Parametric Verification of Address Space Separation

Jason Franklin, Sagar Chaki, Anupam Datta, Jonathan M. McCune, Amit Vasudevan

Carnegie Mellon University

Abstract. The address translation subsystem of operating systems, hypervisors, and virtual machine monitors must correctly enforce address space separation in the presence of adversaries. The size, and hierarchical nesting, of the data structures over which such systems operate raise challenges for automated model checking techniques to be fruitfully applied to them. We address this problem by developing a sound and complete parametric verification technique that achieves the best possible reduction in model size. Our results significantly generalize prior work on this topic, and bring interesting systems within the scope of analysis. We demonstrate the applicability of our approach by modeling shadow paging mechanisms of Xen version 3.0.3 and ShadowVisor, a research hypervisor developed for the x86 platform.

1 Introduction

A common use of protection mechanisms in systems software is to prevent one execution context from accessing memory regions allocated to a different context. For example, hypervisors, such as Xen [5], are designed to support memory separation not only among guest operating systems, but also between the guests and the hypervisor itself. Separation is achieved by an address translation subsystem that is self-contained and relatively small (around 7000 LOC in Xen version 3.0.3). Verifying security properties of such separation mechanisms is both: (i) important, due to their wide deployment in environments with malicious guests, e.g., the cloud; and (ii) challenging, due to their complexity. Addressing this challenge is the subject of our paper.

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

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

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

Second, we propose a parametric specification formalism for expressing security policies of separation mechanisms modeled in *PGCL*$^+$. Our formalism is able to express both safety and liveness properties (via a new logic *PTSL*$^+$) that involve arbitrary nesting of quantifiers over multiple levels of the nested parametric arrays in *PGCL*$^+$.

Third, we prove a set of *small model theorems* that roughly state that for any system M expressible in *PGCL*$^+$, and any security property φ in our specification formalism, an instance of M with a data structure of arbitrary size satisfies φ iff the instance of M where the data structure has 1 element at every level satisfies φ. These theorems yield the best possible reduction – e.g., verifying security of a separation mechanism over an arbitrarily large page table is reduced to verifying the mechanism with just 1 page table entry at each level. This ameliorates the lack of scalability of verification due to data structure size. For brevity, we defer proofs to the full version [17].

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

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 presents an overview of address translation mechanisms and associated separation properties. Section 4 presents the parametric modeling language, the specification logic, as well as the small model theorems and the key ideas behind their proofs. Section 5 presents the case studies. Finally, Section 6 presents our conclusions.

2 Related Work

Parametric verification has been applied to a wide variety of problems [10, 11, 13, 15], notably to verify cache coherence protocols [9, 11, 12, 14, 19]. However, we are distinguished by the focus on security properties in the presence of an adversary (or, attacker). Existing formalisms for parameterized verification of data-independent systems either do not allow whole-array operations [23], or restrict them to a reset or copy operation that updates array elements to fixed values [24]. Neither case can model our adversary.

Pnueli et al [26], Arons et al., [4], and Fang et al. [16] investigate finite bounded-data systems, which support stratified arrays that map to Booleans, a notion similar to our hierarchical arrays. However, they consider a restricted logic that allows for safety properties and a limited form of liveness properties referred to as response properties. In contrast, we consider both safety and more expressive liveness properties that can include both next state and until operators in addition to the forall operator. Moreover, the cutoffs of their small model theorems are a function of the type signatures, number of quantified index variables, and other factors. When instantiated with the same types used in our language, their small model theorems have larger cutoffs than our own. By focusing on the specific case of address translation systems and address separation properties, we are able to arrive at smaller cutoffs.

This paper generalizes our prior work [18] from a single parametric array to a tree of parametric arrays of arbitrary depth. The generalization requires new conceptual and technical insights and brings interesting systems (such as multi-level paging and context caching as used in Xen) within the scope of analysis. The concept of hierarchical row uniformity did not arise in the previous work. Moreover, our language *PGCL*$^+$ supports a more general form of guarded commands. At a technical level, the proofs are more involved because of the generality of the language and the logic. In particular, the use of mutual recursion in the definition of the programming language necessitates the use of mutual induction in establishing several key lemmas.

