leftrightarrow \sigma \models \pi \bigwedge \sigma \models \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$$

$\triangleright$ by Lemma 29 and Lemma 30

$$\Leftrightarrow \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \bigwedge \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$$

$\triangleright$ since $\forall$ distributes over $\wedge$

$$\Leftrightarrow \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \wedge \sigma \downharpoonleft \mathsf{j} \models \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$$

$\triangleright$ again by Definition 11

$$\Leftrightarrow \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \wedge \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$$

By alpha renaming of j to i, we get our desired result. $\square$

The next lemma states that if a store σ is transformed to σ' by executing an instantiated guarded command $\mathtt{gc}(\mathsf{k})$, then every projection of σ is transformed to the corresponding projection of σ' by executing $\mathtt{gc}(\mathsf{k})$.

Lemma 41 (Instantiated Command Projection)**.** *For any stores* σ, σ' *and instantiated* *guarded command* $\mathsf{gc(k)}$*:*

$$\{\sigma\}\ \mathsf{gc(k)}\ \{\sigma'\} \Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}(1^d)\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Proof. The proof proceeds as follows. By Figure 4.5:

$$\{\sigma\}\ \mathsf{gc(k)}\ \{\sigma'\} \Rightarrow \sigma^n = \sigma'^n = \mathsf{k} \wedge \{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By Lemma 35

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot (\sigma \downharpoonleft \mathsf{i})^n = (\sigma' \downharpoonleft \mathsf{i})^n = 1^d \wedge \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Again by Figure 4.5

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}(1^d)\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

This completes the proof. □

The last lemma relating store projection and formulas states that if every projection of a store σ satisfies a generic state formula φ, then σ satisfies φ.

Lemma 42. *Let* $\varphi \in \mathsf{GSF}$ *and* σ *be any store. Then:*

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi \Rightarrow \sigma \models \varphi$$

Proof. By considering the structure of φ. We consider three cases:

Case 1. $\varphi \in \mathsf{USF}$. Follows directly from Lemma 40.

Case 2. $\varphi \in \mathsf{ESF}$. Follows directly from Lemma 32.

Case 3. $\varphi \in \mathsf{USF} \wedge \mathsf{ESF}$. Without loss of generality, let $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where $\varphi_1 \in \mathsf{BP}$,

$\varphi_2 \in \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \mathsf{PP}(\mathtt{i}_1 \ldots \mathtt{i}_z)$, and $\varphi_3 \in Æ\mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \mathsf{PP}(\mathtt{i}_1 \ldots \mathtt{i}_z)$. by Definition 11:

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi \Leftrightarrow$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Leftrightarrow$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \downharpoonleft \mathsf{i} \models \varphi_3 \Leftrightarrow$$

$\triangleright$ playing around with $\wedge$ and $\forall$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi_3 \Rightarrow$$

$\triangleright$ weakening $\forall$ to $\exists$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi_3 \Rightarrow$$

$\triangleright$ by Lemma 40 and Lemma 30

$$\sigma \models (\varphi_1 \wedge \varphi_2) \wedge \sigma \models \varphi_3 \Rightarrow \sigma \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Rightarrow \sigma \models \varphi$$

This completes the proof. □

So far, we used the concept of store projection to show that the effect of executing a PGCL program carries over from larger stores to unit stores (i.e., stores obtained via projection). To prove our small model theorems, we also need to show that the effect of executing a PGCL program propagate in the opposite direction, i.e., from unit stores to larger stores. To this end, we first present a notion, called store generalization, that relates unit stores to those of arbitrarily large size.

Definition 19 (Store Generalization). *Let* $\sigma = (\sigma^B, 1^d, \sigma^P)$ *be any store. For any* $\mathsf{k} \in \mathbb{N}^d$

we write $\sigma \upharpoonright \mathsf{k}$ *to mean the store satisfying the following condition:*

$$(\sigma \upharpoonright \mathsf{k})^n = \mathsf{k} \wedge \forall \mathsf{i} \in \otimes(\mathsf{k}) \centerdot (\sigma \upharpoonright \mathsf{k}) \downharpoonright \mathsf{i} = \sigma$$

Intuitively, $\sigma \upharpoonright \mathsf{k}$ is constructed by duplicating the only rows of σ^P at each depth, and leaving the other components of σ unchanged. We now present a lemma related to store generalization, which is needed for the proof of Theorem 9. The lemma states that if a store σ is transformed to σ' by executing an instantiated guarded command $\mathsf{gc}(\mathsf{k})$, then every generalization of σ is transformed to the corresponding generalization of σ' by executing $\mathsf{gc}(\mathsf{k})$.

Lemma 43 (Instantiated Command Generalization)**.** *For any stores* σ, σ' *and instantiated guarded command* $\mathsf{gc}(\mathtt{1}^d)$*:*

$$\{\sigma\}\ \mathsf{gc}(\mathtt{1}^d)\ \{\sigma'\} \Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}(\mathsf{k})\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Proof. The proof proceeds as follows. By Figure 4.5:

$$\{\sigma\}\ \mathsf{gc}(\mathtt{1}^d)\ \{\sigma'\} \Rightarrow \sigma^n = \sigma'^n = 1^d \wedge \{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By Definition 19 and Lemma 38

$$\forall \mathsf{k} \in \mathbb{N}^d \centerdot (\sigma \upharpoonright \mathsf{k})^n = (\sigma' \upharpoonright \mathsf{k})^n = \mathsf{k} \bigwedge \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ Again by Figure 4.5

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}(\mathsf{k})\ \{\sigma' \upharpoonright \mathsf{k}\}$$

This completes the proof. □

B.1.6 Proof of Theorem 9

Theorem 1 (Small Model Safety 1)**.** *A Kripke structure $M(\mathsf{gc}(\mathsf{k}), Init)$ exhibits a formula φ iff there is a reachable state σ of $M(\mathsf{gc}(\mathsf{k}), Init)$ such that $\sigma \models \varphi$. Let $\mathsf{gc}(\mathsf{k})$ be any instantiated guarded command. Let $\varphi \in \mathsf{GSF}$ be any generic state formula, and $Init \in \mathsf{USF}$ be any universal state formula. Then $M(\mathsf{gc}(\mathsf{k}), Init)$ exhibits φ iff $M(\mathsf{gc}(\mathtt{1}^d), Init)$ exhibits φ.*

Proof. For the forward implication, let $\sigma_1, \sigma_2, \ldots, \sigma_w$ be a sequence of states of $M(\mathsf{gc}(\mathsf{k}), Init)$ such that:

$$\sigma_1 \models Init \bigwedge \sigma_w \models \varphi \bigwedge \forall i \in [1, w-1] \centerdot \{\sigma_i\}\ \mathsf{gc}(\mathsf{k})\ \{\sigma_{i+1}\}$$

Since $\varphi \in \mathsf{GSF}$, by Lemma 39 we know that:

$$\exists \mathsf{j} \in \otimes(\sigma_w^n) \centerdot \sigma_w \downharpoonleft \mathsf{j} \models \varphi$$

Let j_0 be such a j. By Lemma 40, since $Init \in \mathsf{USF}$:

$$\sigma_1 \downharpoonleft \mathsf{j}_0 \models Init$$

By Lemma 41, we know that:

$$\forall i \in [1, w-1] \centerdot \{\sigma_i \downharpoonleft \mathsf{j}_0\}\ \mathsf{gc}(\mathtt{1}^d)\ \{\sigma_{i+1} \downharpoonleft \mathsf{j}_0\}$$

Therefore, $\sigma_w \downharpoonleft \mathsf{j}_0$ is reachable in $M(\mathsf{gc}(\mathtt{1}^d), Init)$ and $\sigma_w \downharpoonleft \mathsf{j}_0 \models \varphi$. Hence, $M(\mathsf{gc}(\mathtt{1}^d), Init)$ exhibits φ. For the reverse implication, let $\sigma_1, \sigma_2, \ldots, \sigma_w$ be a sequence of states of $M(\mathsf{gc}(\mathtt{1}), \mathit{Init})$ such that:

$$\sigma_1 \models \mathit{Init} \bigwedge \sigma_w \models \varphi \bigwedge \forall i \in [1, w-1] \centerdot \{\sigma_i\}\ \mathsf{gc}(\mathtt{1}^d)\ \{\sigma_{i+1}\}$$

For each $i \in [1, w]$, let $\widehat{\sigma_i} = \sigma_i \upharpoonright \mathsf{k}$. Therefore, since $\mathit{Init} \in \mathsf{USF}$, by Lemma 40, we know:

$$\forall \mathsf{j} \in \otimes(\mathsf{k}) \centerdot \widehat{\sigma_1} \downharpoonright \mathsf{j} \models \mathit{Init} \Rightarrow \widehat{\sigma_1} \models \mathit{Init}$$

Also, since $\varphi \in \mathsf{GSF}$, by Lemma 42 we know that:

$$\forall \mathsf{j} \in \otimes(\mathsf{k}) \centerdot \widehat{\sigma_n} \downharpoonright \mathsf{j} \models \varphi \Rightarrow \widehat{\sigma_w} \models \varphi$$

Finally, by Lemma 43, we know that:

$$\forall i \in [1, w-1] \centerdot \{\widehat{\sigma_i}\}\ \mathsf{gc}(\mathsf{k})\ \{\widehat{\sigma_{i+1}}\}$$

Therefore, $\widehat{\sigma_w}$ is reachable in $M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$ and $\widehat{\sigma_w} \models \varphi$. Hence, $M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$ exhibits φ. This completes the proof.

