# Scalable Parametric Verification of Secure Systems: How to Verify Reference Monitors without Worrying about Data Structure Size

Jason Franklin, Sagar Chaki, Anupam Datta
Carnegie Mellon University
Pittsburgh, PA

Arvind Seshadri
IBM Research
Yorktown Heights, NY

## Abstract

*The security of systems such as operating systems, hypervisors, and web browsers depend critically on reference monitors to correctly enforce their desired security policy in the presence of adversaries. Recent progress in developing reference monitors with small code size and narrow interfaces has made automated formal verification of reference monitors a more tractable goal. However, a significant remaining factor for the complexity of automated verification is the size of the data structures (e.g., access control matrices) over which the programs operate. This paper develops a parametric verification technique that scales 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. The central technical results of the paper are a set of small model theorems. These 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 two hypervisors, SecVisor and the sHype mandatory-access-control extension to Xen. 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.*

## 1. Introduction

A central focus of system security is on the design and implementation of reference monitors. A *reference monitor* observes execution of a target system and prevents actions that violate the relevant security policy [1]. For example, a virtual machine monitor might mediate access to memory and enforce a separation policy, i.e. it may prevent one virtual machine from accessing memory regions allocated to a different virtual machine. There are numerous other examples of reference monitors in practical systems such as operating systems, hypervisors, and web browsers. The security of these systems depend critically on whether the reference monitor correctly enforces the desired security policy in the presence of adversaries. Therefore, providing assurance in the security of reference monitors is of critical importance.

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 paper 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 paper are a set of *small model theorems* that state that for any system $M$ expressible in the language, any security property $\varphi$ 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.

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. That 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 [40] and the sHype [39] mandatory-access-control extension to Xen [3]—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 [5], a well-known access control policy used in commercial settings. These applications are presented in Section 4. In addition, we further demonstrate the expressivity of the programming language by showing in Section 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 work builds on a line of work on data independence, introduced in an influential paper by Wolper [45]. He considered programs whose control flow behavior was completely independent of the data over which they operated. The motivating example for him was the alternating bit protocol. Subsequent work on parametric verification, which we survey in the related work section (Section 6), has relaxed this strong independence assumption to permit the control flow of the program to depend in limited ways on the data. The closest line of work to ours is by Emerson and Kahlon [9], and Lazic et al. on verifying correctness of cache coherence protocols [28], [29]. 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. Conclusions and directions for future work appear in Section 7.

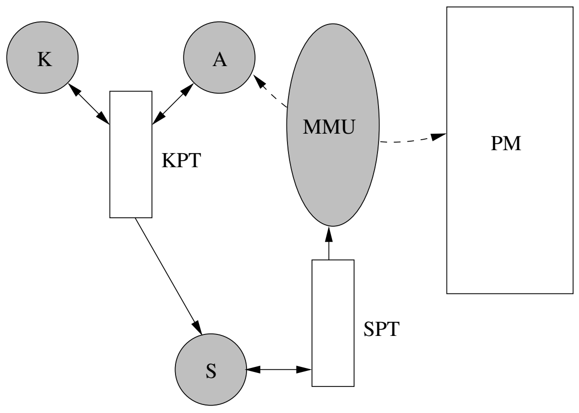


Fig. 1. Functional overview of SecVisor; K = GUEST kernel; A = adversary; S = SecVisor; PM = physical memory; KPT and SPT = kernel and shadow page tables; shaded shape = active element; unshaded shape = data structure; direction of arrows indicate kind of access (read or write) of data structure by element.

## 2. Motivating Example: SecVisor

We use SecVisor [40], a security hypervisor, as our motivating example. We informally discuss the design of SecVisor below, and in subsequent sections 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 1 shows a functional overview of SecVisor. 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 a minimal 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 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. 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 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 $\varphi$ holds on a PGCL program *Prog* for arbitrarily large instantiations of P if and only if $\varphi$ 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.

### 3.1. 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 $\mathtt{q} \in \mathtt{K}$. 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, 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 P, and n be the variable used to store the number of rows of P.