Neumann et al. [25], Rushby [27], and Shapiro and Weber [28] propose verifying the design of secure systems by manually proving properties using a logic and without an explicit adversary model. A number of groups [20, 22, 29] have employed theorem proving to verify security properties of OS implementations. Barthe et al. [6] formalized an idealized model of a hypervisor in the Coq proof assistant and Alkassar et al. [1, 2] and Baumann et al. [7] annotated the C code of a hypervisor and utilized the VCC verifier to prove correctness properties. Our approach is based on automatic verification via model checking.

3 Address Space Translation and Separation

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

3.1 Address Space Translation

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

More sophisticated address translation schemes use multi-level PTs. A n-level PT is essentially a set of tables linked to form a tree of depth n. Specifically, each row of a table at level i contains either a destination address, or the starting address of a table at level $i+1$. In addition to addresses, rows contain flags (e.g., to indicate if the

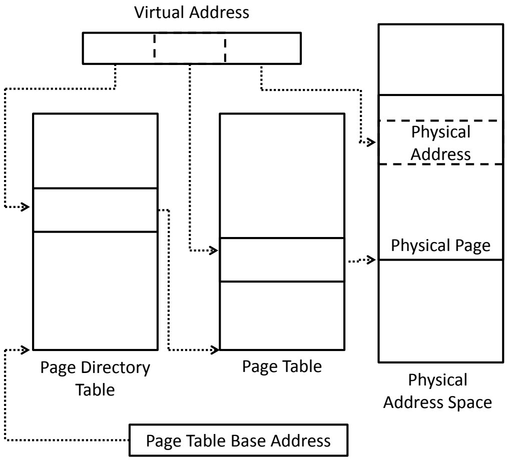

Fig. 1. Typical two-level page table structure.

row contains a destination addresses or the address of another table). We now present a concrete example.

Example 1. Figure 1 shows a typical two-level address translation. A 2-level PT consists of a top level Page Directory Table (PDT) and a set of leaf PTs. A source address i is split into two parts, whose sizes are determined during the design of the PT. Let $i = (i_1, i_2)$. To compute the destination address corresponding to i, we first find the row (i_1, o_1) in the PDT. The entry o_1 contains an address a_1, a Page Size Extension flag PSE, and a present flag $PRESENT$. If $PRESENT$ is unset, then there is no destination address corresponding to i. Otherwise, if PSE is set, then the destination address is a_1. Finally, if PSE is unset, we find the entry (i_2, a_2) in the table located at address a_1, and return a_2 as the destination address. Note the use of PSE and $PRESENT$ to disambiguate between different types of rows. Also, note the dual use of the address field a_1 as either a destination address or a table address.

3.2 Address Space Separation

While the systems we target are address translation schemes, the broad class of properties we aim for is address separation. This is a crucial property – in essence requiring that disjoint source addresses spaces be mapped to disjoint destination address spaces. Our notion of address separation is conceptually similar to that used by Baumann et al. [7]. Formally, an address translation scheme M violates separation if it maps addresses a_1 and a_2 from two different source address spaces to the same destination address. For example, an OS's virtual memory manager enforces separation between the address spaces of the OS kernel and various processes. Address space separation is a safety property since its violation is exhibited by a finite execution.

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

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

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

4 Definitions of $PGCL^+$ and $PTSL^+$

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

4.1 $PGCL^+$ Syntax

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

Let 1 and 0 be, respectively, the representations of the truth values **true** and **false**. Let $\mathtt{B}$ be a set of Boolean variables, $\mathtt{i}_1,\ldots,\mathtt{i}_d$ be variables used to index into $\mathtt{P}_1,\ldots,\mathtt{P}_d$, respectively, and $\mathtt{n}_1,\ldots,\mathtt{n}_d$ be variables used to store the number of rows of $\mathtt{P}_1,\ldots,\mathtt{P}_d$,

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

Fig. 2. *PGCL*$^+$ Syntax, z denotes depth.

respectively. The syntax of *PGCL*$^+$ is shown in Figure 2. *PGCL*$^+$ supports natural numbers, Boolean variables, propositional expressions over Boolean variables and F elements, guarded commands that update Boolean variables and F elements, and parallel composition of guarded commands. A skip command does nothing. A guarded command $\mathtt{e}\ ?\ \mathtt{c}_1 : \mathtt{c}_2$ executes $\mathtt{c}_1$ or $\mathtt{c}_2$ depending on if e evaluates to true or false. We write e ? c to mean e ? c : skip. The parallel composition of two guarded commands executes by non-deterministically picking one of the commands to execute. The sequential composition of two commands executes the first command followed by the second command. Note that commands at depth $z+1$ are nested within those at depth z.

