Verification of a Generative Separation Kernel

Inzemamul Haque[1(✉)], D. D'Souza[1], P. Habeeb[1], A. Kundu[2], and Ganesh Babu[2]

[1] Indian Institute of Science, Bangalore, India
inzemamul@iisc.ac.in
[2] CAIR, Defence Research and Development Organization, Bangalore, India

Abstract. We present a formal verification of the functional correctness of the Muen Separation Kernel. Muen is representative of the class of modern separation kernels that leverage hardware virtualization support, and are *generative* in nature in that they generate a specialized kernel for each system configuration. We propose a verification framework called conditional parametric refinement which allows us to formally reason about generative systems. We use this framework to prove the correctness of Muen. Our analysis of several system configurations shows that our technique is effective in producing mechanized proofs of correctness, and also in identifying issues that may compromise the separation property.

1 Introduction

A separation kernel (SK) is a small specialized operating system or microkernel, that provides a sand-boxed or "separate" execution environment for a given set of processes (or "subjects"). The subjects may communicate only via declared memory channels, and are otherwise isolated from each other. Unlike a general operating system these kernels have a fixed set of subjects to run according to a specific schedule on the different CPUs of a processor-based system. Such kernels are often employed in security and safety-critical applications in military and aerospace domains, and the correct functioning of the kernel is of critical importance in guaranteeing the secure and timely execution of the subjects.

One way of obtaining a high level of assurance in the correct functioning of a system is to carry out a refinement-based proof of functional correctness [17,18], as has been done in the context of OS verification [22,31]. Here one specifies an abstract model of the system's behaviour, and then shows that the system implementation conforms to the abstract specification. Such a proof subsumes standard security properties related to separation, like no-exfiltration/infiltration and temporal and spatial separation of subjects considered for instance in [16].

Our aim here is to carry out a similar refinement-based proof of functional correctness for the Muen separation kernel [6], which is an open-source representative of a class of modern separation kernels (including commercial products [13,25,27,34,36]) that use hardware virtualization support and are *generative*

D. V. Hung and O. Sokolsky (Eds.): ATVA 2020, LNCS 12302, pp. 305–322, 2020.
https://doi.org/10.1007/978-3-030-59152-6_17

in nature. By the latter we mean that these tools take an input specification describing the subjects and the schedule of execution, and generate a tailor-made processor-based system that includes subject binaries, page tables, and a kernel that acts like a Virtual Machine Monitor (VMM).

There are several challenges in carrying out such an exercise. Each generated system employs a mix of Ada, Assembly, hardware virtualization features, and complex 4-level paging structures, and is challenging to reason about as a stand-alone system. However, the main challenge lies in reasoning about the generative aspect of such a system: we need to show that for *every* possible input specification, the kernel generator produces a correct system. A possible approach to handle this challenge could be to verify the generator code, along the lines of the CompCert project [24]. However with the generator code running close to 41K LOC, with little compositional structure, this would be a formidable task. Translation validation [30] is another possibility but would require manual effort from scratch each time.

We overcame the challenge of virtualization by simply choosing to model the virtualization layer (in this case Intel's VT-x layer) along with the rest of the hardware components like registers and memory, programmatically in software. Thus we modeled VT-x components like the per-CPU VMX-Timer and EPTP as 64-bit variables in Ada, and implicit structures like the VMCS as a record with appropriate fields as specified by Intel [19]. Instructions like VMLAUNCH were then implemented as methods that accessed these variables. In many ways, virtualization turned out to be more of a boon than a bane. We solved the problem of generativeness (and the problem of handling page tables too), by leveraging a key feature of such systems: the kernel is essentially a *template* which is largely fixed, independent of the input specification. The kernel accesses variables which represent input-specific details like subject details and the schedule, and these structures are generated by Muen based on the given input specification. The kernel can thus be viewed as a *parametric* program, much like a method that computes using its formal parameter variables. In fact, taking a step back, the whole processor system generated by Muen can be viewed as a parametric program with parameter values like the schedule, subject details, page tables, and memory elements being filled in by the generator based on the input specification. This view of Muen as a parametric program turned out to be the key enabler for us.

Such a view suggests a novel two-step technique for verifying generative systems that can be represented as parametric programs. We call this approach *conditional parametric refinement*. We first perform a general verification step (independent of the input spec) to verify that the parametric program refines a parametric abstract specification, *assuming* certain natural conditions on the parameter values (for example *injectivity* of the page tables) that are to be filled in. This first step essentially tells us that for *any* input specification P, if the parameter values generated by the system generator satisfy the assumed conditions, then the generated system is correct vis-a-vis the abstract specification. In the second step, which is *input-specific*, we check that for a given input specifi-

cation, the generated parameter values actually satisfy the assumed conditions. This gives us an effective verification technique for verifying generative systems that lies somewhere between verifying the generator and translation validation.

We carried out the first step of this proof technique for Muen, using the Spark Ada [2] verification environment. The effort involved about 20K lines of source code and annotation. No major issues were found, modulo some subjective assumptions we discuss in Sect. 4.3. We have also implemented a tool that automatically and efficiently performs the Step 2 check for a given SK configuration. The tool is effective in proving the assumptions, leading to machine-checked proofs of correctness for 16 different input configurations, as well as in detecting issues like undeclared sharing of memory components in some seeded faulty configurations.

In the sequel we sketch the main components of our theory and its application to the verification of Muen. For further details we refer the reader to the longer version [15].

2 Conditional Parametric Refinement

2.1 Machines and Refinement

A convenient way to reason about systems such as Muen is to view them as an *Abstract Data Type* or *machine* [1]. A machine $\mathcal{A}$ is essentially a set of states along with a set of operations, each of which takes an argument, transforms the current state, and returns an output value. We have a designated intialization operation called *init*. The machine $\mathcal{A}$ induces a transition system $\mathcal{T}_{\mathcal{A}}$ in a natural way, whose states are the states of $\mathcal{A}$, and transitions from one state to another are labelled by triples of the form (n, a, b), representing that operation n with input a was invoked and the return value was b. We denote the language of *initialized* sequences of operation calls produced by this transition system, by $L_{init}(\mathcal{A})$.