$\square$

B.1.7 Proof of Theorem 10

Theorem 2 (Small Model Simulation)**.** *Let* $\mathsf{gc}(\mathsf{k})$ *be any instantiated guarded command. Let Init* $\in \mathsf{GSF}$ *be any generic state formula. Then* $M(\mathsf{gc}(\mathsf{k}), \mathit{Init}) \preceq M(\mathsf{gc}(\mathtt{1}^d), \mathit{Init})$ *and* $M(\mathsf{gc}(\mathtt{1}^d), \mathit{Init}) \preceq M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$.

Proof. Recall the conditions **C1–C3** in Definition 20 for simulation. For the first simulation, we propose the following relation $\mathcal{H}$ and show that it is a simulation relation:

$$(\sigma,\sigma') \in \mathcal{H} \Leftrightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma' = \sigma \downharpoonleft \mathsf{i}$$

C1 holds because our atomic propositions are USF formulas, and Lemma 40; **C2** holds because *Init* $\in$ GSF and Lemma 39; **C3** holds by Definition 12 and Lemma 41. For the second simulation, we propose the following relation $\mathcal{H}$ and show that it is a simulation relation:

$$(\sigma,\sigma') \in \mathcal{H} \Leftrightarrow \sigma' = \sigma \upharpoonleft \sigma'^n$$

Again, **C1** holds because our atomic propositions are USF formulas, Definition 19, and Lemma 40; **C2** holds because *Init* $\in$ GSF, Definition 19, and Lemma 42; **C3** holds by Definition 12 and Lemma 43. This completes the proof. $\square$

B.1.8 Proof of Corollary 11

We begin with the formal definition of simulation [20].

Definition 20. *Let $M_1 = (\mathcal{S}_1, I_1, \mathcal{T}_1, \mathcal{L}_1)$ an $M_2 = (\mathcal{S}_2, I_2, \mathcal{T}_2, \mathcal{L}_2)$ be two Kripke structures over sets of atomic propositions AP_1 and AP_2 such that $\mathsf{AP}_2 \subseteq \mathsf{AP}_1$. Then M_1 is simulated by M_2, denoted by $M_1 \preceq M_2$, iff there exists a relation $H \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ such that the following three conditions hold:*

(C1) $\forall s_1 \in \mathcal{S}_1 \centerdot \forall s_2 \in \mathcal{S}_2 \centerdot (s_1, s_2) \in H \Rightarrow \mathcal{L}_1(s_1) \cap \mathsf{AP}_2 = \mathcal{L}_2(s_2)$

(C2) $\forall s_1 \in I_1 \centerdot \exists s_2 \in I_2 \centerdot (s_1, s_2) \in H$

(C3) $\forall s_1, s_1' \in \mathcal{S}_1 \centerdot \forall s_2 \in \mathcal{S}_2 \centerdot (s_1, s_2) \in H \wedge (s_1, s_1') \in \mathcal{T}_1 \Rightarrow$
$\exists s_2' \in \mathcal{S}_2 \centerdot (s_2, s_2') \in \mathcal{T}_2 \wedge (s_1', s_2') \in H$

Next, we formalize our claim that PTSL formulas are preserved by simulation.

Fact 3. *Let M_1 and M_2 be two Kripke structures over propositions* AP_1 *and* AP_2 *such that* $M_1 \preceq M_2$. *Hence, by Definition 20,* $\mathsf{AP}_2 \subseteq \mathsf{AP}_1$. *Let φ be any* PTSL *formula over* AP_2. *Therefore, φ is also a* PTSL *formula over* AP_1. *Then* $M_2 \models \varphi \Rightarrow M_1 \models \varphi$.

We are now ready to prove Corollary 11.

Corollary 3 (Small Model Safety 2)**.** *Let* $\mathtt{gc(k)}$ *be any instantiated guarded command. Let* $\varphi \in \mathsf{USF}$ *be any universal state formula, and* $Init \in \mathsf{GSF}$ *be any generic state formula. Then* $M(\mathtt{gc(k)}, Init)$ *exhibits φ iff* $M(\mathtt{gc(1}^{d}\mathtt{)}, Init)$ *exhibits φ.*

Proof. Follows from: (i) the observation that exhibition of a USF formula ϕ is expressible in PTSL as the TLF formula $\mathbf{F}\ \phi$, (ii) Theorem 10, and (iii) Fact 3. □

Appendix C

Proofs of Small Model Theorems with Write-only Variables

C.1 Proofs

This appendix contains proofs of our small model theorems and supporting lemmas. We begin with some introductory lemmas.

C.1.1 Introductory Lemmas

First we prove that projection does not affect the evaluation of expressions, as expressed by the following lemma.

Lemma 4. *Let* $\mathsf{e} \in \mathtt{E}$*,* $t \in \mathbb{B}$*,* $z \in [1,d]$*, and* σ *be any store. Then:*

$$\langle \mathsf{e}, \sigma \rangle \rightarrow t \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i} \rangle \rightarrow t$$

$$\Leftrightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i} \rangle \rightarrow t$$

Proof. Follows from the fact that e depends only on Boolean variables, and the following observation:

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma^B = (\sigma \downharpoonleft \mathsf{i})^B$$

□

Next we prove an analogous result that generalization does not affect the evaluation of expressions, as expressed by the following lemma.

Lemma 5. *Let* $\mathsf{e} \in \mathtt{E}$*,* $t \in \mathbb{B}$*,* $z \in [1,d]$*, and* σ *be any store. Then:*

$$\langle \mathsf{e}, \sigma \rangle \to t \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \langle \mathsf{e}, \sigma \upharpoonleft \mathsf{i} \rangle \to t$$

$$\Leftrightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \langle \mathsf{e}, \sigma \upharpoonleft \mathsf{i} \rangle \to t$$

Proof. Follows from the fact that e depends only on Boolean variables, and the following observation:

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma^B = (\sigma \upharpoonleft \mathsf{i})^B$$

□

C.1.2 Store Projection Lemmas

We now present a series of lemmas that explore the relationship between store projection and different types of formulas. These lemmas will be used later to prove our small model theorems. The first lemma states that a store σ satisfies a basic proposition π iff every projection of σ satisfies π.

Lemma 6. *Let* $z \in [1,d]$*,* $\pi \in \mathsf{BP}$*, and* σ *be any store. Then:*

$$\sigma \models \pi \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \bullet \sigma \downharpoonleft \mathsf{i} \models \pi$$
$$\Leftrightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,z}) \bullet \sigma \downharpoonleft \mathsf{i} \models \pi$$

Proof. Follows from the fact that π depends only on Boolean variables, and the following observation:

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,z}) \bullet \sigma^B = (\sigma \downharpoonleft \mathsf{i})^B$$

□

The next lemma states that a store σ satisfies an existentially quantified formula π iff some projection of σ satisfies π.

Lemma 7. *Let* $z \in [1,d]$*,* $\pi = \text{Æ}_1 \mathtt{i}_1 \dots \text{Æ}_z \mathtt{i}_z \bullet \pi'$ *and* $\pi' \in \mathsf{PP}(\mathtt{i}_1, \dots, \mathtt{i}_z)$ *and* σ *be any store. Then:*

$$\sigma \models \pi \Leftrightarrow \text{Æ}_1 i_1 \in [1,\sigma^n_1] \dots \text{Æ}_z i_z \in [1,\sigma^n_z] \bullet \sigma \downharpoonleft (i_1,\dots,i_z) \models \pi$$

Proof. By Definition 11, we know that:

$$\sigma \models \pi \Leftrightarrow \exists \text{Æ}_1 i_1 \in [1,\sigma^n_1] \dots \text{Æ}_z i_z \in [1,\sigma^n_z] \bullet$$

$$\sigma \downharpoonleft (i_1,\dots,i_z) \models \pi'[\mathtt{i}_1 \mapsto 1] \dots [\mathtt{i}_z \mapsto 1]$$

$\triangleright$ let $\mathsf{i} = (i_1,\dots,i_z)$; from Definition 16, since $((\sigma \downharpoonleft \mathsf{i}) \downharpoonleft 1^z) = (\sigma \downharpoonleft \mathsf{i})$

$$\Leftrightarrow \text{Æ}_1 i_1 \in [1,\sigma^n_1] \dots \text{Æ}_z i_z \in [1,\sigma^n_z] \bullet$$

$$(\sigma \downharpoonleft \mathsf{i}) \downharpoonleft 1^z \models \pi'[\mathtt{i}_1 \mapsto 1] \dots [\mathtt{i}_z \mapsto 1]$$

$\triangleright$ let $\mathsf{j} = (j_1,\dots,j_z)$ be fresh variables; since $(\sigma \downharpoonleft i)^n_{1,z} = 1^z$

$$\Leftrightarrow \text{Æ}_1 i_1 \in [1,\sigma_1^n] \ldots \text{Æ}_z i_z \in [1,\sigma_z^n] \centerdot$$

$$\text{Æ}_1 j_1 \in [1,(\sigma \downharpoonleft \mathsf{i})_1^n] \ldots \text{Æ}_z j_z \in [1,(\sigma \downharpoonleft \mathsf{i})_z^n] \centerdot$$

$$(\sigma \downharpoonleft \mathsf{i}) \downharpoonleft \mathsf{j} \models \pi'[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_z \mapsto 1]$$