Language Design Values assigned to an element of an F array at depth z can depend only on: (i) other elements of the same F array; (ii) elements of parent F arrays along the nesting hierarchy (to ensure hierarchical row uniformity); and (iii) Boolean variables. Values assigned to Boolean variables depend on other Boolean variables only. This is crucial to ensure row-independence which is necessary for our small model theorems (cf. Sec. 4.5).

4.2 ShadowVisor Code in *PGCL*$^+$

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

```
$\mathtt{shadow\_page\_fault} \equiv$
  for $i_1$ do
    $PDT[i_1].F[gPRESENT] \wedge PDT[i_1].F[gPSE] \wedge$
    $PDT[i_1].F[gADDR] < MEM\_LIMIT - MPS\_PDT$ ?
      $PDT[i_1].F[sADDR] := PDT[i_1].F[gADDR]$;
    for $i_2$ do
      $PDT[i_1].F[gPRESENT] \wedge PDT[i_1].PT[i_2].F[gPTE\_PRESENT] \wedge$
      $PDT[i_1].PT[i_2].F[gPTE\_ADDR] < MEM\_LIMIT - MPS\_PT$ ?
        $PDT[i_1].PT[i_2].F[sPTE\_ADDR] := PDT[i_1].PT[i_2].F[gPTE\_ADDR]$;
```

```
$\mathtt{shadow\_invalidate\_page} \equiv$
  for $i_1$ do
    $(PDT[i_1].F[sPRESENT] \wedge \neg PDT[i_1].F[gPRESENT]) \vee$
    $(PDT[i_1].F[sPRESENT] \wedge PDT[i_1].F[gPRESENT] \wedge$
    $(PDT[i_1].F[sPSE] \vee PDT[i_1].F[gPSE]))$ ?
      $PDT[i_1].F[sPDE] := 0$;
  for $i_1$ do
    $PDT[i_1].F[sPRESENT] \wedge PDT[i_1].F[gPRESENT] \wedge$
    $\neg PDT[i_1].F[gPSE] \wedge \neg PDT[i_1].F[sPSE]$ ?
      for $i_2$ do
        $PDT[i_1].PT[i_2].F[sPTE] := 0$;
```

```
$\mathtt{adversary} \equiv$
  for $i_1$ do
    $PDT[i_1].F[gPDE] := *$;
    for $i_2$ do
      $PDT[i_1].PT[i_2].F[gPTE] := *$;
```

```
$\mathtt{shadow\_new\_context} \equiv$
  for $i_1$ do
    $PDT[i_1].F[sPDE] := 0$;
```

Fig. 3. ShadowVisor model in *PGCL*$^+$.

For simplicity, let PDT be the top-level array P. Elements PDT[i$_1$].F[gPRESENT] and PDT[i$_1$].F[gPSE] are the present and page size extension flags for the i$_1$-th gPD entry, while PDT[i$_1$].F[gADDR] is the destination address contained in the i$_1$-th gPD entry. Elements sPRESENT, sPSE, and sADDR are defined analogously for sPD entries. Again for simplicity, let PDT[i$_1$].PT be the array P[i$_1$].P. Elements gPTE_PRESENT and gPTE_ADDR of PDT[i$_1$].PT[i$_2$].F are the present flag and destination address contained in the i$_2$-th entry of the PT pointed to by the i$_1$-th gPDT entry. Elements sPTE_PRESENT and sPTE_ADDR of PDT[i$_1$].PT[i$_2$].F are similarly defined for the sPDT. Terms gPDE refers to the set of elements corresponding to a gPDT entry (i.e., gPRESENT, gPSE, and gADDR). Terms gPTE, sPDE and sPTE are defined similarly for the gPT, sPDT, and sPT, respectively.