We will consider machines represented as a program in an imperative programming language. Valuations for the variables of the program make up the state of the machine, while each operation n is given by a method definition of the same name. We call such a program a *machine program*. Figure 1(a) shows a program in a C-like language, that represents a "set" machine with operations *init*, *add* and *elem*. The set stores a subset of the numbers 0–3, in a Boolean array of size 4. However, for certain extraneous reasons, it uses an array T to permute the positions where information for an element x is stored. Thus to indicate that x is present in the set the bit $S[T[x]]$ is set to true.

Refinement [1,17,18] is a way of saying that a "concrete" machine conforms to an "abstract" one, behaviourally. In our setting of total and deterministic machines, we say that machine $\mathcal{B}$ *refines* machine $\mathcal{A}$ if $L_{init}(\mathcal{B}) \subseteq L_{init}(\mathcal{A})$. Refinement is typically exhibited using a "gluing" relation ρ which relates the states of $\mathcal{B}$ to those of $\mathcal{A}$. We say ρ is *adequate* to show that $\mathcal{B}$ refines $\mathcal{A}$ if it satisfies the conditions: (init) the states of $\mathcal{B}$ and $\mathcal{A}$ after initialization are related by ρ, and (sim) if states p and q are related by ρ then after doing any operation n

```
typedef univ
 0..3;
bool S[4];
univ T[4] :=
 {1,2,3,0};

void init(){
 for (int i:=0;
   i < 4; i++)
  S[i] := false;
}

void add(univ x){
 S[T[x]] := true;
}

bool elem(univ x){
 return S[T[x]];
}
```

(a)

```
// Abstract spec
typedef univ
 0..3;
bool absS[4];
void add(univ x){
 absS[x] :=
  true;
}

...

// Gluing
// relation
\forall univ x:
  S[T[x]] =
   absS[x]
```

(b)

```
const unsigned
 Usize;
typedef univ
 0..Usize-1;
bool S[Usize];
univ T[Usize];

void init(){
 for (int i:=0;
   i < Usize; i++)
  S[i] := false;
}

void add(univ x){
 S[T[x]] := true;
}

bool elem(univ x){
 return S[T[x]];
}
```

(c)

```
// Abstract param spec
const unsigned absUsize;
typedef absUniv
 0..absUsize-1
bool absS[absUsize];

void add(absUniv x){
 absS[x] := true;
}

...

// Assumption R:
   Usize = absUsize
     && T injective

// Param gluing relation
\forall univ x:
  S[T[x]] = absS[x]
```

(d)

Fig. 1. (a) A machine program P implementing a set machine, (b) an abstract specification A and gluing relation, (c) a parametric machine program $Q[Usize, T]$ representing a parametric set machine, and (d) abstract parametric specification $B[absUsize]$ and parametric gluing predicate.

with input a in these states, the output values agree and the resulting states are again ρ-related. To check the adequacy of a gluing relation, we can use Floyd-Hoare logic based code-level verification tools (like VCC [7] for C, or GNAT Pro [2] for Ada Spark), to phrase the refinement conditions (init) and (sim) as pre/post annotations and carry out a machine-checked proof of refinement [12]. Figure 1(b) shows an abstract specification and a gluing relation, for the set machine program of part (a).

2.2 Generative Systems and Parametric Refinement

A *generative system* is a program G that given an input specification I (in some space of valid inputs), generates a machine program P_I. As an example, one can think of a set machine generator *SetGen*, that given a number k of type unsigned int (representing the universe size), generates a program P_k similar to the one in Fig. 1(a), which uses the constant k in place of the set size 4, and an array T_k of size k, which maps each x in $[0..k-1]$ to $(x+1) \bmod k$. For every I, let us say we have an abstract machine (again similar to the one in Fig. 1(b)) say A_I, describing the intended behaviour of the machine P_I. Then the verification problem of interest to us, for the generative system G, is to show that for *each* input specification I, P_I refines A_I. This is illustrated in Fig. 2(a). We propose a way to address this problem using refinement of *parametric* programs, which we describe next.

Parametric Refinement. A *parametric* program is like a standard program, except that it has certain read-only variables which are left *uninitialized*. These

uninitialized variables act like "parameters" to the program. We denote by $P[\bar{V}]$ a parametric program P with a list of uninitialized variables $\bar{V}$. As such a parametric program has no useful meaning, but if we initialize the variables $\bar{V}$ with the values $\bar{v}$ passed to the program, we get a standard program which we denote by $P[\bar{v}]$. Let N be a set of operation names. A *parametric machine program* of type N is a parametric program $Q[\bar{V}]$ containing a method f_n for each operation $n \in N$. The input/output types of f_n may be dependent on and derived from the parameter values. Given a parameter value $\bar{v}$ for $\bar{V}$, we obtain the machine program $Q[\bar{v}]$. Each method f_n now has a concrete input/output type which we denote by $I_n^{\bar{v}}$ and $O_n^{\bar{v}}$ respectively. Figure 1(c) shows an example parametric machine program $Q[Usize, T]$, representing a parametric version of the set program in Fig. 1(a). Given a value 4 for *Usize* and a list $[1, 2, 3, 0]$ for T, we get the machine program $Q[4, [1, 2, 3, 0]]$, which behaves similar to the one of Fig. 1(a). We note that the input type of the methods *add* and *elem* depend on the value of the parameter *Usize*.