$\triangleright$ let $\mathtt{j}_1,\ldots,\mathtt{j}_z$ be fresh variables

$$\Leftrightarrow \text{Æ}_1 i_1 \in [1,\sigma_1^n] \ldots \text{Æ}_z i_z \in [1,\sigma_z^n] \centerdot$$

$$\text{Æ}_1 j_1 \in [1,(\sigma \downharpoonleft \mathsf{i})_1^n] \ldots \text{Æ}_z j_z \in [1,(\sigma \downharpoonleft \mathsf{i})_z^n] \centerdot$$

$$(\sigma \downharpoonleft \mathsf{i}) \downharpoonleft \mathsf{j} \models \pi'[\mathtt{i}_1 \mapsto \mathtt{j}_1] \ldots [\mathtt{i}_z \mapsto \mathtt{j}_z][\mathtt{j}_1 \mapsto 1] \ldots [\mathtt{j}_z \mapsto 1]$$

$\triangleright$ by Definition 11

$$\Leftrightarrow \text{Æ}_1 i_1 \in [1,\sigma_1^n] \ldots \text{Æ}_z i_z \in [1,\sigma_z^n] \centerdot$$

$$\sigma \downharpoonleft \mathsf{i} \models \text{Æ}_1 \mathtt{j}_1 \ldots \text{Æ}_z \mathtt{j}_z \centerdot (\pi'[\mathtt{i}_1 \mapsto \mathtt{j}_1] \ldots [\mathtt{i}_z \mapsto \mathtt{j}_z])$$

By alpha renaming, we know that $\text{Æ}_1 \mathtt{j}_1 \ldots \text{Æ}_z \mathtt{j}_z \centerdot (\pi'[\mathtt{i}_1 \mapsto \mathtt{j}_1] \ldots [\mathtt{i}_z \mapsto \mathtt{j}_z])$ is equivalent to π. This gives us our desired result. $\square$

The next lemma states that a store σ satisfies an existential state formula φ iff certain projection of σ satisfy φ.

Lemma 8. *Let* $\varphi \in \mathsf{ESF}$ *and* σ *be any store. Then:*

$$\sigma \models \varphi \Rightarrow \exists \mathsf{i} \in \otimes(\sigma_{1,d}^n) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi$$

Proof. From Figure 4.6, we consider three sub-cases:

Case 1. $\varphi \in \mathsf{BP}$. Our result follows from Lemma 29.

Case 2. $\varphi = \text{Æ}_1 \mathtt{i}_1 \ldots \text{Æ}_z \mathtt{i}_z \centerdot \pi$ and $\pi \in \mathsf{PP}(\mathtt{i}_1,\ldots,\mathtt{i}_z)$. Our result follows from Lemma 7.

Case 3. $\varphi = \pi \wedge Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$ such that $\pi \in \mathsf{BP}$, and $\pi' \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. By Definition 11:

$$\sigma \models \varphi \Rightarrow \sigma \models \pi \bigwedge \sigma \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\rhd \text{ by Lemma 6 and Lemma 7}$$

$$\Rightarrow \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \bigwedge$$

$$\exists \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\rhd \text{ playing around with } \wedge, \forall, \text{ and } \exists$$

$$\Rightarrow \exists \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \wedge$$

$$\sigma \downharpoonleft \mathsf{j} \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\rhd \text{ Again by Definition 11}$$

$$\Rightarrow \exists \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \wedge Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

By alpha renaming j to i, we get our desired result. $\square$

Lemma 9. *Let* $\varphi \in \mathsf{ESF}$ *and* σ *be any store. Then:*

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi \Rightarrow \sigma \models \varphi$$

Proof. From Figure 4.6, we consider three sub-cases:

Case 1. $\varphi \in \mathsf{BP}$. Our result follows from Lemma 29.

Case 2. $\varphi = Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi$ and $\pi \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. Our result follows from Lemma 7.

Case 3. $\varphi = \pi \wedge Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$ such that $\pi \in \mathsf{BP}$, and $\pi' \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$.

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \pi \wedge Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ by Definition 11}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \pi \wedge$$

$$\sigma \downharpoonleft \mathsf{i} \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ playing around with } \wedge \text{ and } \forall$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \pi \bigwedge$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ by Lemma 6 and Lemma 7}$$

$$\Rightarrow \sigma \models \pi \bigwedge \sigma \models Æ_1 \mathtt{i}_1 \ldots Æ_z \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ Again by Definition 11}$$

$$\Rightarrow \sigma \models \varphi$$

This completes our proof. □

C.1.3 Store Projection and Command Lemmas

The following lemma states that if a store σ is transformed to σ' by executing a command $\mathtt{c}_1$, then every projection of σ is transformed to the corresponding projection of σ' by executing $\mathtt{c}_1$.

Lemma 10 (Command Projection)**.** *For $z \in [1,d]$, any stores σ, σ' such that $\sigma^n_{1,z-1} = \sigma'^n_{1,z-1} = 1^{z-1}$, and any command $\mathtt{c} \in \mathtt{C}_z[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_{z-1} \mapsto 1]$:*

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathtt{c}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Proof. By induction on the structure of $\mathtt{c}$. We consider four subcases.

Case 0. $\mathtt{c} \triangleq \mathtt{skip}$. Trivially, since by Figure 6.3:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \sigma' = \sigma$$

Case 1. $\mathtt{c} \triangleq \mathtt{b} := \mathtt{e}$. Suppose $\langle \mathtt{e}, \sigma \rangle \to t$. By Figure 6.3:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \sigma' = \sigma[\sigma^B \mapsto \sigma^B[\mathsf{b} \mapsto t]] \Rightarrow$$

$$\rhd \text{ by Definition 16 and Lemma 4, } \forall \mathsf{i} \in \otimes(\sigma^n_{1,d})$$

$$\sigma' \downharpoonleft \mathsf{i} = (\sigma \downharpoonleft \mathsf{i})[(\sigma \downharpoonleft \mathsf{i})^B \mapsto (\sigma \downharpoonleft \mathsf{i})^B[\mathsf{b} \mapsto t]] \wedge \langle \mathtt{e}, \sigma \downharpoonleft \mathsf{i} \rangle \to t$$

$$\rhd \text{ Again by Figure 6.3}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d})\,.\,\{\sigma \downharpoonleft \mathsf{i}\}\ \mathtt{c}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 2. $\mathtt{c} \triangleq \mathtt{for}\ \mathtt{i}_z\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2$. Let $\sigma^n_z = N$. By Figure 6.3:

$$\forall y \in [1,N]\,.\,\{\sigma \downharpoonleft (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft (1^{z-1}, y)\}$$

$$\rhd \text{ By Lemma 11}$$

$$\forall y \in [1,N]\,.\,\forall \mathsf{i} \in \otimes(1^z, \sigma^n_{z+1,d})\,.$$

$$\{\sigma \downharpoonleft (1^{z-1}, y) \downharpoonleft \mathsf{i}\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft (1^{z-1}, y) \downharpoonleft \mathsf{i}\}$$

$$\rhd \text{ Combining the two } \forall \text{ quantifiers}$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d})\,.$$

$$\{\sigma \downharpoonleft \mathsf{i}\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Expanding out

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \,\boldsymbol{.}\, \forall y \in [1,1] \boldsymbol{.}$$

$$\{\sigma \downharpoonleft \mathsf{i} \downharpoonleft (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft \mathsf{i} \downharpoonleft (1^{z-1}, y)\}$$

$\triangleright$ Again by Figure 6.3

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \,\boldsymbol{.}\, \{\sigma \downharpoonleft \mathsf{i}\}\ \mathtt{c}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 3. $\mathtt{c} \triangleq \mathtt{c}_1; \mathtt{c}_2$. By Figure 6.3:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \exists \sigma'' \,\boldsymbol{.}\, \{\sigma\}\ \mathtt{c}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \mathtt{c}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\exists \sigma'' \,\boldsymbol{.}\, \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \,\boldsymbol{.}\, \{\sigma \downharpoonleft \mathsf{i}\}\ \mathtt{c}_1\ \{\sigma'' \downharpoonleft \mathsf{i}\} \wedge$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \,\boldsymbol{.}\, \{\sigma'' \downharpoonleft \mathsf{i}\}\ \mathtt{c}_2\ \{\sigma' \downharpoonleft \mathsf{i}\} \Rightarrow$$

$\triangleright$ Swapping $\forall$ and $\exists$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \,\boldsymbol{.}\, \exists \sigma'' \boldsymbol{.}$$

$$\{\sigma \downharpoonleft \mathsf{i}\}\ \mathtt{c}_1\ \{\sigma'' \downharpoonleft \mathsf{i}\} \wedge \{\sigma'' \downharpoonleft \mathsf{i}\}\ \mathtt{c}_2\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Again by Figure 6.3

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \,\boldsymbol{.}\, \{\sigma \downharpoonleft \mathsf{i}\}\ \mathtt{c}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

This completes the proof. □

Lemma 11 (Parameterized Command Projection)**.** *For* $z \in [1,d]$*, any stores* σ, σ' *such that*

