Extending seL4 Integrity to the Genode OS Framework

Scott Constable
Syracuse University
sdconsta@syr.edu

Arash Sahebolamri
Syracuse University
asahebol@syr.edu

Steve Chapin
Syracuse University
chapin@syr.edu

Abstract

In 2011, the seL4 team formally verified that the seL4 microkernel enforces integrity and authority confinement on user-space threads. We extend these guarantees to virtual machine monitors (VMMs) running on the Genode OS Framework, and using seL4 as the underlying microkernel. We will do so by employing a model checking approach using Alloy, and combining the seL4 integrity proofs with our models of Genode and seL4. Our verification effort will show that VMMs cannot manipulate kernel objects for which they lack the necessary permissions, and that no VMM can escalate its privilege beyond what is allowed and interfere with other VMMs running in the system.

1 Introduction

Genode is described as an operating system framework. Like other operating systems, it provides an abstraction layer that applications use to access system resources like memory, CPU time, etc. Plus, Genode has a capability based access control system wherein accessing any resource by a component (application) in Genode requires the correct capability, and the components can be constrained in the capabilities they can obtain by a static policy provided at system creation time.

Unlike most other operating systems, Genode does not have a kernel of its own, or rather it is not bound to any specific kernel; instead it can be configured to run on a number of kernels. The Core component of Genode bridges that gap between the abstractions provided by the underlying kernel and the abstractions expected of Genode by components running in the system.

By having a capability based access control system, Genode strives to be a secure operating system. One aspect of security is *integrity*, the notion that components in a system cannot make changes to the system beyond what they are allowed to do. Genode's capability based security system attempts to provide that guarantee, but since Genode runs on multiple kernels, any guarantees of security it attempts to provide are contingent on the security and abstractions of the underlying kernel.

For example, while Linux is one of the kernels that Genode supports, Genode cannot enforce its capability based system when run on Linux. A malicious or compromised component can circumvent the Genode API and make system calls directly to the underlying kernel, coercing the system into taking actions that the component's set of capabilities wouldn't allow it to do. Additionally, Linux is a monolithic kernel with hundreds of thousands of lines of code, which means that even if its design matched the requirements of Genode for enforcing its access control system, such a massive TCB size would mean that presence of security-compromising bugs would be highly likely.

With this description, it is apparent that a suitable kernel for Genode would be one that is designed to be a minimalistic kernel, providing basic abstractions and leaving device drivers, file systems, etc. to user-space code.

Of the kernels supported by Genode that fit this description, seL4 is notable. The reason is that seL4 is a microkernel which has been fully and formally verified [5] [6]. The verification of seL4 guarantees that the C implementation of seL4 corresponds to an abstract specification written in a much higher level and more succinct language [7]. The verification of seL4 has been done in the Isabelle/HOL proof assistant. This means that the correspondence theorems have been fully proved (as opposed to being automatically checked in a subset of the space state, as is customary of model checking approaches), and that the proofs are computer-checked and therefore free of human error. In addition to proofs of correspondence, seL4 has been proven to preserve integrity.

The proofs of seL4 integrity can be the basis for Verifying that Genode, when combined with seL4 as its underlying kernel, provides the security related integrity properties that are desired of it.

In this work, we provide such verification, not on the Genode source code directly, since Genode is written in C++ and verifying C++ code is beyond the capabilities of our available tools. Rather, we model the behavior of Genode's most privileged component, Core, with the underlying kernel in the Alloy modeling language [4], and verify that it will not misuse its unbounded authority to compromise integrity. This, combined with the proofs of integrity for seL4 provide a high level of assurance about a Genode/seL4 system.

To model Core, we scrutinized the Genode source code [3], and extracted the essence of what the code does. We recognize that doing so does not provide an absolute guarantee that Core actually behaves according to our model. There are ways to remedy this, like using tools that show a C++ code base is likely free of security critical bugs. We leave that to future work.

Both Genode and seL4 are actively developed systems. Our work here is based on version 17.08 of Genode and seL4 version 6.0.

The rest of this paper is organized as follows. In section 2, we provide a background on seL4 and its integrity proofs, plus a brief introduction to Genode. In section 3 we discuss our approach to verifying security of Genode/seL4, which includes our assumptions about the system setup, plus the properties that we aim to verify. Section 4 discusses the formal models of seL4 and Genode that we build in Alloy; and section 5 introduces our approach to verifying security properties using Alloy, including properties that go beyond seL4's definitions of integrity and authority confinement.