Given two parametric machine programs $Q[\bar{V}]$ and $B[\bar{U}]$ of type N, we are interested in exhibiting a refinement relation between instances of $Q[\bar{V}]$ and $B[\bar{U}]$. Let R be a relation on parameter values $\bar{u}$ for $\bar{U}$ and $\bar{v}$ for $\bar{V}$, given by a predicate on the variables in $\bar{U}$ and $\bar{V}$. We say that $Q[\bar{V}]$ *parametrically refines* $B[\bar{U}]$ w.r.t. the condition R, if whenever two parameter values $\bar{u}$ for $\bar{U}$ and $\bar{v}$ for $\bar{V}$ are such that $R(\bar{u}, \bar{v})$ holds, then $Q[\bar{v}]$ refines $B[\bar{u}]$. We propose a way to exhibit such a conditional refinement using a *single* "universal" gluing relation. A *parametric gluing relation* on $Q[\bar{V}]$ and $B[\bar{U}]$ is a relation π on the state spaces S^Q of $Q[\bar{V}]$ and S^B of $B[\bar{U}]$, given by a predicate on the variables of $Q[\bar{V}]$ and $B[\bar{U}]$. We say π is *adequate*, with respect to the condition R, if the following conditions are satisfied. In the conditions below, we use the standard Hoare triple notation for total correctness $\{G\}\ \boxed{P}\ \{H\}$, to mean that a program P, when started in a state satisfying predicate G, always terminates in a state satisfying H. We use the superscript Q or B to differentiate the components pertaining to the programs $Q[\bar{V}]$ and $B[\bar{U}]$ respectively, and assume that the programs have disjoint state spaces.

1. (type) For each $n \in N$: $R(\bar{u}, \bar{v}) \implies (I_n^{Q,\bar{v}} = I_n^{B,\bar{u}} \wedge O_n^{Q,\bar{v}} = O_n^{B,\bar{u}})$.
2. (init) $\{R\}\ \boxed{init^B(); init^Q()}\ \{\pi\}$.
3. (sim) For each $n \in N$: $\{R \wedge \pi\}\ \boxed{r_B := f_n^B(a); r_Q := f_n^Q(a)}\ \{\pi \wedge r_B = r_Q\}$.

We can now state the following theorem:

Theorem 1. *Let $Q[\bar{V}]$ and $B[\bar{U}]$ be parametric machine programs of type N. Let R be a predicate on $\bar{U}$ and $\bar{V}$, and let π be an adequate parametric gluing relation for $Q[\bar{V}]$ and $B[\bar{U}]$ w.r.t. R. Then $Q[\bar{V}]$ parametrically refines $B[\bar{U}]$ w.r.t. the condition R.* □

Consider the parametric machine program $Q[Usize, T]$ in Fig. 1(c), and the abstract parametric program in Fig. 1(d), which we call $B[absUsize]$. Consider the condition R which requires that $absUsize = Usize$ and T to be injective. Let

π be the parametric gluing predicate $\forall x : \text{unsigned}, (x < Usize) \implies S[T[x]] = absS[x]$). Then π can be seen to be adequate w.r.t. the condition R, and thus $Q[Usize, T]$ parametrically refines $B[absUsize]$ w.r.t. R.

Verifying Generative Systems using Parametric Refinement. Consider a generative system G that given a specification I, generates a machine program P_I by filling a template with values derived from I, and let A_I be the abstract specification for input I. Recall that our aim is to show that for each I, P_I refines A_I. We achieve this by applying the following steps:

1. Associate a parametric program $Q[V]$ with G, such that for each I, G can be viewed as generating the value v_I for the parameter V, so that $Q[v_I]$ is behaviourally equivalent to P_I. $Q[V]$ can be constructed from the template which is filled by G.
2. Construct a parametric abstract specification $B[U]$, and concrete value u_I for each I, such that A_I is equivalent to $B[u_I]$.
3. Construct a condition R on the parameters U and V, and show that $Q[V]$ parametrically refines $B[U]$ w.r.t. R, using an adequate gluing predicate.
4. For a given I, check if u_I and v_I satisfy R. If so, conclude that P_I refines A_I.

We note that the Steps 1–3 are done only once for G, while the last step needs to be done for each I of interest. Figure 2 illustrates this approach.

As a final illustration in our running example, to verify the correctness of the set machine generator *SetGen*, we use the parametric programs $Q[Usize, T]$ and $B[absUsize]$ to capture the concrete program generated and the abstract specification respectively. We then show that $Q[Usize, T]$ parametrically refines $B[absUsize]$ w.r.t. the condition R, using the gluing predicate π, as described above. We note that the actual values generated for the parameters in this case (recall that these are values for the parameters *Usize*, *absUsize* and T) do indeed satisfy the conditions required by R, namely that *Usize* and *absUsize* be equal, and T be injective. Thus we can conclude that for each input universe size k, the machine program P_k refines A_k, and we are done.

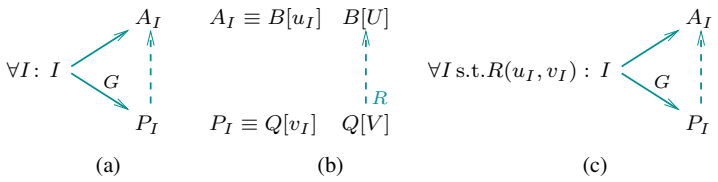


Fig. 2. Proving correctness of a generative system using parametric refinement. (a) The goal, (b) proof artifacts and obligation, and (c) the guarantee. Dashed arrows represent refinement, dashed arrow with R at tail represents conditional refinement w.r.t. R.

3 The Muen Kernel Generator

3.1 Intel X86/64 Processor

The kernel that Muen generates runs on an x86/64 processor with VMX support. We briefly describe this platform (see [19] for more details) and how to view a processor-based system as a machine. Figure 3 depicts the processor system and its components. The CPU components like the 64-bit general purpose registers, model-specific registers like the Time Stamp Counter (TSC), and physical memory components are standard. The layer above shows components like the VMCS pointer (VMPTR), the VMX-timer, and extended page table pointer (EPTP), which are part of the VT-x layer of the VMX mode that supports virtualization. The VMPTR on each CPU points to a VMCS structure, which is used by the VMM (here the kernel) to control launch/exit of subjects. The CR3 register and the EPTP component (set by the active VMCS) control the virtual-to-physical address translation. The top-most layer shows the kernel code that runs on each CPU, with an "Init" component that runs on system initialization, and a "Handler" component that handles VM-exits due to interrupts. The Muen kernel essentially runs as a VMM, and subjects as VMs provided by the VMM. To launch a subject in a VM, the kernel sets the VMPTR to point to one of the VMCSs using the VMPTRLD instruction, and then calls VMLAUNCH which sets the timer, CR3, and EPTP components from the VMCS fields. A subject is caused to exit its VM and return control to the kernel by events like VMX-timer expiry, page table exceptions, and interrupts.