The syntax of PGCL is shown in Figure 2. Expressions in PGCL include natural numbers, Boolean variables, a parameterized array $\mathtt{P}_{\mathtt{n},\mathtt{q}}$, a variable i for indexing into $\mathtt{P}_{\mathtt{n},\mathtt{q}}$, a variable n representing the size of $\mathtt{P}_{\mathtt{n},\mathtt{q}}$, propositional expressions over Boolean variables and elements of $\mathtt{P}_{\mathtt{n},\mathtt{q}}$. For notational simplicity, we often write P to mean $\mathtt{P}_{\mathtt{n},\mathtt{q}}$. The commands in PGCL include guarded commands that update Boolean variables and elements of $\mathtt{P}_{\mathtt{n},\mathtt{q}}$, 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.

#### 3.1.1. SecVisor in PGCL.

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

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 overall SecVisor program is a parallel composition of four guarded commands, as follows:

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

|  |  |  |  |  |
|---|---|---|---|---|
| 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 | $\widehat{\mathtt{E}}$ | $::=$ | $\mathtt{E} \mid \mathtt{P_{n,q}[i][K]} \mid \widehat{\mathtt{E}} \vee \widehat{\mathtt{E}} \mid \widehat{\mathtt{E}} \wedge \widehat{\mathtt{E}} \mid \neg \widehat{\mathtt{E}}$ |  |
| Instantiated Guarded Commands | $\mathtt{G}$ | $::=$ | $\mathtt{GC(K)}$ |  |
| Guarded Commands | $\mathtt{GC}$ | $::=$ | $\mathtt{E} \ ? \ \mathtt{C}$ | Simple guarded command |
|  |  | $\mid$ | $\mathtt{GC} \parallel \mathtt{GC}$ | Parallel composition |
| Commands | $\mathtt{C}$ | $::=$ | $\mathtt{B} := \mathtt{E}$ | Assignment |
|  |  | $\mid$ | $\mathtt{for\ i\ :\ P_{n,q}\ do}\ \widehat{\mathtt{E}} \ ? \ \widehat{\mathtt{C}}$ | Parametric |
|  |  | $\mid$ | $\mathtt{C;C}$ | Sequencing |
| Parameterized Commands | $\widehat{\mathtt{C}}$ | $::=$ | $\mathtt{P_{n,q}[i][K]} := \widehat{\mathtt{E}}$ | Parameterized array assignment |
|  |  | $\mid$ | $\widehat{\mathtt{C}};\widehat{\mathtt{C}}$ | Sequencing |

Fig. 2. PGCL Syntax

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 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(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(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(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(d) where the expression $*$ non-deterministically evaluates to either **true** or **false**.

## 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 $\sigma$ 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 $\sigma^P$ as a function of type $[1,\sigma^n] \to [1,\lceil \mathtt{q} \rceil] \to \mathbb{B}$. In the rest of this paper, we omit the relevant superscript of $\sigma$ 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 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 $\sigma^B$, setting $\sigma^n$ to 1, and projecting away all but the $i$-th row from $\sigma^P$. Note that since projection retains $\sigma^B$, it does not affect the evaluation of expressions that do not refer to elements of $\mathtt{P_{n,q}}$.

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 $\mathtt{X}$, 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 $\mathtt{e}$.

```
Kernel_Entry ≡
  ¬kernelmode ? kernelmode := ⊤;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = UM ?
      $P_{n,q}$[i][SPTRW] := ⊤; $P_{n,q}$[i][SPTX] := ⊥;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = KC ?
      $P_{n,q}$[i][SPTRW] := ⊥; $P_{n,q}$[i][SPTX] := ⊤;
  for i : $P_{n,q}$ do
    $P_{n,q}$[i][SPTPA] = KD ?
      $P_{n,q}$[i][SPTRW] := ⊤; $P_{n,q}$[i][SPTX] := ⊥
```

(a)

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

(b)

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

(c)

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

(d)

Fig. 3. PGCL program for SecVisor.

**Store Transformation.** For any PGCL command c and stores $\sigma$ and $\sigma'$, we write $\{\sigma\}$ c $\{\sigma'\}$ to mean that $\sigma$ is transformed to $\sigma'$ by the execution of c. The rules defining $\{\sigma\}$ c $\{\sigma'\}$, via induction on the structure of c, are shown in Figure 5. Most of the definitions are straightforward. For example, the "GC" rule states that $\sigma$ is transformed to $\sigma'$ by executing the guarded command e ? c if: (i) the guard e evaluates to **true** under $\sigma$, and (ii) $\sigma$ is transformed to $\sigma'$ by executing the command c.