2 Background

2.1 seL4 Integrity

In 2011, the seL4 team proved that the seL4 microkernel enforces integrity and authority confinement [8]. In this context, integrity means that no principal (thread) can modify any kernel object for which it lacks the necessary permission. Authority confinement requires that no principal can exceed its delegated authority, as determined by a user-specified access control policy.

The seL4 integrity and authority confinement theorems are both expressed in terms of a subjective abstract predicate, `pas_refined`, which holds when a given kernel state satisfies a given access control policy. The `pas_refined` predicate is subjective in that its validity varies from subject to subject, depending on that subject's maximum allowed privilege under the given access control policy. These theorems can be used by platform designers to verify whether or not seL4 can guarantee integrity and authority confinement for a given system and access control policy. But there are substantial limitations.

One conjunct of the `pas_refined` predicate is a condition referred to as "wellformedness" of a subject with respect to the policy. The wellformedness constraint specifies that within the bounds of the policy, the subject cannot exert arbitrary control over some other subject or subjects (e.g. by granting and revoking capabilities to virtual memory). Hence, more privileged subjects cannot be wellformed, and thus the integrity and authority confinement theorems cannot be directly applied to any such subjects. The paper refers to these subjects as "trusted" subjects, and lesser-privileged subjects are untrusted.

The paper suggests that the integrity and authority confinement theorems can still be applied to a system consisting of both trusted and untrusted subjects, in the following manner. Consider a (potentially infinite) execution

$$s_0 \xrightarrow{ut} s_1 \xrightarrow{ut} s_2 \xrightarrow{t} s_3 \xrightarrow{ut} s_4 \xrightarrow{ut} \cdots$$

where s_0 is the initial kernel state, s_k for $k \geq 1$ are the kernel states after each successive call to the kernel, ut is an untrusted subject, and t is a trusted subject. Assuming that the system begins in a state which satisfies `pas_refined` for ut, authority confinement and integrity are guaranteed by seL4 for ut on s_1 and s_2. However, since t may arbitrarily elevate ut's authority, or perhaps modify a kernel object accessible by ut, the same cannot be guaranteed for ut on s_3. The problem is not intractable. Rather, it requires some external reasoning about t's behavior, specifically that t preserves `pas_refined` for ut.

2.2 The Genode OS Framework

The Genode Operating System Framework[1] was not specifically designed to play the role of a hypervisor, although it can be configured for this purpose.

Genode provides a hierarchical architecture of components, where each component lives inside its own protection domain, consisting of its address space and capability space. Genode's goal is to guarantee that components cannot break out of the confinements of their protection domains, meaning that they cannot access memory outside their address spaces, and cannot perform operations without the necessary capabilities. This is of course subject to the strength of access control mechanisms in the underlying kernel. The seL4 microkernel, having a capability based access control mechanism itself, is more than capable of providing the requisite underlying access control system for Genode's protection domain guarantees and capability based access control [2], as discussed in sections 1 and 2.1.

Genode's Core component is in charge of providing operating system level services to other components. The services include memory, CPU time, and interrupt handling. Core provides these services by talking directly to the underlying kernel, seL4 for our purposes. Any OS service requested by a component opens a session to an RPC object living inside Core, and requests to that service are satisfied by Core making system calls to seL4. The RPC sessions themselves are powered by seL4's endpoints, the IPC mechanism provided by seL4.

In Genode, granting capabilities to a component is at the discretion of its parent component. The Init component is the special component in charge of bootstrapping the system. An XML configuration file specifies what components Init is supposed to run, and what services each component can and cannot access. This XML configuration file can limit the reach of each child component, which in our case would be the VMMs. A reasonable configuration should easily limit what resources the VMMs can access.

3 Approach

By using seL4 as the underlying microkernel, we extend seL4's integrity and authority confinement guarantees to an arbitrary number of virtual machines running on separate instances of VirtualBox, each in its own Genode protection domain.

3.1 System Setup

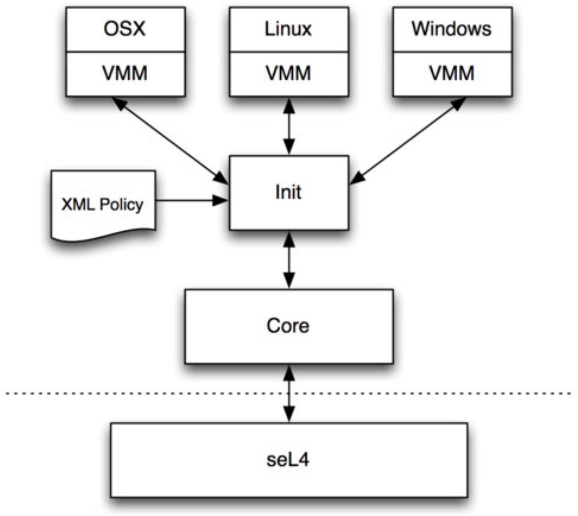

Figure 1: From bottom to top: the seL4 microkernel, the Core and Init components of Genode, and an arbitrary number of VMMs, each hosting its own guest OS. The horizontal dashed line represents the boundary between user-space (above) and kernel-space (below).