We would like to view such a processor system as a machine of Sect. 2.1. The state of the machine is the contents of all its components. The operations are (a) *Init*, where the init code of the kernel is executed on each of the processors starting with the BSP (CPU0); (b) *Execute*, which takes a CPU id and executes the next instruction pointed to by the IP on that CPU. The instruction could be one that does not access memory (like *add*), or one that accesses memory (like *mov*) in which case the given address is translated via the page tables pointed to by the CR3 and EPTP components; or (c) *Event*, which could be timer tick event on a CPU causing the TSC to increment and the VMX-timer of the active

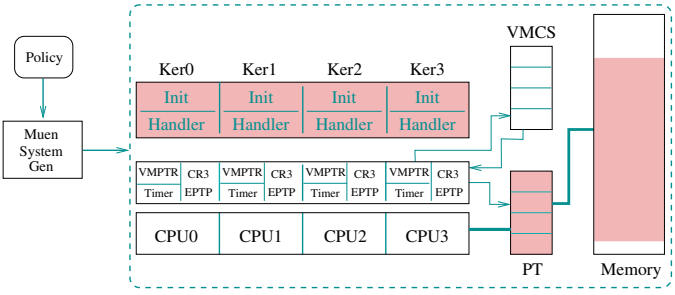
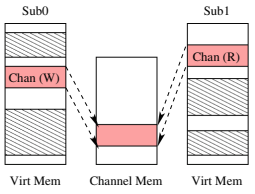


Fig. 3. An x86/64bit VMX processor. Shaded components are generated by Muen.

VM to decrement. If the VMX-timer becomes 0, a VM exit is caused and the corresponding handler invoked. Another kind of event is generated by external interrupts. External interrupts cause a VM exit. The cause of all VM exits is stored in the subject's VMCS, which the handler checks and takes appropriate action for.

3.2 Policy Specification

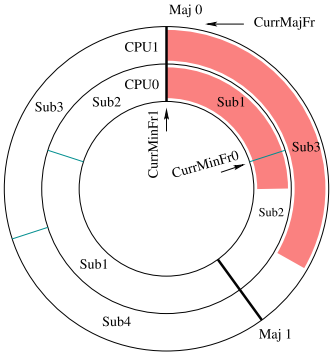
The input specification to Muen is an XML file called a *policy*. It specifies details of the host processor, subjects to be run, and a precise schedule of execution on each CPU of the host processor system. For each subject the policy specifies the size and starting addresses of the components in its virtual memory which could include shared memory components called *channels*. The policy specifies the size and location of each channel in a subject's virtual address space, and read/write permissions, as depicted alongside.



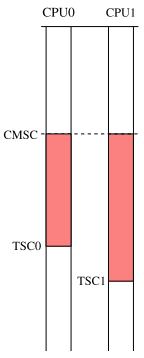
The schedule is a sequence of *major frames* to be performed repeatedly. A major frame specifies for each CPU a sequence of *minor frames*, which specifies a subject and the number of ticks to run it. The beginning of each major frame is a synchronization point for the CPUs. An example scheduling policy in XML is shown in Fig. 4(a), while Fig. 4(b) shows the same schedule viewed as a clock. The shaded portion depicts the passage of time (the tick count) on each CPU.

```
<scheduling tick_rate="10000">
 <major_frame>
  <cpu id="0">
   <minor_fr sub_id="1" ticks="40"/>
   <minor_fr sub_id="2" ticks="40"/>
  </cpu>
  <cpu id="1">
   <minor_fr sub_id="3" ticks="80"/>
  </cpu>
 </major_frame>
 <major_frame>
  <cpu id="0">
   <minor_fr sub_id="1" ticks="80"/>
   <minor_fr sub_id="2" ticks="40"/>
  </cpu>
  <cpu id="1">
   <minor_fr sub_id="4" ticks="60"/>
   <minor_fr sub_id="3" ticks="60"/>
  </cpu>
 </major_frame>
</scheduling>
```

(a)



(b)



(c)

Fig. 4. (a) Example schedule, (b) its clock view, and (c) its implementation in Muen.

3.3 Muen Kernel Generator

Given a policy C, Muen generates the components of a processor system S_C, which is meant to run according to the specified schedule. This is depicted

in Fig. 3, where the Muen toolchain generates the shaded components of the processor system, like the initial memory contents, page tables, and kernel code. The toolchain generates a kernel for each CPU, to orchestrate the execution of the subjects according to the specified schedule on that CPU. The kernel is actually a *template* of code written in Spark Ada, and the toolchain generates the constants for this template based on the given policy. The kernel uses data structures like *subject-specs* to store details like page table and VMCS address for each subject. To implement scheduling, the kernel uses a multidimensional array called *scheduling-plans* representing the schedule for each CPU. The kernel knows the number of ticks elapsed on each CPU from the TSC register. It uses a shared variable called CMSC, which is updated by the BSP, to keep track of the start of the current major frame, as shown in Fig. 4(c). The structure *vector-routing* is also generated by the toolchain to represent the table which maps an interrupt vector to the corresponding destination subject and the destination vector to be sent to the destination subject. The kernel also uses a data structure called *global-events* for each subject to save pending interrupts when the destination subject is not active. The components of the kernel are shown in Fig. 5.