The "Unroll" rule states that if c is a for loop, then $\{\sigma\}$ 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.

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

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

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

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

## 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 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 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 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* [7] 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, and vice versa. 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:

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

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:

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

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

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

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

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

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

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

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

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

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

Fig. 4. Rules for expression evaluation. Recall that $\sigma = (\sigma^B, \sigma^n, \sigma^P)$.

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

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

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

$$\frac{\langle \mathtt{e}, \sigma\rangle \rightarrow t \qquad \lceil \mathtt{i} \rceil \leq \sigma^n \qquad \lceil \mathtt{j} \rceil \leq \lceil \mathtt{q} \rceil}{\{\sigma\}\ \mathtt{P_{n,q}[i][j] := e}\ \{\sigma[\sigma^P \mapsto \sigma^P[(\lceil \mathtt{i} \rceil, \lceil \mathtt{j} \rceil) \mapsto t]]\}}\text{Parameterized Array Assign} \qquad \frac{\{\sigma\}\ \mathtt{gc}\ \{\sigma'\} \vee \{\sigma\}\ \mathtt{gc}'\ \{\sigma'\}}{\{\sigma\}\ \mathtt{gc} \parallel \mathtt{gc}'\ \{\sigma'\}}\text{Parallel}$$

Fig. 5. Rules for commands

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

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

**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 $\pi$ by a store $\sigma$ (denoted $\sigma \models \pi$) is defined, by induction on the structure of $\pi$, as follows:*

- $\sigma \models \mathtt{b}$ *iff* $\sigma^B(\mathtt{b}) = \textbf{true}$.
- $\sigma \models \mathtt{P_{n,q}[k][l]}$ *iff* $\lceil \mathtt{k} \rceil \leq \sigma^n$ *and* $\sigma^P(\lceil \mathtt{k} \rceil, \lceil \mathtt{l} \rceil) = \textbf{true}$.
- $\sigma \models \neg\pi$ *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* $\sigma \models \pi_2$.
- $\sigma \models \forall \mathtt{i} \centerdot \pi$ *iff* $\forall i \in [1, \sigma^n] \centerdot \sigma \downharpoonright i \models \pi[\mathtt{i} \mapsto 1]$.
- $\sigma \models \exists \mathtt{i} \centerdot \pi$ *iff* $\exists i \in [1, \sigma^n] \centerdot \sigma \downharpoonright i \models \pi[\mathtt{i} \mapsto 1]$.

The definition of satisfaction of Boolean formulas and the logical operators are standard. Parametric formulas, denoted $\mathtt{P_{n,q}[k][l]}$, 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, $\forall \mathtt{i} \centerdot \pi$, is satisfied by $\sigma$ if and only if all projections of $\sigma$ satisfy $\pi[\mathtt{i} \mapsto 1]$. Finally, an existentially quantified formula, $\exists \mathtt{i} \centerdot \pi$, is satisfied by $\sigma$ if and only if there exists a projection of $\sigma$ 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 $\mathtt{gc}$ be any PGCL guarded command and $\mathtt{k} \in \mathtt{K}$ be any numeral. We denote the set of stores $\sigma$ such that $\sigma^n = \lceil \mathtt{k} \rceil$, as $Store(\mathtt{gc(k)})$. Note that $Store(\mathtt{gc(k)})$ is finite. Let $Init$ be any formula and AP $=$ USF be the set of atomic propositions. Intuitively, a Kripke structure $M(\mathtt{gc(k)}, Init)$ over AP is induced by executing $\mathtt{gc(k)}$ starting from any store $\sigma \in Store(\mathtt{gc(k)})$ that satisfies $Init$.