Our ShadowVisor model (see Figure 3) is a parallel composition of four guarded commands shadow_page_fault, shadow_invalidate_page, shadow_new_context, and adversary. Command shadow_page_fault synchronizes sPDT and sPT with gPDT and gPT when the guest kernel: (i) loads a new gPT, or (ii) modifies or creates a gPT entry. To ensure separation, shadow_page_fault does not copy addresses from the gPT or gPDT that allow access to addresses at or above MEM_LIMIT. This requires two distinct checks depending on the level of the table since pages mapped in the PDT are of size MPS_PDT and pages mapped in the PT are of size MPS_PT. Command shadow_invalidate_page invalidates entries in the sPD and sPT (by setting to zero) when the corresponding guest entries are not present, the *PSE* bits are inconsistent, or if both structures are consistent and the guest OS invalidates a page. Command shadow_new_context initializes a new context by clearing all the entries of the sPD. Finally, command adversary models the attacker by arbitrarily modifying every gPD entry and every gPT entry.

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

4.3 $PGCL^+$ Semantics

We now present the operational semantics of $PGCL^+$ as a relation on stores. Let $\mathbb{B}$ be the truth values $\{\mathbf{true}, \mathbf{false}\}$. Let $\mathbb{N}$ denote the set of natural numbers. For two natural numbers j and k such that $j \leq k$, we write $[j,k]$ to mean the set of numbers in the closed range from j to k. For any numeral k we write $\lceil k \rceil$ to mean the natural number represented by k in standard arithmetic. Often, we write k to mean $\lceil k \rceil$ when the context disambiguates such usage.

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

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

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

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

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

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

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

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

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

Fig. 4. Rules for commands

We omit the superscript of σ when it is clear from the context. The rules for evaluating *PGCL*$^+$ expressions under stores are defined inductively over the structure of *PGCL*$^+$ expressions, and shown in Figure 5. To define the semantics of *PGCL*$^+$, we first present the notion of store projection.

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

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

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

Fig. 5. Rules for expression evaluation. $[\wedge]$, $[\vee]$, and $[\neg]$ denote logical conjunction, disjunction, and negation, respectively.

We overload the $\mapsto$ operator as follows. For any function $f : X \to Y$, $x \in X$ and $y \in Y$, we write $f[x \mapsto y]$ to mean the function that is identical to f, except that x is mapped to y. $\mathsf{X}[y \mapsto w]$ is a tuple that equals X, except that $(\mathsf{X}[y \mapsto w])_y = w$. For any *PGCL*$^+$

expression or guarded command $\mathtt{X}$, variable $\mathtt{v}$, and expression $\mathtt{e}$, we write $\mathtt{X}[\mathtt{v} \mapsto \mathtt{e}]$ to mean the result of replacing all occurrences of $\mathtt{v}$ in $\mathtt{X}$ simultaneously with $\mathtt{e}$. For any $z \in \mathbb{N}$, $\mathtt{1}^z$ denotes the tuple of z $\mathtt{1}$'s.

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

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

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

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

Store Transformation. For any *PGCL*$^+$ command $\mathtt{c}$ and stores σ and σ', we write $\{\sigma\}$ $\mathtt{c}$ $\{\sigma'\}$ to mean that σ is transformed to σ' by the execution of $\mathtt{c}$. We define $\{\sigma\}$ $\mathtt{c}$ $\{\sigma'\}$ via induction on the structure of $\mathtt{c}$, as shown in Figure 4.

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

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

4.4 Specification Formalism

We support both reachability properties and temporal logic specifications. 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. The syntax of state formulas is defined in Figure 6. We support three types of state formulas – universal, existential, and generic. Specifically, universal formulas allow only nested universal quantification over $\mathtt{P}$, existential formulas allow arbitrary quantifier nesting with at least one $\exists$, while generic formulas allow one of each.