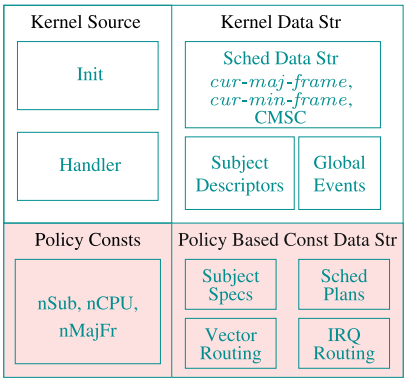


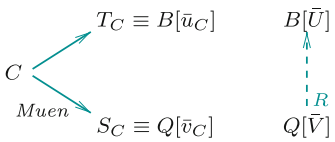
Fig. 5. Components of the generated kernel. Shaded components are generated by the toolchain.

At system startup the Init part of the kernel performs the initialization tasks like setting up the VMCS for each subject, making use of the *subject-specs* structure generated by Muen. The handler part of the kernel is invoked whenever there is a VM exit. For instance if the exit is due to a VMX-timer expiry, it uses *scheduling-plans* to decide whether to schedule the subject in the next minor frame, or to wait for synchronization at the end of a major frame. If the exit is due to an external interrupt, it uses *vector-routing* to decide the subject which will handle the interrupt, and the destination vector which should be sent to the handler subject. The structure *global-events* is used to store the pending interrupt. When the handler subject becomes active, the pending interrupt is *injected* via the VMCS and the pending interrupt is removed from *global-events*.

We have focussed on Ver. 0.7 of Muen. The toolchain implemented in Ada and C, comprises about 41K LoC, while the kernel template is about 3K LoC in Ada.

4 Proof Overview

Given a policy C, let S_C denote the processor system generated by Muen. Let T_C denote an abstract machine spec for the system S_C (we describe T_C next). Our aim is to show that for each valid policy C, S_C refines T_C. We use the parametric refinement technique of Sect. 2 to do this. We first define a parametric program $Q[\bar{V}]$ that models the generic system generated by Muen, so that for a given policy C, if $\bar{v}_C$ corresponds to the parameter values generated by Muen, then S_C and $Q[\bar{v}_C]$ are behaviourally equivalent. In a similar way we define the abstract parametric program $B[\bar{U}]$, so that with appropriate parameters $\bar{u}_C$, $B[\bar{u}_C]$ captures the abstract spec T_C. Next we show that $Q[\bar{V}]$ parametrically refines $B[\bar{U}]$ w.r.t. a condition R. The figure alongside shows the proof artifacts and obligations. Finally, for a given policy C, we check that the parameter values $\bar{u}_C$ and $\bar{v}_C$ satisfy the condition R. In the rest of this section we elaborate on the components and steps of this proof.



4.1 Abstract Specification

The abstract specification T_C implements a simple system that realizes the behaviour specified by a policy C. In T_C each subject s is run on a *separate*, dedicated, single-CPU processor system M_s. The system M_s has its own CPU with registers, and 2^{64} bytes of physical memory *VMem* with permissions as specified in the policy. The policy maps each subject to a CPU of the concrete machine on which it is meant to run. To model this we use a set of *logical* CPUs (corresponding to the number of CPUs specified in the policy), and we associate with each logical CPU, the (disjoint) group of subjects mapped to that CPU. Figure 6 shows a schematic representation of T_C. To model channels, we use a separate memory array *chmem*, as depicted in Sect. 3.2. Memory contents for a subject s are fetched from $VMem_s$ or from *chmem* accordingly. There is no kernel in this system, but a *supervisor* whose job is to process events directed to a logical CPU or subject, and to enable and disable subjects based on the scheduling policy and the current "time". Towards this end it maintains a flag $enabled_s$ for each subject s, which is set whenever the subject is enabled to run based on the current time. To implement the specified schedule it keeps track of time using the clock-like abstraction depicted in Fig. 4(b).

In the *init* operation the supervisor initializes the processor systems M_s, permissions array *perms*, the channel memory *chmem*, and also the schedule-related variables, based on the policy. The *execute* operation, given a logical CPU id, executes the next instruction on the subject machine currently active for that logical CPU id. An *execute* operation does not affect the state of other subject processors, except possibly via the shared memory *chmem*. If the instruction accesses an invalid memory address, the system is assumed to shut down in an error state. Finally, for the *event* operation, which is a tick/interrupt event

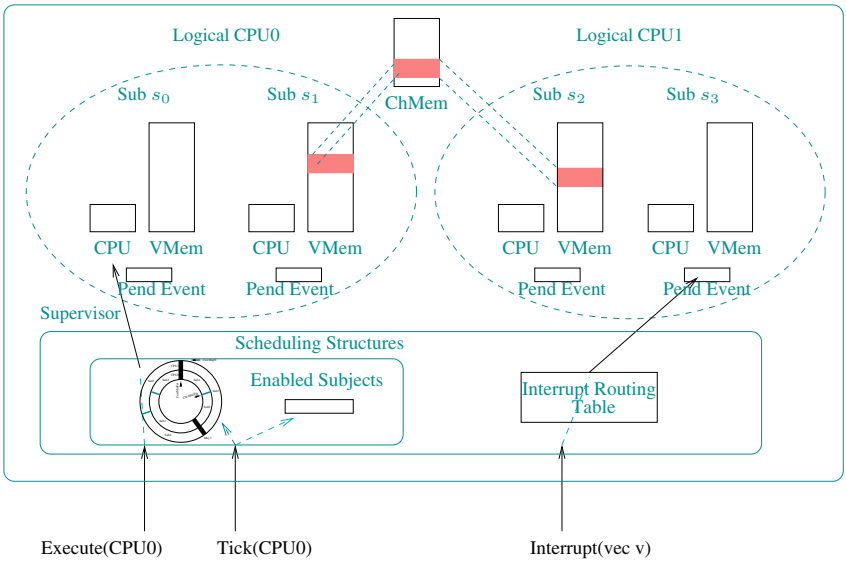


Fig. 6. Schematic diagram of the abstract spcecification T_C

directed to a logical CPU or subject, the supervisor updates the scheduling state, or pending event array, appropriately.