$\sigma^{n}_{1,z} = \sigma'^{n}_{1,z} = 1^z$*, and any command* $\widehat{\mathsf{e}}\ ?\ \widehat{\mathtt{c}} : \widehat{\mathtt{c}}' \in (\widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z : \widehat{\mathtt{C}}_z)[\mathtt{i}_1 \mapsto 1]\ldots[\mathtt{i}_z \mapsto 1]$*:*

$$\{\sigma\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathtt{c}} : \widehat{\mathtt{c}}'\ \{\sigma'\} \Rightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \{\sigma \downharpoonright \mathsf{i}\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathtt{c}} : \widehat{\mathtt{c}}'\ \{\sigma' \downharpoonright \mathsf{i}\}$$

Proof. First consider the case $\langle \widehat{\mathsf{e}}, \sigma \rangle \to \textbf{true}$. Note that $\widehat{\mathsf{e}}$ does not contain any index variables and only refers to the parametric array at depth z. Since $\sigma^{n}_{1,z} = \sigma'^{n}_{1,z} = 1^z$, we claim that:

$$\langle \widehat{\mathsf{e}}, \sigma \rangle \to \textbf{true} \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \langle \widehat{\mathsf{e}}, \sigma \downharpoonright \mathsf{i} \rangle \to \textbf{true}$$

$$\Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \langle \widehat{\mathsf{e}}, \sigma' \downharpoonright \mathsf{i} \rangle \to \textbf{true} \Leftrightarrow \langle \widehat{\mathsf{e}}, \sigma' \rangle \to \textbf{true}$$

Now by Figure 6.3, $\{\sigma\}\ \widehat{\mathtt{c}}\ \{\sigma'\}$. We proceed by induction on the structure of $\widehat{\mathtt{c}}$. We consider three subcases.

Case 1. $\widehat{\mathtt{c}} \triangleq \mathtt{P[1]}\ldots\mathtt{P[1].F[r]} := \widehat{\mathsf{e}}'$. Suppose $\langle \widehat{\mathsf{e}}', \sigma \rangle \to t$. Since $\widehat{\mathsf{e}}'$ only refers to the parametric array at depth z and $\sigma^{n}_{1,z} = \sigma'^{n}_{1,z} = 1^z$. By Figure 6.3:

$$\sigma' = \sigma[\sigma^P \mapsto \sigma^P[\sigma^P_z \mapsto [\sigma^P_z[(1^z, \lceil \mathtt{r} \rceil) \mapsto t]]]]$$

$$\triangleright \text{ By Definition 16, } \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d})$$

$$\sigma' \downharpoonright \mathsf{i} = (\sigma \downharpoonright \mathsf{i})[(\sigma \downharpoonright \mathsf{i})^P \mapsto (\sigma \downharpoonright \mathsf{i})^P[(\sigma \downharpoonright \mathsf{i})^P_z \mapsto [(\sigma \downharpoonright \mathsf{i})^P_z[(1^z, \lceil \mathtt{r} \rceil) \mapsto t]]]] \wedge$$

$$\langle \widehat{\mathsf{e}}', \sigma \downharpoonright \mathsf{i} \rangle \to t$$

$$\triangleright \text{ Again by Figure 6.3}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^{n}_{1,d}) \centerdot \{\sigma \downharpoonright \mathsf{i}\}\ \widehat{\mathtt{c}}\ \{\sigma' \downharpoonright \mathsf{i}\}$$

Case 2. $\widehat{\mathsf{c}} \triangleq \mathtt{w} := \widehat{\mathsf{e}}'$. Suppose $\langle \widehat{\mathsf{e}}', \sigma \rangle \to t$. By Figure 6.3:

$$\sigma' = \sigma[\sigma^W \mapsto \sigma^W[\mathtt{w} \mapsto t]] \Rightarrow$$

$$\triangleright \text{ by Definition 16, } \forall \mathsf{i} \in \otimes(\sigma^n_{1,d})$$

$$\sigma' \downharpoonleft \mathsf{i} = (\sigma \downharpoonleft \mathsf{i})[(\sigma \downharpoonleft \mathsf{i})^W \mapsto (\sigma \downharpoonleft \mathsf{i})^W[\mathtt{w} \mapsto t]] \wedge \langle \widehat{\mathsf{e}}', \sigma \downharpoonleft \mathsf{i} \rangle \to t$$

$$\triangleright \text{ Again by Figure 6.3}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 3. $\widehat{\mathsf{c}} \triangleq \widehat{\mathsf{c}}_1; \widehat{\mathsf{c}}_2$. By Figure 6.3:

$$\{\sigma\}\ \widehat{\mathsf{c}}\ \{\sigma'\} \Rightarrow \exists \sigma'' \centerdot \{\sigma\}\ \widehat{\mathsf{c}}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \widehat{\mathsf{c}}_2\ \{\sigma'\} \Rightarrow$$

$$\triangleright \text{ By inductive hypothesis}$$

$$\exists \sigma'' \centerdot \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}_1\ \{\sigma'' \downharpoonleft \mathsf{i}\} \wedge$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma'' \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}_2\ \{\sigma' \downharpoonleft \mathsf{i}\} \Rightarrow$$

$$\triangleright \text{ Swapping } \forall \text{ and } \exists$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \exists \sigma'' \centerdot$$

$$\{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}_1\ \{\sigma'' \downharpoonleft \mathsf{i}\} \wedge \{\sigma'' \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}_2\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$$\triangleright \text{ Again by Figure 6.3}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 4. $\widehat{\mathsf{c}} \triangleq \mathsf{c}$. Follows directly from Lemma 10. Since $\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \langle \widehat{\mathsf{e}}, \sigma \downharpoonleft \mathsf{i} \rangle \to$ **true** and $\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{c}}\ \{\sigma' \downharpoonleft \mathsf{i}\}$, from Figure 6.3, we have our desired result $\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \widehat{\mathsf{e}} \mathbin{?} \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma' \downharpoonleft \mathsf{i}\}$.

The proof for the case when $\langle \widehat{\mathsf{e}}, \sigma \rangle \rightarrow$ **false** is analogous. This completes the proof. $\square$

Note that the dependence between Lemma 10 and Lemma 11 is not circular since for Lemma 11 to be valid for z, Lemma 10 must be valid for $z+1$, and for $z = d$, Lemma 11 does not require Lemma 10 to be valid. Hence we can argue using mutual recursion, starting from $z = d$, and prove that both lemmas are valid for all $z \in [1,d]$.

The next lemma states that if a store σ is transformed to σ' by executing a guarded command gc, then every projection of σ is transformed to the corresponding projection of σ' by executing gc.

Lemma 12 (Guarded Command Projection)**.** *For any stores σ, σ', and any guarded command* $\mathsf{gc} \in \mathtt{GC}$*:*

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Proof. We consider two cases.

Case 1. $\mathsf{gc} \triangleq \mathsf{e} \mathbin{?} \mathsf{c}_1 : \mathsf{c}_2$. By Figure 6.3, we have two sub-cases:

Case 1.1. In this case:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma \rangle \rightarrow \textbf{true} \wedge \{\sigma\}\ \mathsf{c}_1\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By Lemma 4 and Lemma 10

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i} \rangle \rightarrow \textbf{true} \wedge$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{c}_1\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Since $\wedge$ distributes over $\forall$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \langle \mathsf{e}, \sigma \downharpoonleft \mathsf{i}\rangle \rightarrow \textbf{true} \wedge \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{c}_1\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$$\triangleright \text{ Again by Figure 6.3}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Case 1.2. In this case:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma\rangle \rightarrow \textbf{false} \wedge \{\sigma\}\ \mathsf{c}_2\ \{\sigma'\}$$

The proof is analogous to the previous sub-case.

Case 2. $\mathsf{gc} \triangleq \mathsf{gc}_1 \parallel \mathsf{gc}_2$. By Figure 6.3:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \{\sigma\}\ \mathsf{gc}_1\ \{\sigma'\} \vee \{\sigma\}\ \mathsf{gc}_2\ \{\sigma'\} \Rightarrow$$

$$\triangleright \text{ By inductive hypothesis}$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}_1\ \{\sigma' \downharpoonleft \mathsf{i}\} \bigvee$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}_2\ \{\sigma' \downharpoonleft \mathsf{i}\} \Rightarrow$$

$$\triangleright \text{ Playing around with } \vee \text{ and } \forall$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot$$

$$\{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}_1\ \{\sigma' \downharpoonleft \mathsf{i}\} \vee \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}_2\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$$\triangleright \text{ Again by Figure 6.3}$$

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

This completes the proof. □

C.1.4 Store Generalization Lemmas

We now present a series of lemmas that relate the execution semantics of *PGCL*$^{++}$ to store generalization. The first lemma states that if a store σ is transformed to σ' by executing a command $\mathtt{c}$, then every generalization of σ is transformed to the corresponding generalization of σ' by executing $\mathtt{c}$.

Lemma 13 (Command Generalization)**.** *For* $z \in [1,d]$*, any stores* σ, σ' *such that* $\sigma^n = \sigma'^n = 1^d$*, any* $\mathsf{k} \in \mathbb{N}^d$ *such that* $\mathsf{k}_{1,z-1} = 1^{z-1}$*, and any command* $\mathtt{c} \in \mathtt{C}_z[\mathtt{i}_1 \mapsto 1] \ldots [\mathtt{i}_{z-1} \mapsto 1]$*:*

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \{\sigma \upharpoonleft \mathsf{k}\}\ \mathtt{c}\ \{\sigma' \upharpoonleft \mathsf{k}\}$$

Proof. By induction on the structure of $\mathtt{c}$. We consider four subcases.