An overview of our system is given in Figure 1. The dashed line represents the boundary between user-space memory (above) and kernel-space memory (below). Core is the most privileged component in Genode, and is ultimately responsible for making calls into seL4. Init is less substantial, but is important to our effort because only it can distinguish between requests originating from each user-space VMM. This is because any component in Genode can only distinguish between its immediate children. For instance, if component *A* has child *B*, and child *B* has children *C* and *D*, then from A's point of view, a request originating from *D* or *C* is indistinguishable from a request made by *B*. Any request for a service from *C* or *D* is made to their parent *B*, and if *B* itself or another child of *B* is not capable of satisfying the request, *B* can choose to make the request on behalf of its children to *A*. Any hypothetical additional information on the origin of the request would be provided at the discretion of *B*, therefore trust in the validity of such additional information would be predicated on the trustworthiness of *B*.

Init is both Core's child and the parent of all the VMMs. Consequently if we are to maintain any kind of access control policy which enforces separation between VMMs, Init must be the component to enforce the policy. Thankfully, Genode allows the behavior of each component to be configured using an XML file. For instance we can configure Init to deny memory sharing across VMMs.

Since Init has the power to delegate and revoke capabilities to/from each child VMM, Init must be treated as a trusted component. Since Core is necessarily more privileged than Init, and is the sole component in charge of communicating with the seL4 microkernel, Core must also be trusted. The VMMs (and of course their guest OS's) will be our untrusted subjects. So we have three categories of subjects: Core, of which there is one; Init, of which there is one; and the VMMs, of which there are many. We focus on verifying the behavior of the Core component in this paper. While Init could potentially misbehave, its simplicity for now justifies our focus on Core. What's more, Init, like other non-Core components should not have any access to kernel capabilities (that is something that our verification shows), and therefore, verifying its behavior will be a much simpler task that won't require involving kernel operations (except the IPC calls).

Our approach has several advantages. The most appealing aspect is that our approach minimizes the reasoning we have to on the VMMs, which are very complex. Another advantage is that we are making direct use of the substantial body of work done (e.g. 350,000+ lines of proof) by the seL4 team.

3.2 Verification Goals

With respect to the seL4 integrity and authority confinement theorems, we have set this goal:

Establish that given a configuration in which components are only allowed to communicate with Core, components won't interfere with each other. In the context of seL4's theorems, that means establishing that authority confinement is an invariant of the system, and that integrity is preserved across all kernel transitions.

To achieve this goal, we use a model checking approach. The reason is that we needed to iterate quickly on our model of Genode, and updating proofs for an underlying model that is constantly changing is not practical. We use the Alloy model checker [4]. Alloy supports writing definitions in a language similar to First Order Logic, which has enough expressive power for us to model Genode and seL4 operations.

Central to our verification effort is the PAS record (Policy, Abstraction, Subject) [8]. This record describes the authority each component has over every other component in the system, and also associates kernel objects with components.

We formally represent our goals as two theorems that are similar to those proved in [8], except we show that Core preserves the integrity and authority confinement invariants for all untrusted components, and our theorems are on a combination of Genode's Core and seL4 kernel states rather than the seL4 kernel state alone.

Theorem 1 (Trusted Component Integrity)**.** *The* integrity *property holds for all untrusted threads after any kernel call made by any trusted thread t, assuming the state refines the access control policy, the system invariants* invs *and* ct-active *(the current thread is active) for non-interrupt events, t is the current thread, and st is the system state when the kernel call is invoked. In Isabelle:*

$$
\begin{aligned}
&\forall pas \in P.\ \forall t \in T.\\
&\{\textit{g-invs} \cap\\
&\quad \textsf{pas-refined}\ pas\ \circ\ \textit{seL4-state} \cap \textsf{invs}\ \circ\ \textit{seL4-state} \cap\\
&\quad (\lambda s.\ ev \neq \textsf{Interrupt} \longrightarrow \textsf{ct-active}\ s)\ \circ\ \textit{seL4-state} \cap\\
&\quad (\lambda s.\ t = \textsf{cur-thread}\ s)\ \circ\ \textit{seL4-state} \cap\\
&\quad (\lambda s.\ s = st) \circ\ \textit{seL4-state}\ \}\\
&\textsf{call-kernel}\ ev\\
&\{\lambda\text{-}.\ \textsf{integrity}\ pas\ st \circ\ \textit{seL4-state}\ \}
\end{aligned}
$$