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

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

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

$$\begin{array}{llcl}
\text{Basic Propositions} & \mathsf{BP} & ::= & \mathtt{b}\ ,\ \mathtt{b} \in B \\
 & & \mid & \neg\mathsf{BP} \\
 & & \mid & \mathsf{BP} \wedge \mathsf{BP}
\end{array}
\qquad
\begin{array}{llcl}
\text{Parametric Propositions} & \mathsf{PP}(\mathtt{i}) & ::= & \{\mathtt{P}_{\mathtt{n},\mathtt{q}}[\mathtt{i}][\mathtt{l}] \mid \lceil \mathtt{l} \rceil \le \lceil \mathtt{q} \rceil\} \\
 & & \mid & \neg\mathsf{PP}(\mathtt{i}) \\
 & & \mid & \mathsf{PP}(\mathtt{i}) \wedge \mathsf{PP}(\mathtt{i})
\end{array}$$

$$\begin{array}{llcl}
\text{Universal State Formulas} & \mathsf{USF} & ::= & \mathsf{BP} \\
 & & \mid & \forall \mathtt{i}\,.\,\mathsf{PP}(\mathtt{i}) \\
 & & \mid & \mathsf{BP} \wedge \forall \mathtt{i}\,.\,\mathsf{PP}(\mathtt{i})
\end{array}
\qquad
\begin{array}{llcl}
\text{Existential State Formulas} & \mathsf{ESF} & ::= & \mathsf{BP} \\
 & & \mid & \exists \mathtt{i}\,.\,\mathsf{PP}(\mathtt{i}) \\
 & & \mid & \mathsf{BP} \wedge \exists \mathtt{i}\,.\,\mathsf{PP}(\mathtt{i})
\end{array}$$

$$\begin{array}{llcl}
\text{Generic State Formulas} & \mathsf{GSF} & ::= & \mathsf{USF} \\
 & & \mid & \mathsf{ESF} \\
 & & \mid & \mathsf{USF} \wedge \mathsf{ESF}
\end{array}
\qquad
\begin{array}{llcll}
\mathsf{PTSL}\text{ Path Formulas} & \mathsf{TLPF} & ::= & \mathsf{TLF} & \text{“state formula”} \\
 & & \mid & \mathsf{TLF} \wedge \mathsf{TLF} & \text{“conjunction”} \\
 & & \mid & \mathsf{TLF} \vee \mathsf{TLF} & \text{“disjunction”} \\
 & & \mid & \mathbf{X}\ \mathsf{TLF} & \text{“in the next state”} \\
 & & \mid & \mathsf{TLF}\ \mathbf{U}\ \mathsf{TLF} & \text{“until”}
\end{array}$$

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

Fig. 6. Syntax of PTSL

If $\phi$ is a PTSL formula, then $M,\sigma \models \phi$ means that $\phi$ holds at state $\sigma$ in the Kripke structure $M$. We use a standard inductive definition of the relation $\models$ [7]. Informally, an atomic proposition $\pi$ holds at $\sigma$ iff $\sigma \models \pi$; $\mathbf{A}\ \phi$ holds at $\sigma$ if $\phi$ holds on all possible (infinite) paths starting from $\sigma$. TLPF formulas hold on paths. Informally, a TLF formula $\phi$ holds on a path $\Pi$ iff it holds at the first state of $\Pi$; $\mathbf{X}\ \phi$ holds on a path $\Pi$ iff $\phi$ holds on the suffix of $\Pi$ starting at second state of $\Pi$; $\phi_1\ \mathbf{U}\ \phi_2$ holds on $\Pi$ if $\phi_1$ holds on suffixes of $\Pi$ until $\phi_2$ begins to hold. The definitions for $\neg$, $\wedge$ and $\vee$ are standard.

## 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 3.5.

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

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

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

**Example 1.** *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:*

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

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

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

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

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

*where $\mathbf{G}\phi$ is a path formula meaning that $\phi$ holds at all states of a path, and is a shorthand for ($\phi$ $\mathbf{U}$ **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 [7], 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 $\mathsf{AP}_1$ and $\mathsf{AP}_2$ 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)** $\forall s_1 \in \mathcal{S}_1 \,.\, \forall s_2 \in \mathcal{S}_2 \,.\, (s_1,s_2) \in H \Rightarrow \mathcal{L}_1(s_1) \cap \mathsf{AP}_2 = \mathcal{L}_2(s_2)$
**(C2)** $\forall s_1 \in I_1 \,.\, \exists s_2 \in I_2 \,.\, (s_1,s_2) \in H$
**(C3)** $\forall s_1, s_1' \in \mathcal{S}_1 \,.\, \forall s_2 \in \mathcal{S}_2 \,.\, (s_1,s_2) \in H \wedge (s_1,s_1') \in \mathcal{T}_1 \Rightarrow$
$\exists s_2' \in \mathcal{S}_2 \,.\, (s_2,s_2') \in \mathcal{T}_2 \wedge (s_1',s_2') \in H$