To represent the system T_C concretely, we use an Ada program which we call A_C. A_C is a programmatic realization of T_C, with processor registers represented as 64-bit numeric variables, and memory as byte arrays of size 2^{64}. The operations *init*, *execute*, and *event* are implemented as methods that implement the operations as described above. Finally, we obtain a parametric program $B[\bar{U}]$ from A_C, by parameterizing it as illustrated in Sect. 2. We call the list of parameters $\bar{U}$. By construction, it is evident that if we generate appropriate values $\bar{u}_C$ for the parameters in $\bar{U}$, we will get a machine program $B[\bar{u}_C]$ which is equivalent in behaviour to A_C.

4.2 Parametric Refinement Proof

We begin by describing how we view Muen as a parametric program. Let C be a given policy. We first define a machine program P_C that represents the processor system S_C generated by Muen. This is done similar to A_C, except that we now have a *single* physical memory array which we call *PMem*. Further, since the processor system S_C makes use of the VT-x components, we need to model these components in P_C as well. For example we represent each page table *ptab*, as a 2^{64} size array PT_{ptab} of 64-bit numbers, with the translation $ptab(a)$ of an address a being modelled as $PT_{ptab}[a]$. The operations *init*, *execute*, and *event* are implemented as method calls. The *init* code comes from the Init component of the kernel. In the *execute* method, memory accesses are translated via the

active page table to access the physical memory *PMem*. The implementation of the *event* operation comes from the Handler part of the kernel code. We then move from P_C to a parametric program $Q[\bar{V}]$, by using parameters such as *NSubs*, *scheduling-plans*, *subject-specs*, *PMem*, and *PT*. Once again, for an appropriate list of values $\bar{v}_C$ generated by Muen from a given policy C, $Q[\bar{v}_C]$ is equivalent to P_C, which in turn is equivalent to S_C.

Next we show that the parametric version of the Muen system $Q[\bar{V}]$ conditionally refines the parametric abstract spec $B[\bar{U}]$. From Sect. 2.2, this requires us to identify the condition R, and find a gluing relation π on the state of parametric programs Q and B such that the refinement conditions (type), (init), and (sim) are satisfied. The key conjuncts of our condition R are:

- R_1: The page tables $ptab_s$ associated with a subject s must be *injective* in that no two virtual addresses, within a subject or across subjects, may be mapped to the same physical address, *unless* specified to be part of a channel;
- R_2: For each subject s, the permissions (rd/wr/ex/present) associated with an address a should match with the permissions for a in $ptab_s$;
- R_3: For each subject s, no invalid virtual address is mapped to a physical address by page table $ptab_s$.
- R_4: The values of the parameters (like *NSubs*, *subject-specs*, *scheduling-plans* and *IOBitmap*) in the concrete should match with those in the abstract.

The gluing relation π has the following key conjuncts: The CPU register contents of each subject in the abstract match with the register contents of the CPU on which the subject is active, if the subject is enabled, and with the subject descriptor, otherwise; For each subject s and valid address a in its virtual address space, the contents of $VMem_s(a)$ and $PMem(ptab_s(a))$ should match; The value $(\text{TSC} - \text{CMSC})$ on each CPU in the concrete, should match with how much the ideal clock for the subject's logical CPU is ahead of the beginning of the current major frame in the abstract.

We carry out the adequacy check for π, described in Sect. 2.2, by constructing a "combined" version of Q and B that has the disjoint union of their state variables, as well as a joint version of their operations, and phrase the adequacy conditions as pre/post conditions on the joint operations. We carry out these checks using the Spark Ada tool [2] which uses provers Z3 [28], CVC4 [3], and Alt-Ergo [8] in the backend. We faced several challenges in carrying out this proof to completion. For instance, to prove the kernel's handling of the tick event correct, we used 8 subcases to break up the reasoning into manageable subgoals for both the engineer and the prover. The table alongside shows details of our proof effort in terms of lines of code (LoC) and lines of annotations (LoA) in the combined proof artifact. In the combined artifact the LoC count includes comments and repetition of code due to case-splits. All the proof artifacts used in this project are available at `shorturl.at/ilqMU`.

$B[\bar{U}]$		$Q[\bar{V}]$		Combined	
LoC	LoA	LoC	LoA	LoC	LoA
793	0	1,914	0	13,970	6,214

4.3 Checking Condition R

We now describe how to efficiently check that for a given policy C, the parameters generated by Muen and those of the abstract specification, satisfy the condition R. A naive way to check R would be to iterate over the virtual addresses for each subject and check the conditions. This runs in time $\mathcal{O}(N_v)$ where N_v is the size of the virtual address space (typically 2^{48}), and would take days to run. Instead we exploit the fact that the actual size of the memory components is relatively small. We make use of Muen's B-policy which defines the physical address and size of physical memory segments, and the mapping of virtual components to it, so that checking R_1 reduces to checking overlap of physical components. To check R_3, we exploit the fact that translation of a valid virtual address uses certain entries of paging structures which have their "present" bit set to 1. We check that the present bit is set only in the entries which are used for translation of valid virtual addresses. These checks run in time $\mathcal{O}(N_u)$ where N_u is the actual used virtual address space of a subject.

We implemented our algorithms above in C and Ada, using the Libxml2 library to process policy files, and a Linux utility `xxd` to convert the Muen image and individual files from raw format to hexadecimal format. We ran our tool on 16 system configs, 9 of which (D7-*,D9-*) were available as demo configurations from Muen. The remaining configs (DL-*) were configured by us to mimic a Multi-Level Security (MLS) system from [32]. Details of representative configs are shown alongside. We used the 3 configs D9-* (from Ver. 0.9 of Muen) as seeded faults to test our tool. Ver. 0.9 of Muen generates *implicit* shared memory components, and this undeclared sharing was correctly flagged by our tool. The average running time on a configuration was 5.6s. The experiments were carried out on an Intel Core i5 machine with 4GB RAM running Ubuntu 16.04.