Theorem 2 (Trusted Component Authority Confinement)**.** *The* pas-refined *property holds for all untrusted threads after any kernel call made by any trusted thread t, assuming the state refines the access control policy, the system invariants* invs *and* ct-active *(the current thread is active) for non-interrupt events, and t is the current thread. In Isabelle:*

$$
\begin{aligned}
&\forall pas \in P.\ \forall t \in T.\\
&\{\textit{g-invs} \cap\\
&\quad \textsf{pas-refined}\ pas\ \circ\ \textit{seL4-state} \cap \textsf{invs}\ \circ\ \textit{seL4-state} \cap\\
&\quad (\lambda s.\ ev \neq \textsf{Interrupt} \longrightarrow \textsf{ct-active}\ s)\ \circ\ \textit{seL4-state} \cap\\
&\quad (\lambda s.\ t = \textsf{cur-thread}\ s)\ \circ\ \textit{seL4-state} \cap\\
&\quad (\lambda s.\ s = st) \circ\ \textit{seL4-state}\ \}\\
&\textsf{call-kernel}\ ev\\
&\{\lambda\text{-}.\ \textsf{pas-refined}\ pas \circ\ \textit{seL4-state}\ \}
\end{aligned}
$$

Theorems 1 and 2, together with their analagous theorems from the seL4 integrity paper[8], show that both trusted and untrusted threads preserve integrity and authority confinement for all untrusted threads in our Genode deployment.

As mentioned in Sections 4.2 and 4.3 of the seL4 Integrity paper, the original theorems for untrusted components are quite limited: they can only be applied to components which are delegated very little privilege. To be more precise, components which either grant or receive capabilities from any other component must be trusted, and therefore are not covered by these theorems. We would like the VMMs to fall into the untrusted category, so that we will not have to perform any further verification on them. We can force this requirement by mandating that the VMMs cannot request further capabilities after they have already been initialized. This setup will be less flexible, but much easier to verify.

4 Formal Models

For establishing the theorems stated in section 3.2, as stated earlier, we opted for a model checking approach using the Alloy model checker. Choosing the model checker admits rapid prototyping and visualization of our models of Genode. There is one more reason that compelled us to employ a model checker. Proofs written in a proof assistant are susceptible to being rendered invalid if an error is later found in the models on which they are based; and the fact that we needed to go through multiple iterations of modeling to gain confidence in our Genode model meant that using a proof assistant was not very practical.

4.1 The seL4 Model

Employing the Alloy model checker to verify Genode/seL4 required us to have a model of seL4 in Alloy. We carefully examined the seL4 abstract specification, and translated it into the Alloy language. Our translation effort involved translating the definitions of seL4's state, seL4's kernel calls, and the definitions of integrity and authority confinement into Alloy.

Alloy's language is based on First Order Logic, whereas the seL4 specification and proofs are written in Isabelle/HOL, with HOL standing for Higher Order Logic. While in theory there can be predicates in HOL that are not expressible in FOL, we did not encounter any such definitions in translating the seL4 spec into Alloy (the important difference between the two is higher-order quantification, and Alloy allows using higher-order quantification in limited situations, which we take advantage of).

The bigger issue with such translation is the problem of state space explosion. To partially work around this issue and for other reasons, we made some simplifications and under-approximations of note in translating the specification. We can categorize these deviations from the seL4 spec as follows:

1. Simplifications that were informed by seL4's invariants proofs

   As part of the seL4's verification, numerous invariants have been proved about the seL4's state mutations by kernel operations. We took advantage of these invariants to simplify seL4's model in Alloy where it made sense. As an example of this simplification, we point to kheap and cur-thread, elements of the state record in the seL4 Isabelle/HOL specification.

   ```
   record abstract_state =
    kheap::"32 word=> kernel_object option"
    cur_thread :: "32 word"
    ...
   ```

   `kheap` is a partial function from memory addresses to kernel objects in the specification. It is coupled with multiple pointers or references to these objects in other elements of the seL4 state definition, `cur_thread` among them. There are invariants in the seL4 proofs that state these references are type-safe, meaning they do point to objects of the correct type. `valid_pspace` is an example of such an invariant. It maintains that all pointers that kernel objects have to other kernel objects are type-safe. `cur_tcb` is another invariant asserting that `cur_thread` references a valid TCB.

   By taking advantage of these invariants, we simplified our Alloy model by removing `kheap`, and having the state elements that point to kernel memory in the Isabelle/HOL model reference kernel objects directly in the Alloy model.