It is known [7] 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 $\mathsf{AP}_1$ and $\mathsf{AP}_2$ such that $M_1 \preceq M_2$. Hence, by Definition 6, $\mathsf{AP}_2 \subseteq \mathsf{AP}_1$. Let $\varphi$ be any* PTSL *formula over $\mathsf{AP}_2$. Therefore, $\varphi$ is also a* PTSL *formula over $\mathsf{AP}_1$. 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 $M(\texttt{gc(k)}, Init) \preceq M(\texttt{gc(1)}, Init)$ and $M(\texttt{gc(1)}, Init) \preceq M(\texttt{gc(k)}, Init)$.*

The following is a corollary of Theorem 2. Note that this corollary is the dual of Theorem 1 obtained by swapping the types of $\varphi$ 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 $M(\texttt{gc(k)}, Init)$ exhibits $\varphi$ iff $M(\texttt{gc(1)}, Init)$ exhibits $\varphi$.*

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

## 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 full version [17].

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 $\sigma$ satisfies an universal state formula $\varphi$ iff every projection of $\sigma$ satisfies $\varphi$.

**Lemma 4.** *Let $\varphi \in \mathsf{USF}$ and $\sigma$ 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 $\sigma$ satisfies a generic state formula $\varphi$, then some projection of $\sigma$ satisfies $\varphi$.

**Lemma 5.** *Let $\varphi \in \mathsf{GSF}$ and $\sigma$ 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 $\sigma$ is transformed to $\sigma'$ by executing an instantiated guarded command $\texttt{gc(k)}$, then every projection of $\sigma$ is transformed to the corresponding projection of $\sigma'$ by executing $\texttt{gc(k)}$.

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

**Lemma 6** (Instantiated Command Projection)**.** *For any stores $\sigma, \sigma'$ 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 $\sigma$ satisfies a generic state formula $\varphi$, then $\sigma$ satisfies $\varphi$.

**Lemma 7.** *Let $\varphi \in \mathsf{GSF}$ and $\sigma$ 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 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 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 $\sigma^P$, and leaving the other components of $\sigma$ 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 $\sigma$ is transformed to $\sigma'$ by executing an instantiated guarded command $\texttt{gc(k)}$, then every generalization of $\sigma$ is transformed to the corresponding generalization of $\sigma'$ by executing $\texttt{gc(k)}$.

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

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

**3.5.1. 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 $M(\texttt{gc(k)}, Init)$ such that:

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

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 $Init \in \mathsf{USF}$:

$$\sigma_1 \downharpoonleft j_0 \models Init$$

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, $\sigma_n \downharpoonright j_0$ is reachable in $M(\texttt{gc(1)}, \mathit{Init})$ and $\sigma_n \downharpoonright j_0 \models \varphi$. Hence, $M(\texttt{gc(1)}, \mathit{Init})$ exhibits $\varphi$. 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 $\widehat{\sigma_i} = \sigma_i \upharpoonright \lceil \texttt{k} \rceil$. Therefore, since $\mathit{Init} \in$ USF, by Lemma 4, we know:

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

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

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

Finally, by Lemma 8, we know that:

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

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

**3.5.2. 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:

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

**C1** holds because our atomic propositions are USF formulas, and Lemma 4; **C2** holds because $\mathit{Init} \in$ 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:

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

Again, **C1** holds because our atomic propositions are USF formulas, Definition 7, and Lemma 4; **C2** holds because $\mathit{Init} \in$ 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 $\mathit{Init} \in$ 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:

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

Unfortunately, Lemma 5 is too weak to imply that $\mathcal{H}$ satisfies even condition **C1**. Specifically, this is because in the consequent of the implication in Lemma 5 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. [29] 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.

## 4. Applications

We demonstrate our methodology on two hypervisors: SecVisor and the sHype [39] mandatory-access-control extension to Xen [3].

### 4.1. SecVisor

Recall our model of SecVisor from Section 3.1.1 and the expression of SecVisor's security properties as PTSL formulas from Section 3.3.