Case 0. $\mathtt{c} \triangleq \mathtt{skip}$. Trivially since by Figure 6.3, $\sigma = \sigma'$.

Case 1. $\mathtt{c} \triangleq \mathtt{b} := \mathtt{e}$. Suppose $\langle \mathtt{e}, \sigma \rangle \to t$. By Figure 6.3:

$$\{\sigma\}\ \mathtt{c}\ \{\sigma'\} \Rightarrow \sigma' = \sigma[\sigma^B \mapsto \sigma^B[\mathtt{b} \mapsto t]] \Rightarrow$$

$\triangleright$ by Definition 21 and Lemma 5

$$\sigma' \upharpoonleft \mathsf{k} = (\sigma \upharpoonleft \mathsf{k})[(\sigma \upharpoonleft \mathsf{k})^B \mapsto (\sigma \upharpoonleft \mathsf{k})^B[\mathtt{b} \mapsto t]] \wedge \langle \mathtt{e}, \sigma \upharpoonleft \mathsf{k} \rangle \to t$$

$\triangleright$ again by Figure 6.3

$$\Rightarrow \{\sigma \upharpoonleft \mathsf{k}\}\ \mathtt{c}\ \{\sigma' \upharpoonleft \mathsf{k}\}$$

Case 2. $\mathtt{c} \triangleq \mathtt{for}\ \mathtt{i}_z\ \mathtt{do}\ \widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2$. By Figure 6.3:

$$\forall y \in [1,1] \centerdot \{\sigma \downharpoonleft (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \downharpoonleft (1^{z-1}, y)\}$$

$\triangleright$ Simplifying

$$\{\sigma\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma'\}$$

$\triangleright$ By Lemma 14

$$\{\sigma \upharpoonright \mathsf{k}[z \mapsto 1]\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \upharpoonright \mathsf{k}[z \mapsto 1]\}$$

$\triangleright$ Let $\mathsf{k}_z = N$. Expanding out

$$\forall y \in [1, N] \centerdot \{\sigma \upharpoonright \mathsf{k} \downharpoonright (1^{z-1}, y)\}\ (\widehat{\mathtt{e}}\ ?\ \widehat{\mathtt{c}}_1 : \widehat{\mathtt{c}}_2)[\mathtt{i}_z \mapsto 1]\ \{\sigma' \upharpoonright \mathsf{k} \downharpoonright (1^{z-1}, y)\}$$

$\triangleright$ Again by Figure 6.3

$$\Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 3. $\mathsf{c} \triangleq \mathsf{c}_1; \mathsf{c}_2$. By Figure 6.3:

$$\{\sigma\}\ \mathsf{c}\ \{\sigma'\} \Rightarrow \exists \sigma'' \centerdot \{\sigma\}\ \mathsf{c}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \mathsf{c}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\exists \sigma'' \centerdot \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}_1\ \{\sigma'' \upharpoonright \mathsf{k}\} \wedge$$

$$\forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma'' \upharpoonright \mathsf{k}\}\ \mathsf{c}_2\ \{\sigma' \upharpoonright \mathsf{k}\} \Rightarrow$$

$\triangleright$ Playing around with $\exists$, $\forall$, and $\wedge$

$$\forall \mathsf{k} \in \mathbb{N}^d \centerdot \exists \sigma'' \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}_1\ \{\sigma'' \upharpoonright \mathsf{k}\} \wedge \{\sigma'' \upharpoonright \mathsf{k}\}\ \mathsf{c}_2\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ Again by Figure 6.3

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

This completes the proof. $\square$

Lemma 14 (Parameterized Command Generalization)**.** *For $z \in [1,d]$, any stores σ, σ' such that $\sigma^n = \sigma'^n = 1^d$, any $\mathsf{k} \in \mathbb{N}^d$ such that $\mathsf{k}_{1,z} = 1^z$, and any command $\widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}' \in (\widehat{\mathtt{E}}_z\ ?\ \widehat{\mathtt{C}}_z : \widehat{\mathtt{C}}_z)[\mathtt{i}_1 \mapsto 1]\ldots[\mathtt{i}_z \mapsto 1]$:*

$$\{\sigma\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma'\} \Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Proof. First consider the case $\langle \widehat{\mathsf{e}}, \sigma \rangle \to \mathbf{true}$. Note that $\widehat{\mathsf{e}}$ does not contain any index variables and only refers to the parametric array at depth z. Since $\sigma^n_{1,z} = \sigma'^n_{1,z} = 1^z$, we claim that:

$$\langle \widehat{\mathsf{e}}, \sigma \rangle \to \mathbf{true} \Leftrightarrow \langle \widehat{\mathsf{e}}, \sigma \upharpoonright \mathsf{k} \rangle \to \mathbf{true}$$

$$\Leftrightarrow \langle \widehat{\mathsf{e}}, \sigma' \upharpoonright \mathsf{k} \rangle \to \mathbf{true} \Leftrightarrow \langle \widehat{\mathsf{e}}, \sigma' \rangle \to \mathbf{true}$$

Now by Figure 6.3, $\{\sigma\}\ \widehat{\mathsf{c}}\ \{\sigma'\}$. We proceed by induction on the structure of $\widehat{\mathsf{c}}$. We consider three subcases.

Case 0. $\widehat{\mathsf{c}} \triangleq \mathtt{P[1]}\ldots\mathtt{P[1].F[r]} := \widehat{\mathsf{e}}'$. Suppose $\langle \widehat{\mathsf{e}}', \sigma \rangle \to t$. Since $\widehat{\mathsf{e}}'$ only refers to the parametric array at depth z and $\sigma^n_{1,z} = \sigma'^n_{1,z} = 1^z$. By Figure 6.3:

$$\sigma' = \sigma[\sigma^P \mapsto \sigma^P[\sigma^P_z \mapsto [\sigma^P_z[(1^z, \lceil \mathtt{r} \rceil) \mapsto t]]]]$$

$\triangleright$ by Definition 21

$$\sigma' \upharpoonright \mathsf{k} = (\sigma \upharpoonright \mathsf{k})[(\sigma \upharpoonright \mathsf{k})^P \mapsto (\sigma \upharpoonright \mathsf{k})^P[(\sigma \upharpoonright \mathsf{k})^P_z \mapsto [(\sigma \upharpoonright \mathsf{k})^P_z[(1^z, \lceil \mathtt{r} \rceil) \mapsto t]]]] \wedge$$

$$\langle \widehat{\mathsf{e}}, \sigma \upharpoonright \mathsf{k} \rangle \to t$$

$\triangleright$ Again by Figure 6.3

$$\Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 1. $\widehat{\mathsf{c}} \triangleq \mathtt{w} := \widehat{\mathsf{e}}'$. Suppose $\langle \widehat{\mathsf{e}}', \sigma\rangle \to t$. By Figure 6.3:

$$\sigma' = \sigma[\sigma^W \mapsto \sigma^W[\mathtt{w} \mapsto t]] \Rightarrow$$

$$\triangleright \text{ by Definition 21}$$

$$\sigma' \upharpoonright \mathsf{k} = (\sigma \upharpoonright \mathsf{k})[(\sigma \upharpoonright \mathsf{k})^W \mapsto (\sigma \upharpoonright \mathsf{k})^W[\mathtt{w} \mapsto t]] \wedge \langle \widehat{\mathsf{e}}', \sigma \upharpoonright \mathsf{k}\rangle \to t$$

$$\triangleright \text{ Again by Figure 6.3}$$

$$\Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 2. $\widehat{\mathsf{c}} \triangleq \widehat{\mathsf{c}}_1; \widehat{\mathsf{c}}_2$. By Figure 6.3:

$$\{\sigma\}\ \widehat{\mathsf{c}}\ \{\sigma'\} \Rightarrow \exists \sigma'' \centerdot \{\sigma\}\ \widehat{\mathsf{c}}_1\ \{\sigma''\} \wedge \{\sigma''\}\ \widehat{\mathsf{c}}_2\ \{\sigma'\} \Rightarrow$$

$$\triangleright \text{ By inductive hypothesis}$$

$$\exists \sigma'' \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}_1\ \{\sigma'' \upharpoonright \mathsf{k}\} \wedge \{\sigma'' \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}_2\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$$\triangleright \text{ Again by Figure 6.3}$$

$$\Rightarrow \{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 3. $\widehat{\mathsf{c}} \triangleq \mathsf{c}$. Follows directly from Lemma 13.

Since $\langle \widehat{\mathsf{e}}, \sigma \upharpoonright \mathsf{k}\rangle \to$ **true** and $\{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{c}}\ \{\sigma' \upharpoonright \mathsf{k}\}$, from Figure 6.3, we have our desired result $\{\sigma \upharpoonright \mathsf{k}\}\ \widehat{\mathsf{e}}\ ?\ \widehat{\mathsf{c}} : \widehat{\mathsf{c}}'\ \{\sigma' \upharpoonright \mathsf{k}\}$.

The proof for the case when $\langle \widehat{\mathsf{e}}, \sigma\rangle \to$ **false** is analogous. This completes the proof. □

Note that the dependence between Lemma 13 and Lemma 14 is not circular since for

Lemma 14 to be valid for z, Lemma 13 must be valid for $z+1$, and for $z = d$, Lemma 14 does not require Lemma 36 to be valid. Hence we can argue using mutual recursion, starting from $z = d$, and prove that both lemmas are valid for all $z \in [1,d]$.