   ```
   sig KernelState {
     cur_thread : one TCB,
     ...
   }
   ```

2. Under-approximations

   Since we are concerned with the integrity aspect of seL4, and not its functional correctness in general, we found it unnecessary to exactly replicate the seL4's behavior in Alloy according to its specification. Any model that is an under-approximation of the seL4 spec (meaning the model allows a superset of the actual seL4 behavior) and still allows us to show that integrity and authority confinement are preserved by the actions of the Core component in a Genode/seL4 configuration would be acceptable to us.

   Our definitions of system calls in Alloy are under-approximations of their counterparts in seL4 spec. We aimed to capture just enough of their behavior to allow constraining the Genode's Core operations to show our desired properties about them.

3. Deviations to accommodate the Genode model

   We had to take some liberties when defining seL4 operations to accommodate the Genode model. The best example of such liberties is `seL4_unmap_multiple`.

   ```
   pred seL4_page_unmap_multiple
     [s,s': KernelState,
      pageCapCptrs: set CNodeIndex]{...}
   ```

   The actual seL4 page unmap operation takes *one* capability pointer and unmaps the page capability pointed to by the pointer from its page table. But Genode's detach operation in general flushes multiple data pages from multiple page tables (the former due to the fact that a dataspace in Genode can span multiple physical pages, and the latter due to the fact that region maps can be nested in Genode). Modeling the detach operation as a sequence of calls to seL4-page-unmap would waste Alloy atoms, and make model checking more expensive. For these reasons, we opted to have a version of the unmap operation that acts on multiple page capabilities at once, which we believed was a reasonable compromise between faithful modeling and tractability of model checking.

4. Other subjective simplifications

   There are more instances of simplifications that we made that are not strictly under-approximations of the Isabelle/HOL spec, nor are they explicitly supported by the seL4 invariants. However, we found them necessary to keep the size of the Alloy model manageable and the model checking possible. We were careful that making these simplifications would not alter the protection state of any components in a Genode/seL4 system. Virtual memory is a good example of the kinds of simplifications that we made. In the Isabelle/HOL specification, Virtual memory representation closely mirrors the underlying hardware, with 3 levels of page table-like objects (PageMapL4, PageDirectory, and PageTable), each represented as a mapping from addresses to page table entry-like objects, which in turn point to the lower level page table objects or frame objects. We realized that bringing this level of detail to the Alloy model would not alter anything about the protection state, on the other hand, it would increase the model size and make the state-space unreasonably large, to the point where model-checking would be impossible. Instead, we chose a simpler representation of virtual memory.

```
sig KernelState {
  ...
  pd_map : PageDirectory ->
    PageCap -> set AccessRight
}
```

This simplified representation maps PageDirectories (as a stand-in for all levels of page table-like objects) indirectly to frame objects. We believe this captures all that is needed for our purposes of verifying integrity.

4.2 The Genode Model

From previous discussions we know that the Core and Init components of a Genode system are considered trusted, and other components are untrusted. And from the seL4 integrity proofs we know that to establish the security of such a system, we need to show that the trusted components of the system preserve security properties, which means showing Core and Init components preserve integrity. Init has only a limited role in initializing the system. Core on the other hand, has complete authority to all the resources of a system, and its correctness is crucial to the security of a Genode/seL4 system. So we need to model the behavior of Core and its internal workings. This means that for integrity related concerns, there are two relevant pieces of state: the seL4 kernel state, and the internal state of Core. Thus, our model of an seL4/Genode state is made up of two parts: the seL4 state, and Core's internal state.

```
sig GenodeSeL4State{
 seL4_state : KernelState
 // Core internal state:
 ...
}
```

In addition to modeling the state of a Genode/seL4 system, we need to model the services/operations that the Core component provides. To do so, we consult the Genode book and Genode's source code.

Specifying Core's behavior means specifying the behavior of its threads. We focus on two such threads that are relevant to integrity concerns here: the page fault handler thread, and the entrypoint thread.

The page fault handler thread is invoked whenever a thread belonging to an untrusted component causes a page fault. The page fault handler thread is responsible for finding what pages should be mapped into the faulting thread's address space at the fault virtual address, finding and copying the relevant page capabilities and mapping the capabilities to the page table representing the faulting thread's address space. Predicate `page_fault_handler_thread_behavior` defines this thread's behavior. This is a predicate on Genode/seL4 state. It is worth noting that consistent with the Genode implementation and our assumptions, this predicate (and others defining Core's behavior) manipulates the Genode state directly, and the seL4 state only through making system calls.