**4.1.1. Initial Failed Verification and Vulnerabilities.** We used the Mur$\varphi$ model checker to verify $\varphi_{exec}$ and $\varphi_{code}$ on our SecVisor model. Mur$\varphi$ 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 Mur$\varphi$'s counterexample to $\varphi_{exec}$. The counterexample involves the attacker modifying a $\mathtt{P_{n,q}}$ row with $\texttt{SPTPA} = \texttt{KC}$ and $\texttt{SPTX} = \top$. Specifically, the attacker changes the value of `KPTPA` from `KC` to `UM`. The new `KPTPA` value is then copied by `Sync` into `SPTPA`. Since $\texttt{SPTX} = \top$, this results in a violation of $\varphi_{exec}$. The key flaw here, in 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.

To demonstrate that this vulnerability is present in SecVisor's implemenation, 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

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 [40] detailing the vulnerabilities is available.

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 $\varphi_{code}$. 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 $\varphi_{code}$. The key flaw here, in 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.

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.

**4.1.2. 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:

```
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 $\varphi_{exec}$ or $\varphi_{code}$ in the fixed SecVisor program. Note that the initial condition, $\varphi_{init}$, is expressible in USF, and the negations of $\varphi_{exec}$ and $\varphi_{code}$ are expressible in GSF. Therefore, by Theorem 1, we know that the fixed SecVisor satisfies both $\varphi_{exec}$ and $\varphi_{code}$ 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 [18].

**4.1.3. 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.

Initially, we 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 statespace 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 approach handles models of unbounded size.

## 4.2. sHype Security Architecture

Next, we explore the expressiveness of PGCL and PTSL by analyzing the Chinese Wall Policy as implemented in sHype hypervisor security architecture [39]. sHype is a mandatory-access-control-based system implemented in the Xen hypervisor [3], the research hypervisor rHype [36], and the PHYP commercial hypervisor [43].

**4.2.1. Chinese Wall Policy.** Access control policies distinguish between two primary types: principals and objects. The Chinese Wall Policy (CWP) [5] 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)* $WL = \{w_1, \dots, w_L\}$ *is a finite set of L workloads, (ii)* $CWTypes = \{cwt_1, \dots, cwt_T\}$ *is a finite set of T CWTypes, (iii)* $CoIClasses = \{coi_1, \dots, coi_C\}$ *is a finite set of C Conflict of Interest classes, (iv) TypeMap maps WL to CWTypes, and (v) ClassMap maps WL to CoIClasses.*

### 4.2.2. Encoding sHype CWP in PGCL.

Each row of the parameterized array P represents a VM, and each column of P represents a workload. Thus, $\mathtt{P}[i][j] = \top$ 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).

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

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

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

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:

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

### 4.2.3. 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 $\mathtt{hypercall} = \top$. Moreover, for each $w \in WL$, the column $\mathtt{ACCESS}(w)$ 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 : $\mathrm{P_{n,q}}$ do
    $\mathrm{P_{n,q}}[\mathtt{i}][\mathtt{ACCESS}(\mathtt{w_1})] \wedge g(\mathtt{i}, \mathtt{w_1})$ ? $\mathrm{P_{n,q}}[\mathtt{i}][\mathtt{w_1}] := \top$;
        ⋮
    $\mathrm{P_{n,q}}[\mathtt{i}][\mathtt{ACCESS}(\mathtt{w_L})] \wedge g(\mathtt{i}, \mathtt{w_L})$ ? $\mathrm{P_{n,q}}[\mathtt{i}][\mathtt{w_L}] := \top$;
```

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 : $\mathrm{P_{n,q}}$ do
    $\top$ ? $\mathrm{P_{n,q}}[\mathtt{i}][\mathtt{ACCESS}(\mathtt{w_1})] := *$;
        ⋮
    $\top$ ? $\mathrm{P_{n,q}}[\mathtt{i}][\mathtt{ACCESS}(\mathtt{w_L})] := *$
```

Finally, we define the overall sHype system:

```
sHype_CWP $\equiv$ access_ref_monitor $\|$ 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 Mur$\varphi$ 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.

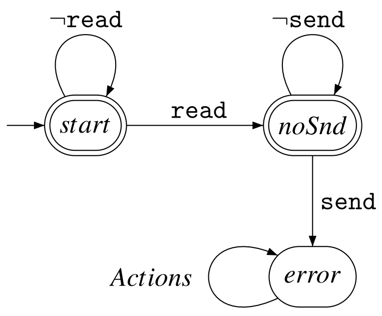
Although no additional verification is necessary to demonstrate the absence of counterexamples, we explore scaling trends and demonstrate the utility of our technique by increasing the number of virtual machines instances. With two virtual machines instances, the verification completed in less than one 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.