The next lemma states that if a store σ is transformed to σ' by executing a guarded command gc, then every generalization of σ is transformed to the corresponding generalization of σ' by executing gc.

Lemma 15 (Guarded Command Generalization)**.** *For any stores* σ, σ' *such that* $\sigma^n = \sigma'^n = 1^d$*, and any guarded command* $\mathsf{gc} \in \mathtt{GC}$*:*

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Proof. We consider two cases.

Case 1. $\mathsf{gc} \triangleq \mathsf{e}\ ?\ \mathsf{c}_1 : \mathsf{c}_2$. By Figure 6.3, we have two sub-cases:

Case 1.1. In this case:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma\rangle \to \textbf{true} \wedge \{\sigma\}\ \mathsf{c}_1\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By Lemma 5 and Lemma 13

$$\forall \mathsf{k} \in \mathbb{N}^d \centerdot \langle \mathsf{e}, \sigma \upharpoonright \mathsf{k}\rangle \to \textbf{true} \wedge \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}_1\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ Since $\wedge$ distributes over $\forall$

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \langle \mathsf{e}, \sigma \upharpoonright \mathsf{k}\rangle \to \textbf{true} \wedge \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{c}_1\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ Again by Figure 6.3

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

Case 1.2. In this case:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \langle \mathsf{e}, \sigma\rangle \rightarrow \textbf{false} \wedge \{\sigma\}\ \mathsf{c}_2\ \{\sigma'\}$$

The proof is analogous to the previous sub-case.

Case 2. $\mathsf{gc} \triangleq \mathsf{gc}_1 \parallel \mathsf{gc}_2$. By Figure 6.3:

$$\{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow \{\sigma\}\ \mathsf{gc}_1\ \{\sigma'\} \vee \{\sigma\}\ \mathsf{gc}_2\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By inductive hypothesis

$$\forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}_1\ \{\sigma' \upharpoonright \mathsf{k}\} \bigvee$$

$$\forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}_2\ \{\sigma' \upharpoonright \mathsf{k}\} \Rightarrow$$

$\triangleright$ Playing around with $\vee$ and $\forall$

$$\forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}_1\ \{\sigma' \upharpoonright \mathsf{k}\} \vee \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}_2\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ Again by Figure 6.3

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

This completes the proof. □

C.1.5 Proofs of Small Model Theorems

In this section, we prove our small model theorems. We first present a set of supporting lemmas for the proof of Theorem 14. In some cases, the proof of the lemma is in the appendix. In addition, the proofs of these lemmas rely on other lemmas, which are in the appendix.

In the following proofs, for $z \in [1,d]$, and any tuple $\mathsf{i}^z = (i_1, \ldots, i_z)$, we write $Æ\mathsf{i}^z$ to mean $Æ_1 \mathtt{i}_1, \ldots, Æ_z \mathtt{i}_z$ where the tuple of quantifiers $(Æ_1, \ldots, Æ_z)$ (where for each z, $Æ$ is either $\forall$ or $\exists$) is fixed and remains fixed across all instances of $Æ$ in the same line of reasoning. This ensures consistency between all $Æ\mathsf{i}^z$ and $Æj^z$. Note that our results hold for any combination of quantifiers as long as they are consistent.

The first lemma states that if a store σ satisfies a generic state formula φ, then some projection of σ satisfies φ.

Lemma 16. *Let* $\varphi \in \mathsf{GSF}$*, and* σ *be any store. Then:*

$$\sigma \models \varphi \Rightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi$$

Proof. By considering the structure of φ. We consider three cases:

Case 1. $\varphi \in \mathsf{USF}$. By Lemma 17, we know that:

$$\sigma \models \varphi \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi \Rightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi$$

Case 2. $\varphi \in \mathsf{ESF}$. Follows directly from Lemma 8.

Case 3. $\varphi \in \mathsf{USF} \wedge \mathsf{ESF}$. Without loss of generality, let $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where $\varphi_1 \in \mathsf{BP}$, $\varphi_2 \in \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$, and $\varphi_3 \in Æ_1 \mathtt{i}_1, \ldots, Æ_z \mathtt{i}_z \centerdot \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. By Definition 11:

$$\sigma \models \varphi \Leftrightarrow \sigma \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \models \varphi_3 \Leftrightarrow$$

$$\triangleright \text{ by Lemma 17 and Lemma 7}$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonleft \mathsf{i} \models \varphi_3 \Rightarrow$$

$$\triangleright \text{ playing around with } \wedge, \forall, \text{ and } \exists$$

$$\exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \downharpoonright \mathsf{i} \models \varphi_3 \Rightarrow$$

$$\triangleright \text{ Again by Definition 11}$$

$$\exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Rightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models \varphi$$

This completes the proof. □

The next lemma states that a store σ satisfies an universal state formula φ iff every projection of σ satisfies φ.

Lemma 17. *Let* $\varphi \in \mathsf{USF}$ *and* σ *be any store. Then:*

$$\sigma \models \varphi \Leftrightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models \varphi$$

Proof. From Figure 4.6, we consider three sub-cases:

Case 1. $\varphi \in \mathsf{BP}$. Our result follows from Lemma 6.

Case 2. $\varphi = \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi$ and $\pi \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. Our result follows from Lemma 7.

Case 3. $\varphi = \pi \wedge \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$ such that $\pi \in \mathsf{BP}$, and $\pi' \in \mathsf{PP}(\mathtt{i}_1, \ldots, \mathtt{i}_z)$. In this case, by Definition 11:

$$\sigma \models \varphi \Leftrightarrow \sigma \models \pi \bigwedge \sigma \models \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ by Lemma 6 and Lemma 7}$$

$$\Leftrightarrow \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonright \mathsf{j} \models \pi \bigwedge \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonright \mathsf{j} \models \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ since } \forall \text{ distributes over } \wedge$$

$$\Leftrightarrow \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonright \mathsf{j} \models \pi \wedge \sigma \downharpoonright \mathsf{j} \models \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$$

$$\triangleright \text{ Again by Definition 11}$$

$$\Leftrightarrow \forall \mathsf{j} \in \otimes(\sigma^n_{1,z}) \centerdot \sigma \downharpoonleft \mathsf{j} \models \pi \wedge \forall \mathtt{i}_1 \ldots \forall \mathtt{i}_z \centerdot \pi'$$

By alpha renaming of j to i, we get our desired result. □

The next lemma states that if a store σ is transformed to σ' by executing an instantiated guarded command $\mathsf{gc}(\mathsf{k})$, then every projection of σ is transformed to the corresponding projection of σ' by executing $\mathsf{gc}(\mathsf{k})$.

Lemma 18 (Instantiated Command Projection)**.** *For any stores* σ, σ' *and instantiated guarded command* $\mathsf{gc}(\mathsf{k})$*:*

$$\{\sigma\}\ \mathsf{gc}(\mathsf{k})\ \{\sigma'\} \Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}(1^d)\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

Proof. The proof proceeds as follows. By Figure 6.3:

$$\{\sigma\}\ \mathsf{gc}(\mathsf{k})\ \{\sigma'\} \Rightarrow \sigma^n = \sigma'^n = \mathsf{k} \wedge \{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By Lemma 12

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot (\sigma \downharpoonleft \mathsf{i})^n = (\sigma' \downharpoonleft \mathsf{i})^n = 1^d \wedge \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

$\triangleright$ Again by Figure 6.3

$$\Rightarrow \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \{\sigma \downharpoonleft \mathsf{i}\}\ \mathsf{gc}(1^d)\ \{\sigma' \downharpoonleft \mathsf{i}\}$$

This completes the proof. □

The last lemma relating store projection and formulas states that if every projection of a store σ satisfies a generic state formula φ, then σ satisfies φ.

Lemma 19. *Let* $\varphi \in \mathsf{GSF}$ *and* σ *be any store. Then:*

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models \varphi \Rightarrow \sigma \models \varphi$$

Proof. By considering the structure of φ. We consider three cases:

Case 1. $\varphi \in \mathsf{USF}$. Follows directly from Lemma 17.

Case 2. $\varphi \in \mathsf{ESF}$. Follows directly from Lemma 9.

Case 3. $\varphi \in \mathsf{USF} \wedge \mathsf{ESF}$. Without loss of generality, let $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where $\varphi_1 \in \mathsf{BP}$, $\varphi_2 \in \forall \mathtt{i}_1 \dots \forall \mathtt{i}_z \centerdot \mathsf{PP}(\mathtt{i}_1 \dots \mathtt{i}_z)$, and $\varphi_3 \in Æ\mathtt{i}_1 \dots \forall \mathtt{i}_z \centerdot \mathsf{PP}(\mathtt{i}_1 \dots \mathtt{i}_z)$. by Definition 11:

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models \varphi \Leftrightarrow$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Leftrightarrow$$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \sigma \downharpoonright \mathsf{i} \models \varphi_3 \Leftrightarrow$$

$\triangleright$ playing around with $\wedge$ and $\forall$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models \varphi_3 \Rightarrow$$

$\triangleright$ weakening $\forall$ to $\exists$

$$\forall \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models (\varphi_1 \wedge \varphi_2) \bigwedge \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma \downharpoonright \mathsf{i} \models \varphi_3 \Rightarrow$$

$\triangleright$ by Lemma 17 and Lemma 7

$$\sigma \models (\varphi_1 \wedge \varphi_2) \wedge \sigma \models \varphi_3 \Rightarrow \sigma \models (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \Rightarrow \sigma \models \varphi$$

This completes the proof. □

So far, we used the concept of store projection to show that the effect of executing a

PGCL^{++} program carries over from larger stores to unit stores (i.e., stores obtained via projection). To prove our small model theorems, we also need to show that the effect of executing a *PGCL*$^{++}$ program propagate in the opposite direction, i.e., from unit stores to larger stores. To this end, we first present a notion, called store generalization, that relates unit stores to those of arbitrarily large size.