The entrypoint thread plays the role of a dispatcher for Core's services; it listens on an endpoint for incoming requests to Core services, checks the transferred badge to resolve the service responsible for handling the request, and invokes the service. Predicates `core_Entrypoint_thread_behavior` and `core_EntryPoint_handle_request` define the behavior of this thread. The latter predicate forwards each requested operation to the predicate defined for handling that operation.

In addition to definitions for these thread behaviors, we modeled Services provided by Core. Services related to virtual memory management made up the bulk of our effort, as they are crucial to an operating system's security, and Genode has a sophisticated system for handling virtual memory. Genode's virtual memory management includes the notions of region maps, which represent address spaces, and data spaces, which represent chunks of physical memory that can be mapped into region maps. Additionally, region maps can act as data spaces (what Genode calls managed data spaces) and be mapped into other region maps, creating hierarchical virtual memory structures with arbitrary depths.

These intricate abstractions call for careful examination and modeling, and are worth the effort undertaken to ensure the correctness of their implementations on top of seL4.

We examined the Genode source code, and extracted a model representing its implementation of virtual memory and its concepts of region maps and data spaces on top of seL4. Our model includes a predicate for the page fault handler routine, plus predicates representing operations on region maps: attach and detach.

Besides VM related operations, we modelled these Core services as well:

- the Genode Protection Domain RPC object, which is responsible for the management of a component
- the thread management service of Core
- the services and RPC objects related to Genode's signaling mechanism, which includes SignalSource and SignalContext RPC objects, and a number of operations on these objects plus the Protection Domain RPC object.

Since in our setting only the behavior of Core is known, such a system can evolve in two ways: (1) by a thread belonging to Core making system calls or updating Core's internal state, and (2) by any other thread making any system call with arbitrary parameters. By allowing the state to evolve in these 2 ways, we make no assumptions about the behavior of untrusted components. What's more, we do not presuppose enforcement of Genode's capability system. Rather, our model recognizes that Genode's access control mechanism is built on the seL4's capability system, and its enforcement is contingent upon the way it is implemented and the guarantees provided by seL4. In fact, this means that we end up not needing to model Genode Caps as a separate entity at all, since they are no more than wrappers around seL4 end point caps, and an untrusted component is free to make any system calls, including invoking endpoint caps to which it has access with arbitrary parameters.

5 Verification

Employing Alloy to check our desired integrity and authority confinement properties about Genode/seL4 required translating seL4's integrity related definitions into Alloy, which was a straightforward task. We translated the definitions of authority confinement and integrity into Alloy, and to make sure that our translation was correct, had Alloy check properties about these definitions proven by the seL4 team: the reflexivity and transitivity of the integrity predicate.

We also needed to constrain state evolution based on our assumptions that Core's behavior is known and any other thread's behavior is unknown.

```
pred evolve
 [s,s' : GenodeSeL4State]{
 {
  s.seL4_state.cur_thread in
  tcb_of_thread[pageFaultHandler, s] =>
   page_fault_handler_thread_behavior[s,s']
  else s.seL4_state.cur_thread in
  tcb_of_thread[coreEntryPoint, s] =>
   core_Entrypoint_thread_behavior[s,s']
  else // non-Core thread
  {
   chaosMonkey[s.seL4_state,s'.seL4_state]
   g_differ_in[SeL4_state, s , s']
  }
 } or
 {
  switch_threads[s.seL4_state,s'.seL4_state]
  g_differ_in[SeL4_state, s, s']
 }
}
```

Based on the evolve predicate the state can evolve in 3 ways: (1) by randomly switching threads, (2) if the current thread is a thread belonging to Core, according to the thread's well-defined behavior, and (3) if the thread does not belong to Core, by making arbitrary seL4 system calls (the `chaosMonkey` predicate allows any system call with arbitrary parameters to be invoked).

Before attempting to verify integrity and authority confinement, we need to define a set of invariants for our system. The invariants constrain the state space in which we verify authority confinement and integrity; and without which our security properties would not hold.

For example, to make sure that all the threads in the system belong to a protection domain that Core knows about, we have the `g_invariants_all_tcbs_accounted_for` predicate. To make sure that all SignalSource objects in Core point to seL4 notification capabilities (and not capabilities to other objects), we have the `g_invariants_signalSource_objs_valid` predicate.

All such constraints are combined together in the `g_invariants` predicate. To make use of this predicate as a precondition for verifying integrity and authority confinement, we of course have to verify that it is actually an invariant of the system.

```
check genode_invariants{
 all s,s': GenodeSeL4State {
  g_invariants[s] =>
  evolve[s,s'] =>
  g_invariants[s']
 }
}
```

In addition to being a requirement for verifying security properties, these invariants provide insight into the behavior of Core, and verifying them in and of itself is a useful check that Core behaves as intended.