Temporal logic specifications are expressed in *PTSL*$^+$, a new logic we propose in this paper. In essence, *PTSL*$^+$ is a subset of the temporal logic ACTL* [8] with USF as atomic propositions. The syntax of *PTSL*$^+$ is defined in Figure 6. The quantification nesting allowed in our specification logic allows expressive properties spanning multiple levels of $\mathtt{P}$. This will be crucial for our case studies, as shown in Sec. 5.

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

Fig. 6. Syntax of $PTSL^+$ ($1 \leq z \leq d$). In ESF, Æ_y is $\forall$ or $\exists$, at least one Æ_y is $\exists$.

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

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

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

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

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

Definition 2 *The satisfaction of a formula π by a store σ (denoted $\sigma \models \pi$) is defined, by induction on the structure of π, as follows:*

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

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

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

Definition 3 *Let Init* $\in \mathsf{USF}$ *be any formula. Formally,* $M(\mathsf{gc}(\mathsf{k}),\mathit{Init})$ *is a four tuple* $(\mathcal{S},I,\mathcal{T},\mathcal{L})$*, where:*

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

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

If ϕ is a *PTSL*$^+$ formula, then $M,\sigma \models \phi$ means that ϕ holds at state σ in the Kripke structure M. We use an inductive definition of $\models$ [8]. Informally, an atomic proposition π holds at σ iff $\sigma \models \pi$; $\mathbf{A}\ \phi$ holds at σ if ϕ holds on all possible (infinite) paths starting from σ. TLPF formulas hold on paths. A TLF formula ϕ holds on a path Π iff it holds at the first state of Π; $\mathbf{X}\ \phi$ holds on a path Π iff ϕ holds on the suffix of Π starting at second state of Π; $\phi_1\ \mathbf{U}\ \phi_2$ holds on Π if ϕ_1 holds on suffixes of Π until ϕ_2 begins to hold. The definitions for $\neg$, $\wedge$ and $\vee$ are standard.

Simulation. For Kripke structures M_1 and M_2, we write $M_1 \preceq M_2$ to mean that M_1 is simulated by M_2. We use the standard definition of simulation [8] (presented in the full version of our paper [17]). Since satisfaction of ACTL* formulas is preserved by simulation [8], and *PTSL*$^+$ is a subset of ACTL*, we claim that *PTSL*$^+$ formulas are also preserved by simulation.

4.5 Small Model Theorems

In this section, we present two small model theorems. Both theorems relate the behavior of a *PGCL*$^+$ program when `P` has arbitrarily many rows to its behavior when `P` has a single row. First, a definition.

Definition 1 (Exhibits). *A Kripke structure $M(\mathsf{gc}(\mathsf{k}), Init)$ exhibits a formula φ iff there is a reachable state σ of $M(\mathsf{gc}(\mathsf{k}), Init)$ such that $\sigma \models \varphi$.*

The first theorem applies to safety properties.

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

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

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

Since, simulation preserves *PTSL*$^+$ specifications, we obtain the following immediate corrollary to Theorem 2.

Corollary 1 (Small Model Safety 2). *Let $\mathsf{gc}(\mathsf{k})$ 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 $M(\mathsf{gc}(\mathsf{k}), Init)$ exhibits φ iff $M(\mathsf{gc}(\mathtt{1}^d), Init)$ exhibits φ.*

Note that Corollary 1 is the dual of Theorem 1 obtained by swapping the types of φ and *Init*. The proofs of Theorems 1 and 2 involve mutual induction over both the structure of commands, and the depth of the parametric array `P`. This is due to the recursive nature of *PGCL*$^+$, where commands at level z refer to paramaterized commands at level z, which in turn refer to commands at level $z+1$. For brevity, we defer these proofs to the full version of our paper [17].

5 Case Studies

We present two case studies – ShadowVisor and Xen – to illustrate our approach. In addition to these two examples, we believe that our approach is, in general, applicable to all paging systems that are strictly hierarchical. This includes paging modes of x86 [21], and paging modes of ARM except for the super-pages [3] (due to the requirement that 16 adjacent entries must be identical).