System	Sub	CPU	PMem (MB)	Image (MB)	Time (s)	Check Passed
D7_Bochs	8	4	527.4	13.8	3.7	✓
DL_conf1	8	4	506.5	12.9	3.7	✓
DL_conf2	9	4	1552.7	15.1	6.8	✓
DL_conf3	12	4	1050.1	23.3	6.7	✓
DL_conf4	16	4	1571.4	15.1	9.2	✓
D9_Bochs	10	2	532.9	16.2	4.9	✗
D9_vtd	16	4	1057.8	18.4	5.9	✗
D9_IntelNuc	10	2	567.0	16.2	5.5	✗

Discussion. We believe that the property we have proved for Muen (namely conformance to an abstract specification via a refinement proof) is the canonical security property needed of a separation kernel. However security standards often require specific basic security properties to be satisfied. In [15] we discuss how some of these properties mentioned in [11,16] follow from our proof.

The validity of the verification proof carried out in this work depends on several assumptions we have made. Apart from implicit assumptions like page table translation and VMX instructions behave the way we have modelled them, we made explicit assumptions like the 64-bit TSC counter does not overflow (it would take *years* to happen), and a minor frame length is never more than 2^{32}

ticks. If any of these assumptions are violated, the proof will not go through, and we would have counter-examples to conformance with the abstract specification.

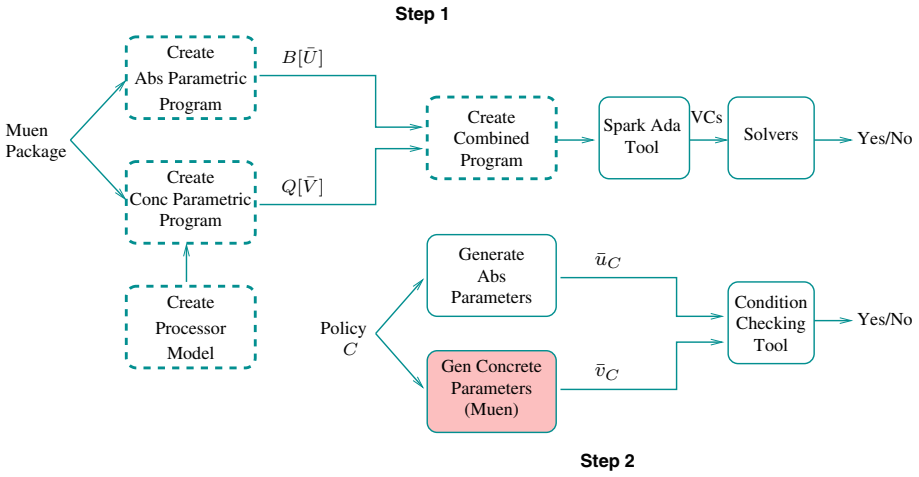


Fig. 7. Components in Muen verification. Untrusted components are shown shaded while non-automated (manual) steps are shown with dashed boxes.

Finally, we show the various components used in our verification in Fig. 7. Each box represents a automated tool (full boxes) or manual transformation carried out (dashed boxes). Components that we trust in the proof are unshaded, while untrusted components are shown shaded.

We would like to mention that the developers of Muen were interested in adding our condition checking tool to the Muen distribution, as they felt it would strengthen the checks they carry out during the kernel generation. We have updated our tool to work on the latest version (v0.9) of Muen, and handed it over to the developers.

5 Related Work

We classify related work based on general OS verification, verification of separation kernels, and translation validation based techniques.

Operating System Verification. There has been a great deal of work in formal verification of operating system kernels in the last few decades. Klein [20] gives an excellent survey of the work till around 2000. In recent years the most comprehensive work on OS verification has been the work on seL4 [21], which gave a refinement-based proof of the functional correctness of a microkernel using the Isabelle/HOL theorem prover. They also carry out an impressive verification of page table translation [35]. The CertiKOS project [14] provides a technique for

proving contextual functional correctness across the implementation stack of a kernel, and also handles concurrency. Other recent efforts include verification of a type-safe OS [37], security invariants in ExpressOS [26], and the Hyperkernel project [29].

While verification of a general purpose OS is a more complex task than ours—in particular a general kernel has to deal with dynamic creation of processes while in our setting we have a *fixed* set of processes and a fixed schedule—the techniques used there cannot readily reason about generative kernels like Muen. We would also like to note here that while it is true in such verification one often needs to reason about parametric components (like a method that computes based on its parameters), the whole programs themselves are *not* parametric. In particular, a standard operating system is *not* parametric: it begins with a concrete initial state, unlike a parametric program in which the initial state has unitialized parameters. Thus the techniques developed in this paper are needed to reason about such programs. Finally, we point out that none of these works address the use of VT-x virtualization support.

Verification of Separation Kernels. There has been substantial work in formal verification of separation kernels/hypervisors. seL4 [21] can also be configured as a separation kernel, and the underlying proof of functional correctness was used to prove information flow enforcement. Heitmeyer et al. [16] proved data separation properties using a refinement-based approach for a special-purpose SK called ED, in an embedded setting. As far as we can make out these systems are not generative in nature, and either do not use or do not verify hardware virtualization support. Additionally, unlike our work, none of these works (including OS verification works) are *post-facto*: they are developed *along* with verification.

Dam et al. [10] verify a prototype SK called PROSPER, proving information flow security on the specification and showing a bisimulation between the specification and the implementation. PROSPER works for a minimal configuration with exactly two subjects, and is not a generative system. The Verisoft XT project [4] attempted to prove the correctness of Microsoft's Hyper-V hypervisor [23] and Sysgo's PikeOS, using VCC [7]. While the Hyper-V project was not completed, the PikeOS memory manager was proved correct in [5]. Sanan et al. [33] propose an approach towards verification of the XtratuM kernel [9] in Isabelle/HOL, but the verification was not completed.