As stated in section 3.2, the authority confinement and integrity predicates revolve around a PAS record, which includes a policy, an abstraction function mapping kernel objects to labels (components), and a subject (the currently running component).

Defining policy is relatively straightforward, as we are assuming that untrusted VMMs are running in separate protection domains of Genode, and our goal it to establish that these VMMs cannot interfere with each other. This assumption informs the policy that we need to supply to the integrity and authority confinement predicates. We assume that in addition to Core, there are two untrusted components running in the system (checking with more components is possible, but won't add more insight into Genode's behavior) ; and our policy allows Core to have Control authority over untrusted components, and the untrusted components to only send messages to Core.

Defining the abstraction function is more challenging. As mentioned, the abstraction function maps kernel objects to labels. The component based isolation mechanism in Genode is a good candidate for deciding labels of kernel objects: a thread belonging to comp1 is mapped to comp1 by the abstraction function, a page in comp2's address space is mapped to comp2, and so on.

This way of creating abstraction functions raises a subtle point: The abstraction function is state-dependent. On the face of it, this fact can be worrying; we look at the authorities in one state, declare those to be how things should be, and check that by one step of state evolution, those authorities are preserved. But in fact, we do not look at the authorities in the seL4 portion of the state, rather, we look at Core's view of the system to extract the abstraction function. Doing so avoids the circular argument above, and has the effect of verifying that the actual authorities in the system are consistent with Core's view of how things should be.

Since we are not appealing to the integrity and authority confinement proofs for verifying the Core behavior, the subject part of PAS is required only for asserting integrity (whether a state change is allowed depends on who is doing it), and is unambiguously determined.

With the policy, abstraction function, and subject definitions in place, and following seL4's authority confinement and integrity theorems, these are the assertions we expect to be true:

```
check auth_confined{
 all s, s': GenodeSeL4State{
  let abs =  core_view_abs[s] |
  g_invariants[s] =>
  pa_refined[policy, abs, s .seL4_state] =>
  evolve[s, s'] =>
  pa_refined[policy, abs, s'.seL4_state]
 }
}
```

This property asserts that given general Genode invariants as a precondition, authority confinement is an invariant of the system.

```
check integrity{
 all s, s': GenodeSeL4State{
  let abs =  core_view_abs[s],
  subject = abs[s.seL4_state.cur_thread] |
  g_invariants[s] =>
  pa_refined[policy, abs, s .seL4_state] =>
  evolve[s, s'] =>
  integrity[policy, abs, subject,
            s.seL4_state, s'.seL4_state]
 }
}
```

This property asserts that given Genode invariants as a precondition, integrity is preserved with all state changes.

It is worth noting that these assertions verify integrity and authority confinement properties when both the trusted Core threads are running, and when the untrusted threads of other components are running. As presented here, the latter should not be necessary, since the seL4 theorems prove that given a suitable policy that is wellformed for an untrusted component, it cannot break authority confinement or integrity. However, including untrusted components in the assertions provides an extra level of assurance about the correctness of the model.

Comparing these assertions with the theorems presented earlier, we see that these are essentially the Alloy versions of the same theorems, with simplifications necessitated by translating the seL4 definitions into Alloy applied.

5.1 Verification in Presence of Badged Capabilities

While the properties verified here show that untrusted components are not able to influence each other directly, and that they are only able to send requests to Core, and are not able to manipulate the state of the system in any other way; these properties leave open the possibility that one component can gain capabilities intended for another component and interfere with the other component's operation. That is because the policy related definitions do not support the notion of *badges*.

In seL4, endpoint capabilities can be badged, where messages sent through badged capabilities are accompanied by the badge, allowing the receiver of the message to identify the sender by the badge it received. Genode uses this mechanism to distinguish clients of different services of Core: multiple clients with different Genode capabilities all send messages to the same endpoint, and the entrypoint thread in Core listening on the endpoint identifies the service for which each message was intended by looking up the badge and forwards the message to the service.

In seL4's definition of policy, `SynSend` is the authority that corresponds to components sending messages through endpoints. However, seL4 policies do not distinguish badged endpoint capabilities from unbadged ones or from endpoint capabilities with different badges. If component A has a capability with badge x to (an endpoint in) component R, and component B has a capability with badge y to R, the intention is for R to recognize which messages are from A and which ones are from B. But there is no way to represent this fact in the policy, as both A and B would have `SynSend` authority to R. Consequently, using seL4's definition of policy, there is no way to verify that component A won't have the authority intended for component B or vice versa.