Definition 21 (Store Generalization)**.** *Let* $\sigma = (\sigma^B, \sigma^W, 1^d, \sigma^P)$ *be any store. For any* $\mathsf{k} \in \mathbb{N}^d$ *we write* $\sigma \upharpoonleft \mathsf{k}$ *to mean the store satisfying the following condition:*

$$(\sigma \upharpoonleft \mathsf{k})^n = \mathsf{k} \wedge \forall \mathsf{i} \in \otimes(\mathsf{k}) \centerdot (\sigma \upharpoonleft \mathsf{k}) \downharpoonleft \mathsf{i} = \sigma$$

Intuitively, $\sigma \upharpoonleft \mathsf{k}$ is constructed by duplicating the only rows of σ^P at each depth, and leaving the other components of σ unchanged.

We now present a lemma related to store generalization, which is needed for the proof of Theorem 14. The lemma states that if a store σ is transformed to σ' by executing an instantiated guarded command $\mathsf{gc}(\mathsf{k})$, then every generalization of σ is transformed to the corresponding generalization of σ' by executing $\mathsf{gc}(\mathsf{k})$.

Lemma 20 (Instantiated Command Generalization)**.** *For any stores* σ, σ' *and instantiated guarded command* $\mathsf{gc}(1^d)$*:*

$$\{\sigma\}\ \mathsf{gc}(1^d)\ \{\sigma'\} \Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonleft \mathsf{k}\}\ \mathsf{gc}(\mathsf{k})\ \{\sigma' \upharpoonleft \mathsf{k}\}$$

Proof. The proof proceeds as follows. By Figure 6.3:

$$\{\sigma\}\ \mathsf{gc}(1^d)\ \{\sigma'\} \Rightarrow \sigma^n = \sigma'^n = 1^d \wedge \{\sigma\}\ \mathsf{gc}\ \{\sigma'\} \Rightarrow$$

$\triangleright$ By Definition 21 and Lemma 15

$$\forall \mathsf{k} \in \mathbb{N}^d \centerdot (\sigma \upharpoonright \mathsf{k})^n = (\sigma' \upharpoonright \mathsf{k})^n = \mathsf{k} \bigwedge \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}\ \{\sigma' \upharpoonright \mathsf{k}\}$$

$\triangleright$ Again by Figure 6.3

$$\Rightarrow \forall \mathsf{k} \in \mathbb{N}^d \centerdot \{\sigma \upharpoonright \mathsf{k}\}\ \mathsf{gc}(\mathsf{k})\ \{\sigma' \upharpoonright \mathsf{k}\}$$

This completes the proof. □

C.1.6 Proof of Theorem 14

Theorem 1 (Small Model Safety 1)**.** *A Kripke structure $M(\mathsf{gc}(\mathsf{k}), Init)$ exhibits a formula φ iff there is a reachable state σ of $M(\mathsf{gc}(\mathsf{k}), Init)$ such that $\sigma \models \varphi$. Let $\mathsf{gc}(\mathsf{k})$ be any instantiated guarded command. Let $\varphi \in \mathsf{GSF}$ be any generic state formula, and $Init \in \mathsf{USF}$ be any universal state formula. Then $M(\mathsf{gc}(\mathsf{k}), Init)$ exhibits φ iff $M(\mathsf{gc}(1^d), Init)$ exhibits φ.*

Proof. For the forward implication, let $\sigma_1, \sigma_2, \ldots, \sigma_w$ be a sequence of states of $M(\mathsf{gc}(\mathsf{k}), Init)$ such that:

$$\sigma_1 \models Init \bigwedge \sigma_w \models \varphi \bigwedge \forall i \in [1, w-1] \centerdot \{\sigma_i\}\ \mathsf{gc}(\mathsf{k})\ \{\sigma_{i+1}\}$$

Since $\varphi \in \mathsf{GSF}$, by Lemma 16 we know that:

$$\exists \mathsf{j} \in \otimes(\sigma_w^n) \centerdot \sigma_w \downharpoonright \mathsf{j} \models \varphi$$

Let j_0 be such a j. By Lemma 17, since $Init \in \mathsf{USF}$:

$$\sigma_1 \downharpoonright \mathsf{j}_0 \models Init$$

By Lemma 18, we know that:

$$\forall i \in [1, w-1] \centerdot \{\sigma_i \downharpoonleft \mathsf{j}_0\}\ \mathsf{gc}(\mathtt{1}^d)\ \{\sigma_{i+1} \downharpoonleft \mathsf{j}_0\}$$

Therefore, $\sigma_w \downharpoonleft \mathsf{j}_0$ is reachable in $M(\mathsf{gc}(\mathtt{1}^d), \mathit{Init})$ and $\sigma_w \downharpoonleft \mathsf{j}_0 \models \varphi$. Hence, $M(\mathsf{gc}(\mathtt{1}^d), \mathit{Init})$ exhibits φ. For the reverse implication, let $\sigma_1, \sigma_2, \ldots, \sigma_w$ be a sequence of states of $M(\mathsf{gc}(\mathtt{1}), \mathit{Init})$ such that:

$$\sigma_1 \models \mathit{Init} \bigwedge \sigma_w \models \varphi \bigwedge \forall i \in [1, w-1] \centerdot \{\sigma_i\}\ \mathsf{gc}(\mathtt{1}^d)\ \{\sigma_{i+1}\}$$

For each $i \in [1, w]$, let $\widehat{\sigma_i} = \sigma_i \upharpoonleft \mathsf{k}$. Therefore, since $\mathit{Init} \in \mathsf{USF}$, by Lemma 17, we know:

$$\forall \mathsf{j} \in \otimes(\mathsf{k}) \centerdot \widehat{\sigma_1} \downharpoonleft \mathsf{j} \models \mathit{Init} \Rightarrow \widehat{\sigma_1} \models \mathit{Init}$$

Also, since $\varphi \in \mathsf{GSF}$, by Lemma 19 we know that:

$$\forall \mathsf{j} \in \otimes(\mathsf{k}) \centerdot \widehat{\sigma_n} \downharpoonleft \mathsf{j} \models \varphi \Rightarrow \widehat{\sigma_w} \models \varphi$$

Finally, by Lemma 20, we know that:

$$\forall i \in [1, w-1] \centerdot \{\widehat{\sigma_i}\}\ \mathsf{gc}(\mathsf{k})\ \{\widehat{\sigma_{i+1}}\}$$

Therefore, $\widehat{\sigma_w}$ is reachable in $M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$ and $\widehat{\sigma_w} \models \varphi$. Hence, $M(\mathsf{gc}(\mathsf{k}), \mathit{Init})$ exhibits φ. This completes the proof.

$\square$

C.1.7 Proof of Theorem 15

Theorem 2 (Small Model Simulation)**.** *Let* $\mathsf{gc(k)}$ *be any instantiated guarded command. Let Init* $\in$ GSF *be any generic state formula. Then* $M(\mathsf{gc(k)}, \mathit{Init}) \preceq M(\mathsf{gc(1}^d\mathsf{)}, \mathit{Init})$ *and* $M(\mathsf{gc(1}^d\mathsf{)}, \mathit{Init}) \preceq M(\mathsf{gc(k)}, \mathit{Init})$.

Proof. Recall the conditions **C1–C3** in Definition 22 for simulation. For the first simulation, we propose the following relation $\mathcal{H}$ and show that it is a simulation relation:

$$(\sigma, \sigma') \in \mathcal{H} \Leftrightarrow \exists \mathsf{i} \in \otimes(\sigma^n_{1,d}) \centerdot \sigma' = \sigma \downharpoonleft \mathsf{i}$$

C1 holds because our atomic propositions are USF formulas, and Lemma 17; **C2** holds because *Init* $\in$ GSF and Lemma 16; **C3** holds by Definition 4 and Lemma 18. For the second simulation, we propose the following relation $\mathcal{H}$ and show that it is a simulation relation:

$$(\sigma, \sigma') \in \mathcal{H} \Leftrightarrow \sigma' = \sigma \upharpoonleft \sigma'^n$$

Again, **C1** holds because our atomic propositions are USF formulas, Definition 21, and Lemma 17; **C2** holds because *Init* $\in$ GSF, Definition 21, and Lemma 19; **C3** holds by Definition 4 and Lemma 20. This completes the proof. □

C.1.8 Proof of Corollary 16

We begin with the formal definition of simulation [20].