PT-Size	Time(s)	Vars	Clauses
1	0.07	1816	3649
10	3.48	93503	199752
20	37.8	360275	775462
30	*	*	*

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

5.1 ShadowVisor

Recall our model of ShadowVisor from Section 4.2 and the expression of ShadowVisor's initial condition and security properties as $PTSL^+$ formulas from Section 4.4. ShadowVisor's separation property states that the physical addresses accessible by the guest OS must be less than the lowest address of hypervisor protected memory, denoted MEM_LIMIT. This requires two distinct conditions depending on the table containing the mapping. Pages mapped in PDTs are of size MPS_PDT and pages mapped in PTs are of size MPS_PT. Given a page frame of size s with starting address a, a guest OS can access any address in the range $[a, a+s]$. Hence, subtracting the maximum page size prevents pages from overlapping the hypervisor's protected memory. Note that we are making the reasonable assumption that MEM_LIMIT $>$ MPS_PDT and MEM_LIMIT $>$ MPS_PT to avoid underflow.

In ShadowVisor's original shadow page fault handler (shown in shadow_page_fault_original), the conditionals allowed page directory and page table entries to start at addresses up to MEM_LIMIT. As a result, ShadowVisor running shadow_page_fault_original has a serious vulnerability where separation is violated by an adversary that non-deterministically chooses an address a such that $a +$ MPS_PDT $\geq$ MEM_LIMIT or $a +$ MPS_PT $\geq$ MEM_LIMIT.

```
shadow_page_fault_original $\equiv$
  for $i_1$ do
    PDT[$i_1$].F[gPRESENT] $\wedge$ PDT[$i_1$].F[gPSE] $\wedge$ PDT[$i_1$].F[gADDR] $<$ MEM_LIMIT ?
      PDT[$i_1$].F[sADDR] := PDT[$i_1$].F[gADDR];
    for $i_2$ do
      PDT[$i_1$].F[gPRESENT] $\wedge$ PDT[$i_1$].PT[$i_2$].F[gPTE_PRESENT] $\wedge$
      PDT[$i_1$].PT[$i_2$].F[gPTE_ADDR] $<$ MEM_LIMIT ?
        PDT[$i_1$].PT[$i_2$].F[sPTE_ADDR] := PDT[$i_1$].PT[$i_2$].F[gPTE_ADDR];
```

Verification of our initial model of ShadowVisor detected this vulnerability. The vulnerability exists in ShadowVisor's design and C source code implementation. We were able to fix the vulnerability by adding appropriate checks and verify that the resulting model is indeed secure. We present our verification of $PGCL^+$ models below.

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

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

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

Basis: The initial state of the system ensures separation;

Inductive step: If the system started in a state that ensures separation, executing any of the four guarded commands in the ShadowVisor model preserves separation.

By induction, this guarantees that ShadowVisor ensures separation perpetually. Our results are shown in Table 1. Note that verification for size 1 (which is sound and complete due to our small model theorem) is quick, while it blows up for even page tables of size 30 (an unrealistically small number, implying that brute-force verification of ShadowVisor is intractable). The tools and benchmarks for our experiments are available at `https://www.cs.cmu.edu/~jfrankli/post12/vrfy-expr.tgz`.

5.2 Xen

Next, we analyzed address separation in a model of the Xen hypervisor, built from the source code of Xen version 3.0.3. Xen manages multiple virtual machines (VMs), each running a guest OS instance with multiple processes (i.e., contexts). Xen maintains a separate sPT for each context, and uses context caching (cf. Sec. 3).

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

Our separation property requires that the destination addresses accessible by a guest OS are less than a pre-defined constant $\mathtt{MEM\_LIMIT}$. We consider a natural extension of this separation property for a context caching system with multiple VMs that states that all VMs and contexts should be separate from VMM protected memory. This security property is stated in the following USF formula:

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

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

We model Xen as starting in an initial state where all entries of all of the shadow page directory tables and shadow page tables are marked as not present. This is expressed by the following USF formula:

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

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

```
XenAddressTrans ≡    shadow_page_fault
                   ‖ shadow_invalidate_page
                   ‖ context_caching_new_context
                   ‖ Xen_adversary
```

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

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

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

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

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

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

Effectiveness of Small Model Theorems. As in the case of ShadowVisor we verify the Xen model with increasing (but equal) sizes of page tables at both levels, and 2 VMs and 2 contexts per VM. We verify the same two properties as for ShadowVisor

PT-Size	Time(s)	Vars	Clauses
1	0.41	5726	13490
3	2.38	34192	80802
6	12.07	121206	286650
9	*	*	*

Table 2. Xen verification with increasing PT size. * means out of 1GB memory limit; Vars, Clauses = # of CNF variables and clauses generated by CBMC.

to inductively prove that Xen ensures separation perpetually. Our results are shown in Table 2. Note again that verification for size 1 (which is sound and complete due to our small model theorem) is quick, while it blows up for even page tables of size 9 (an unrealistically small number, implying that brute-force verification of Xen is also intractable).