Translation Validation Techniques. Our verification problem can also be viewed as translation validation problem, where the Muen generator translates the input policy specification to an SK system. The two kinds of approaches here aim to verify the generator code itself (for example the CompCert project [24]) which can be a challenging task in our much less structured, *post-facto* setting; or aim to verify the generated output for each specific instance [30]. Our work can be viewed as a via-media between these two approaches: we leverage the template-based nature of the generated system to verify the generator conditionally, and then check whether the generated parameter values satisfy our assumed conditions.

6 Conclusion

In this work we have proposed a technique to reason about *template*-based generative systems, and used it to carry out effective *post-facto* verification of the separation property of a complex, generative, virtualization-based separation kernel. In future work we plan to extend the scope of verification to address concurrency issues that we presently ignore in this work.

Acknowledgement. We thank the developers of Muen, Reto Buerki and Adrian-Ken Rueegsegger, for their painstaking efforts in helping us understand the Muen separation kernel. We also thank Arka Ghosh for his help in the proof of interrupt handling.

References

1. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. AdaCore: GNAT Pro Ada toolsuite (2018). https://www.adacore.com/gnatpro
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Baumann, C., Bormer, T.: Verifying the PikeOS microkernel: first results in the Verisoft XT avionics project. In: Doctoral Symposium on Systems Software Verification. p. 20 (2009)
5. Baumann, C., Bormer, T., Blasum, H., Tverdyshev, S.: Proving memory separation in a microkernel by code level verification. In: Proceedings of the 14th Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, pp. 25–32. IEEE (2011)
6. Buerki, R., Rueegsegger, A.K.: Muen - An x86/64 separation kernel for high assurance. Technical report, Univ. Applied Sc. Rapperswils (HSR) (2013). https://muen.codelabs.ch/
7. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
8. Conchon, S.: SMT techniques and their applications: from Alt-Ergo to Cubicle. Thèse d'habilitation, Université Paris-Sud (Dec 2012)
9. Crespo, A., Ripoll, I., Masmano, M.: Partitioned Embedded Architecture Based on Hypervisor: The XtratuM approach. In: European Dependable Computing Conference (EDCC), Spain. pp. 67–72. IEEE (2010)
10. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple ARM-based separation kernel. In: Proeedings. ACM Conference on Computer and Communications Security, CCS 2013. pp. 223–234. ACM (2013)
11. Directorate, I.A.: U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Version 1.03 29 June 2007 (2007). https://www.niap-ccevs.org/Profile/Info.cfm?id=65
12. Divakaran, S., D'Souza, D., Sridhar, N.: Efficient refinement checking in VCC. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 21–36. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_2

13. Green Hills Software: INTEGRITY Multivisor (2019). https://www.ghs.com
14. Gu, R., et a: CertiKOS: an extensible architecture for building certified concurrent OS kernels. In: USENIX Symposium Operating Systems Design and Implementation (OSDI), pp. 653–669 (2016)
15. Haque, I., D'Souza, D., P, H., Kundu, A., Babu, G.: Verification of a generative separation kernel (2020). https://arxiv.org/abs/2001.10328v2
16. 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 the Conference on Computer and Communications Security (CCS), pp. 346–355. ACM (2006)
17. Hoare, C.A.R.: Proof of correctness of data representations. In: Bauer, F.L., et al. (eds.) Language Hierarchies and Interfaces. LNCS, vol. 46, pp. 183–193. Springer, Heidelberg (1976). https://doi.org/10.1007/3-540-07994-7_54
18. Hoare, C.A.R., et al.: Data Refinement Refined (Draft). Technical report, Oxford University (1985)
19. Intel Corp.: Intel 64 and IA-32 architectures software developer's manual, vol 3C (May 2018)
20. Klein, G.: Operating system verification: an overview. Sadhana **34**(1), 27–69 (2009)
21. Klein, G., et al.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**, 1–70 (2014). Article 2
22. Klein, G., et al.: seL4: formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, Big Sky. pp. 207–220. ACM (2009)
23. Leinenbach, D., Santen, T.: Verifying the Microsoft hyper-v hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_51
24. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Proceedings of the Principles of Program Languages (POPL), pp. 42–54. ACM (2006)
25. LynxSecure: LynxSecure Sep. Kernel Hypervisor (2019). http://www.lynx.com/
26. Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in ExpressOS. In: Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 293–304. ACM (2013)
27. Masmano, M., Ripoll, I., Crespo, A., Jean-Jacques, M.: XtratuM: A Hypervisor for Safety Critical Embedded Systems. In: Real Time Linux Workshop (2009)
28. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
29. Nelson, L., et al.: Hyperkernel: push-button verification of an OS kernel. In: Proceedings of the Symposium Operating Systems Principles (SOSP), pp. 252–269. ACM (2017)
30. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054170
31. Rushby, J.M.: Design and verification of secure systems. In: Proceedings of the Symposium Operating System Principles (SOSP), pp. 12–21. ACM (1981)
32. Rushby, J.M.: Proof of separability a verification technique for a class of security kernels. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 352–367. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_23

33. Sanan, D., Butterfield, A., Hinchey, M.: Separation kernel verification: the xtratum case study. In: Proceedings of the Verified Software: Theories, Tools and Exp. (VSTTE), pp. 133–149 (2014)
34. Sysgo AG: PikeOS 4.2 hypervisor (2018). https://www.sysgo.com/
35. Tuch, H., Klein, G.: Verifying the L4 virtual memory subsystem. In: Proceedings of the NICTA Formal Methods Workshop on Operating Systems Verification, pp. 73–97 (2004)
36. Wind River: VxWorks MILS Platform (2019). https://www.windriver.com
37. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: Proceedings of the Programming Language Design and Implementation (PLDI), pp. 99–110. ACM (2010)