Definition 22. *Let* $M_1 = (\mathcal{S}_1, I_1, \mathcal{T}_1, \mathcal{L}_1)$ *and* $M_2 = (\mathcal{S}_2, I_2, \mathcal{T}_2, \mathcal{L}_2)$ *be two Kripke structures over sets of atomic propositions* AP_1 *and* AP_2 *such that* $\mathsf{AP}_2 \subseteq \mathsf{AP}_1$. *Then* M_1 *is simulated by* M_2*, denoted by* $M_1 \preceq M_2$*, iff there exists a relation* $H \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ *such that the following three conditions hold:*

(C1) $\forall s_1 \in \mathcal{S}_1 \centerdot \forall s_2 \in \mathcal{S}_2 \centerdot (s_1,s_2) \in H \Rightarrow \mathcal{L}_1(s_1) \cap \mathsf{AP}_2 = \mathcal{L}_2(s_2)$

(C2) $\forall s_1 \in I_1 \centerdot \exists s_2 \in I_2 \centerdot (s_1,s_2) \in H$

(C3) $\forall s_1, s_1' \in \mathcal{S}_1 \centerdot \forall s_2 \in \mathcal{S}_2 \centerdot (s_1,s_2) \in H \wedge (s_1,s_1') \in \mathcal{T}_1 \Rightarrow$
$\exists s_2' \in \mathcal{S}_2 \centerdot (s_2,s_2') \in \mathcal{T}_2 \wedge (s_1',s_2') \in H$

Next, we formalize our claim that PTSL formulas are preserved by simulation.

Fact 4. *Let M_1 and M_2 be two Kripke structures over propositions* AP_1 *and* AP_2 *such that $M_1 \preceq M_2$. Hence, by Definition 6,* $\mathsf{AP}_2 \subseteq \mathsf{AP}_1$. *Let φ be any* PTSL *formula over* AP_2. *Therefore, φ is also a* PTSL *formula over* AP_1. *Then $M_2 \models \varphi \Rightarrow M_1 \models \varphi$.*

We are now ready to prove Corollary 16.

Corollary 3 (Small Model Safety 2)**.** *Let* $\mathsf{gc(k)}$ *be any instantiated guarded command. Let $\varphi \in$* USF *be any universal state formula, and Init $\in$* GSF *be any generic state formula. Then $M(\mathsf{gc(k)},\mathit{Init})$ exhibits φ iff $M(\mathsf{gc(1}^d\mathsf{)},\mathit{Init})$ exhibits φ.*

Proof. Follows from: (i) the observation that exhibition of a USF formula ϕ is expressible in PTSL as the TLF formula $\mathbf{F}\ \phi$, (ii) Theorem 15, and (iii) Fact 4. $\square$

Bibliography

[1] E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. Paul. Verifying shadow page table algorithms. In *Proceedings of FMCAD*, 2010.

[2] Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. Automated verification of a small hypervisor. In *Proceedings of VSTTE*, 2010.

[3] Anonymous. Xbox 360 hypervisor privilege escalation vulnerability. http://www.securityfocus.com/archive/1/461489, Feb. 2007.

[4] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[5] ARM Holdings. ARM1176JZF-S technical reference manual. Revision r0p7, 2009.

[6] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions. In *Proceedings of CAV*, 2001.

[7] Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of

Lecture Notes in Computer Science, pages 103–122, Toronto, Canada, May 19–20, 2001. New York, NY, May 2001. Springer-Verlag.

[8] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proc. of PASTE*, pages 97–103, 2001.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of SOSP*, 2003.

[10] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *Proceedings of 17th USENIX Security Symposium*, 2008.

[11] G. Barthe, G. Betarte, J. D. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *Proc. of FM*, 2011.

[12] C. Baumann, H. Blasum, T. Bormer, and S Tverdyshev. Proving memory separation in a microkernel by code level verification. In *Proc. of AMICS*, 2011.

[13] D.M. Berry. Towards a formal basis for the formal development method and the ina jo specification language. *IEEE Transactions on Software Engineering*, 13:184–201, 1987.

[14] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[15] D. F. C. Brewer and M.J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1989.

[16] CBMC website, Accessed August, 2011. `http://www.cprover.org/cbmc|`.

[17] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

[18] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[19] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer Berlin / Heidelberg, 2004.

[20] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.

[21] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kirli Kaynar. A logic of secure systems and its application to trusted computing. In *IEEE Symposium on Security and Privacy*, pages 221–236, 2009.

[22] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded model checking with qbf. In *Proc. of SAT*, pages 408–414, 2005.

[23] DIMACS. Satisability Suggested Format. Accessed from ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/, May 1993.

[24] Nancy A. Durgin, Patrick Lincoln, and John C. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[25] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Proc. of CADE*, 2000.

[26] E. Allen Emerson and Vineet Kahlon. Model checking large-scale and parameterized resource allocation systems. In *Proc. of TACAS*, 2002.

[27] E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Proc. of CHARME*, 2003.

[28] E. Allen Emerson and Vineet Kahlon. Model checking guarded protocols. In *Proceedings of LICS*, 2003.

[29] E. Allen Emerson and Vineet Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Proc. of TACAS*, 2003.

[30] E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *Proceedings of CAV*, 1996.

[31] E. Allen Emerson and Kedar S. Namjoshi. Verification of parameterized bus arbitration protocol. In *Proceedings of CAV*, 1998.

[32] M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counting implementations. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '09)*, 2009.

[33] Yi Fang, Nir Piterman, Amir Pnueli, and Lenore Zuck. Liveness with invisible ranking. In *Proceedings of VMCAI*, 2003.

[34] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. Attacking, repairing, and verifying SecVisor: A retrospective on the security of a hypervisor. Technical Report CMU-CyLab-08-008, Carnegie Mellon University, 2008.

[35] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the Symposium on Operating System Principals (SOSP)*, 2003.

[36] Deepak Garg, Jason Franklin, Dilsun Kirli Kaynar, and Anupam Datta. Compositional system security with interface-confined adversaries. *Electr. Notes Theor. Comput. Sci.*, 265:49–71, 2010.

[37] S. L. Gerhart. An overview of Affirm: A specification and verification system. In *IFIP*, 1980.

[38] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[39] Virgil D. Gligor. Analysis of the hardware verification of the Honeywell SCOMP. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1985.

[40] D. I. Good and R. M. Cohen. Verifiable communications processing in gypsy. In *17th COMPCON*, 1978.

[41] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in security-enhanced linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[42] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, 2002.

[43] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8), August 1976.

[44] Bret A. Hartman. A gypsy-based kernel. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1984.

[45] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John D. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of ACM CCS*, 2006.

[46] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '02)*, volume 37(1) of *SIGPLAN Notices*, pages 58–70, Portland, OR, January 16–18, 2002. New York, NY, January 2002. Association for Computing Machinery.

[47] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[48] Intel Corporation. Intel 64 and IA-32 Intel architecture software developer's manual. Intel Publication nos. 253665–253669, 2008.

[49] N. Kidd, T. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, 2009.

[50] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an os kernel. In *Proceedings of SOSP*, 2009.

[51] William Klieber, Samir Sapra, Sicun Gao, and Edmund M. Clarke. A non-prenex, non-clausal qbf solver with game-state learning. In *Proc. of 13th International Con-*

ference on Theory and Applications of Satisfiability Testing (SAT'10), pages 128–142, 2010.

[52] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Loopfrog: A static analyzer for ansi-c programs. In *Proc. of ASE*, pages 668–670, 2009.

[53] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[54] Ranko Lazić, Tom Newcomb, and Bill Roscoe. On model checking data-independent systems with arrays with whole-array operations. *LNCS*, 3525:275–291, July 2004.

[55] R.S. Lazić, T.C. Newcomb, and A.W. Roscoe. On model checking data-independent systems with arrays without reset. *Theory and Practice of Logic Programming*, 4(5&6), 2004.

[56] David Lie, John Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[57] Gavin Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(1), 1999.

[58] Jonathan Millen. A necessarily parallel attack. In *Proceedings of the Workshop on Formal Methods and Security Protocols (FMSP)*, 1999.

[59] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In *Proceedings of the Seventh USENIX Security Symposium*, pages 201–216, 1998.

[60] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated Analysis of Cryptographic Protocols Using Murφ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[61] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical report, SRI International, 1980.

[62] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *Proc. of TACAS*, 2001.

[63] QuBE website, Accessed August, 2011. `www.star.dist.unige.it/ qube|`.

[64] IBM Research. The research hypervisor - a multi-platform, multi-purpose research hypervisor. http://www.research.ibm.com/hypervisor, 2012.

[65] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal Computer Security*, 7(2-3):147–190, 1999.

[66] John Rushby. The design and verification of secure systems. In *Proceedings of SOSP*, 1981. (ACM *OS Review*, Vol. 15, No. 5).

[67] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype:secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research Report, 2005.

[68] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-Based security architecture for the Xen open-source hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[69] W. L. Schiller. Design and abstract specification of a multics security kernel. Technical report, MITRE Corp, Bedford MA, 1977.

[70] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[71] Jonathan S. Shapiro and Sam Weber. Verifying the eros confinement mechanism. In *Proceedings of IEEE S&P*, 2000.

[72] sKizzo website, Accessed August, 2011. `http://skizzo.info|`.

[73] I. Suzuki. Proving properties of a ring of finite state machines. *Information Processing Letters*, 28:213–213, 1988.

[74] Core Security Technologies. Virtual pc hypervisor memory protection vulnerability. http://www.coresecurity.com/content/virtual-pc-2007-hypervisor-memory-protection-bug, March 2010.

[75] Enriquillo Valdez, Reiner Sailer, and Ronald Perez. Retrofitting the ibm power hypervisor to support mandatory access control. In *Proceedings of the Twenty-Third Annual Computer Security Applications Conference (ACSAC)*, 2007.

[76] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

[77] Jeannette M. Wing and Mandana Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28:273–299, 1997.

[78] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of POPL*, 1986.

[79] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 273–288, 2004.