## 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 1 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, $\sigma_i$ and $\sigma_i'$, which represent the current and next states of the FSA respectively. We need the $\sigma_i'$ columns to simulate FSA since FSA is non-deterministic, and could end up in multiple possible states after a sequence of actions.
- Each $a_i \in Actions$ 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:

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

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

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

- 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 $\sigma_k'$ columns to $\bot$.
  2) Sets appropriate $\sigma_k'$ columns to $\top$ based on the $\sigma_i$ and $a_j$ columns, and $b_{\sigma_i, a_j, \sigma_k}$.
  3) Copies $\sigma_k'$ columns into the $\sigma_i$ columns.

  Note that this essentially simulates the execution of $FSA$ from the states encoded by the $\sigma_i$ 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 ≡
  $\top$ ?
  for i : $P_{n,q}$ do
    $P_{n,q}[i][\sigma'_1] := \bot; \ldots ; P_{n,q}[i][\sigma'_S] := \bot$;
  for i : $P_{n,q}$ do
    $P_{n,q}[i][\sigma_1] \wedge P_{n,q}[i][a_1] \wedge b_{\sigma_1,a_1,\sigma_1}$ ? $P_{n,q}[i][\sigma'_1] := \top$;
        ⋮
  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 ≡
  $\top$ ?
  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 $\forall i \in \mathbb{N} \centerdot \varphi(i)$ 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:

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

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 [4], that are not row-independent and row-uniform. In general, safety analysis for access-control-based systems in the style of the HRU model [22] allows for the posssibility of enforcing richer policies that are not expressible with our restrictions.

## 6. Related Work

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

**Parametric Verification for Correctness.** Parametric verification has been applied successfully to a wide variety of problems, including cache coherence [11], [13], bus arbitration [15], and resource allocation [10]. The general parametric model checking problem is undecidable [2], [42]. However, restricted versions of the problem, typically tailored to cache coherence protocols, yield decision procedures [12], [19]. Not surprisingly, these decision procedures are more efficient [9], [11], [14] when the problem is restricted to a greater degree.

We consider data-independent systems [45], for which efficient decision procedures [28], [29], that enable verification of *all* finite parameter instantiations by considering only a *finite* number of such instantiations, are available. However, to our knowledge, 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 over unbounded data structures. Existing formalisms for parameterized verification of data-independent systems either do not allow whole-array operations [28], or restrict them to a reset or copy operation that updates array elements to fixed values [29]. 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 [9].

**Parametric Verification for Security.** Lowe et al. [31] study parametric verification of authentication properties of network protocols. Roscoe and Broadfoot [37] apply data independence techniques to model check security protocols. Durgin et al. [8] show that small model theorems do not exist for a general class of security protocols. Millen [32] presents a family of protocols such that for any $k \in \mathbb{N}$, a member of the family has a cutoff greater than $k$. To our knowledge, we present the first small model theorems for system security.

**Model Checking for Security.** Guttman et al. [20] employ model checking to verify information-flow properties of SELinux. Lie et al. verify XOM [30] using Mur$\varphi$[3]. XOM is a hardware-based approach for tamper-resistance and copy-resistance. In contrast, SecVisor's goal is to protect the integrity of kernel code using memory protection. Mitchell et al. [33], [34] use Mur$\varphi$ to verify the correctness of (and find bugs in) security protocol specifications. We use Mur$\varphi$ only as an exemplar of a model checker to verify the cutoff instance. Our approach is amenable to verification via other model checkers, such as SPIN [24], TLA+ [27] and SMV [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 [6], [21], [46] target a general class of bugs. Others focus on specific types of errors, e.g., Kidd et al. [25] detect atomic set serializability violations, while Emmi et al. [16] verify correctness of reference counting implementation. All these approaches require abstraction, e.g., random isolation [25] or predicate abstraction [16], to handle source code, and therefore, are unsound and/or incomplete. In contrast, our focus is on bug detection, and verification, in the presence

3. `http://verify.stanford.edu/dill/murphi.html`
4. `http://www.cs.cmu.edu/~modelcheck/smv.html`

of an adversary with precisely defined capabilities. Also, we do not require abstraction, and are both sound and complete.