6 Conclusion

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

References

1. Alkassar, E., Cohen, E., Hillebrand, M., Kovalev, M., Paul, W.: Verifying shadow page table algorithms. In: Proceedings of FMCAD (2010)
2. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: Proceedings of VSTTE (2010)
3. ARM Holdings: ARM1176JZF-S technical reference manual. Revision r0p7 (2009)
4. Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.: Parameterized verification with automatically computed inductive assertions. In: Proceedings of CAV (2001)
5. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of SOSP (2003)
6. Barthe, G., Betarte, G., Campo, J.D., Luna., C.: Formally verifying isolation and availability in an idealized model of virtualization. In: Proc. of FM (2011)
7. Baumann, C., Blasum, H., Bormer, T., Tverdyshev, S.: Proving memory separation in a microkernel by code level verification. In: Proc. of AMICS (2011)

8. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge, MA (2000)
9. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: Proc. of CADE (2000)
10. Emerson, E.A., Kahlon, V.: Model checking large-scale and parameterized resource allocation systems. In: Proc. of TACAS (2002)
11. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Proc. of CHARME (2003)
12. Emerson, E.A., Kahlon, V.: Model checking guarded protocols. In: Proceedings of LICS (2003)
13. Emerson, E.A., Kahlon, V.: Rapid parameterized model checking of snoopy cache coherence protocols. In: Proc. of TACAS (2003)
14. Emerson, E.A., Namjoshi, K.S.: Automatic verification of parameterized synchronous systems (extended abstract). In: Proceedings of CAV (1996)
15. Emerson, E.A., Namjoshi, K.S.: Verification of parameterized bus arbitration protocol. In: Proceedings of CAV (1998)
16. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.: Liveness with invisible ranking. In: Proceedings of VMCAI (2003)
17. Franklin, J., Chaki, S., Datta, A., McCune, J.M., Vasudevan, A.: Parametric verification of address space separation. Tech. Rep. CMU-CyLab-12-001, CMU (2012)
18. Franklin, J., Chaki, S., Datta, A., Seshadri, A.: Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In: Proceedings of IEEE S&P (2010)
19. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. Journal of the ACM 39(3), 675–735 (1992)
20. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.D.: Formal specification and verification of data separation in a separation kernel for an embedded system. In: Proceedings of ACM CCS (2006)
21. Intel Corporation: Intel 64 and IA-32 Intel architecture software developer's manual. Intel Publication nos. 253665–253669 (2008)
22. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an os kernel. In: Proceedings of SOSP (2009)
23. Lazić, R., Newcomb, T., Roscoe, B.: On model checking data-independent systems with arrays with whole-array operations. LNCS 3525, 275–291 (July 2004)
24. Lazić, R., Newcomb, T., Roscoe, A.: On model checking data-independent systems with arrays without reset. Theory and Practice of Logic Programming 4(5&6) (2004)
25. Neumann, P., Boyer, R., Feiertag, R., Levitt, K., Robinson, L.: A provably secure operating system: The system, its applications, and proofs. Tech. rep., SRI International (1980)
26. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Proc. of TACAS (2001)
27. Rushby, J.: The design and verification of secure systems. In: Proceedings of SOSP. (ACM *OS Review*, Vol. 15, No. 5)
28. Shapiro, J.S., Weber, S.: Verifying the eros confinement mechanism. In: Proceedings of IEEE S&P (2000)
29. Walker, B.J., Kemmerer, R.A., Popek, G.J.: Specification and verification of the UCLA Unix security kernel. CACM 23(2), 118–131 (1980)