**Operating System Verification.** Prior work [35], [38], [41] 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. In contrast, our focus is on automated verification of manually constructed models that include an explicit adversary model.

A number of groups – Walker et al. [44] were one of the first – have used theorem proving to verify security properties of OS implementations. For example, Heitmeyer et al. [23] use PVS to verify correctness of a data separation kernel, while Klein et al. [26] use Isabelle to prove functional correctness properties of the L4 microkernel. In contrast, we use model checking to automatically verify security properties of systems that enforce protections with unbounded data structures.

## 7. Conclusion and Future Work

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 the paper 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.

In future work we plan to extend PGCL and PTSL to include security properties and programs that allow relations between rows of the parameterized array. These extensions will 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. Our long term goal is to extend these results and apply them to verify security properties of reference monitor implementations.

## Acknowledgment

We would like to thank Ning Qu for implementing and testing attacks against SecVisor. We thank our shepherd, David Molnar, for his guidance. We thank the anonymous reviewers and program committee for their helpful feedback and suggestions. This work was partially supported by the U.S. Army Research Office contract on Perpetually Available and Secure Information Systems (DAAD19-02-1-0389) to CMU's CyLab, the NSF Science and Technology Center TRUST, the NSF CyberTrust grant "Realizing Verifiable Security Properties on Untrusted Computing Platforms", and the Architecture Centric Engineering initiative at the Software Engineering Institute, CMU. Jason Franklin performed this research while under the support of an NSF Graduate Research Fellowship and a U.S. Department of Homeland Security (DHS) Fellowship.

## References

[1] J. P Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, ESD/AFSC, Hanscom AFB, Bedford, MA, Oct. 1972.

[2] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[3] 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 the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[4] 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.

[5] 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.

[6] 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.

[7] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.

[8] 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.

[9] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Proceedings of the 17th International Conference on Automated Deduction*, July 2000.

[10] E. Allen Emerson and Vineet Kahlon. Model checking large-scale and parameterized resource allocation systems. In *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, 2002.

[11] E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '03)*, pages 247–262, 2003.

[12] E. Allen Emerson and Vineet Kahlon. Model checking guarded protocols. In *18th Annual IEEE Symposium on Logic in Computer Science (LICS'03)*, 2003.

[13] E. Allen Emerson and Vineet Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03)*, 2003.

[14] E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *Proceedings of the 8th International Conference on Computer Aided Verification*, 1996.

[15] E. Allen Emerson and Kedar S. Namjoshi. Verification of parameterized bus arbitration protocol. In *Proceedings of the 10th International Conference on Computer Aided Verification*, 1998.

[16] 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.

[17] Jason Franklin, Sagar Chaki, Anupam Datta, and Arvind Seshadri. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. Technical Report CMU-CyLab-10-005, Carnegie Mellon University, 2010.

[18] 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.

[19] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[20] 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.

[21] 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.

[22] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8), August 1976.

[23] 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 the ACM Conference on Computer and Communications Security*, pages 346–355, 2006.

[24] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[25] 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.

[26] 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 the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[27] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[28] Ranko Lazić, Tom Newcomb, and Bill Roscoe. On model checking data-independent systems with arrays with whole-array operations. *Lecture Notes in Computer Science*, 3525:275–291, July 2004.

[29] 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 (TPLP)*, 4(5&6), 2004.

[30] 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.

[31] Gavin Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(1), 1999.

[32] Jonathan Millen. A necessarily parallel attack. In *Proceedings of the Workshop on Formal Methods and Security Protocols (FMSP)*, 1999.

[33] 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.

[34] 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.

[35] 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.

[36] IBM Research. The research hypervisor - a multi-platform, multi-purpose research hypervisor. http://www.research.ibm.com/hypervisor.

[37] 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.

[38] John Rushby. The design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating System Principles (SOSP)*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[39] 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.

[40] 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.

[41] Jonathan S. Shapiro and Sam Weber. Verifying the eros confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[42] I. Suzuki. Proving properties of a ring of finite state machines. *Information Processing Letters*, 28:213–213, 1988.

[43] 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.

[44] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[45] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, 1986.

[46] 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.