This can be remedied by adding the notion of badges to seL4 policy related definitions. In seL4 proofs, the `Auth` type is the type that represents the kinds of authorities components can have to one another in a policy (and includes `SyncSend`). This type can be augmented to include badges for the `SyncSend` authority:

```
datatype auth =
  ... | SyncSend (badge option)|  ...
```

With this updated definition, `SyncSend (Some x)` would represent the case where a component is intended to be only allowed to send messages with badge x, and `SyncSend None` would represent the case where the component is not restricted in the badges its messages could have. In addition to `Auth`, the definitions of authority confinement (`PAS_refined`) and integrity would also need to be updated to account for badged `SyncSend` authorities. As expected, the updated definition of authority confinement wouldn't allow a component with authority `SyncSend (Some x)` to some other component to have an unbadged or differently badged capability to an endpoint in the target component.

While current seL4 integrity proofs do not support badges in policies, we suspect that including this notion would retain the validity of the integrity theorems; they would of course need to be re-proved with the updated definitions. In any case, with the updated definitions, our approach allows us to use Alloy to check whether integrity and authority confinement would hold for Genode, and have a reasonable level of confidence if the answer is yes.

Having badges in policies complicates the task of defining a suitable policy for our Genode configuration. Without badges, the policy could simply restrict components to only have Send and Receive authorities to Core, and allow Core to have Control authority to all components. But with badges, the Send authority of components to Core needs to be partitioned. We tackled this issue by defining upper and lower bounds for policies, and adding a predicate `policy_send_auths_to_core_disjoint` that asserts that the send authority between components is partitioned. With these definitions, we could verify that authority confinement holds for all policies within the policy bounds that satisfy `policy_send_auths_to_core_disjoint`.

```
check auth_confined_badged{
 all s, s': GenodeSeL4State,
     policy : (L -> Auth_badged -> set L)|
 {
  policy_in[policy, policy_upper_bound]
  policy_in[policy_lower_bound, policy]
  send_auths_to_core_disjoint[policy]
 } =>
 {
  let abs = core_view_abs[s] |
  g_invariants[s] =>
  pa_refined_badged[policy, abs,
                    s.seL4_state] =>
  evolve[s, s'] =>
  pa_refined_badged[policy, abs,
                    s'.seL4_state]
 }
}
```

This property asserts that for any policy within the policy lower and upper bounds in which the send authorities of components to Core are partitioned, authority confinement (with the updated definition) is an invariant of the system, given Genode invariants.

What this property shows is that components are incapable of obtaining any capabilities intended for other components of the system, and therefore cannot interfere with the operations of other components by forcing Core or the seL4 kernel to take action on their behalf.

In addition to authority confinement, we also could verify a version of integrity updated to support badged policies. We used the same technique for constraining the policies as we did with authority confinement.

```
check integrity_badged{
 all s, s': GenodeSeL4State,
     policy : (L -> Auth_badged -> set L)|
 {
  policy_in[policy, policy_upper_bound]
  policy_in[policy_lower_bound, policy]
  send_auths_to_core_disjoint[policy]
 } =>
 let abs = core_view_abs[s],
     subject = abs[s.seL4_state.cur_thread]|
 g_invariants[s] =>
 pa_refined_badged[policy, abs,
                   s.seL4_state] =>
 evolve[s, s'] =>
 integrity_badged[policy, abs, subject,
   s.seL4_state, s'.seL4_state]
}
```

`integrity_badged` asserts that for any policy within the policy lower and upper bounds with the send authorities a components to Core partitioned, given Genode invariants and authority confinement as preconditions, integrity is preserved with state changes.

The definition of integrity here accounts for badged send authorities, so if a component A has `Send (Some x)` authority to Core, integrity is only preserved when the message it sends to Core is badged with x.

With these properties we are able to verify that Genode's Core preserves badged versions of integrity and authority confinement, and to some extent verify that seL4 guarantees these properties for untrusted components (since `evolve` allows untrusted components to make any seL4 system calls).

6 Conclusion and Future Work

We took a hybrid approach to verifying the integrity of a Genode/seL4 system with a restrictive access control policy. We took the seL4 integrity and authority confinement proofs done in Isabelle and translated them into Alloy. We then modeled the behavior of Genode's security-critical component Core in Alloy, and used the model to verify that Genode's Core component preserves security properties integrity and authority confinement. That, combined with the seL4 proofs that untrusted and unprivileged components cannot break integrity and authority confinement bounds, indicate that a Genode/seL4 combination is suitable to security critical applications.

During our work, we realized that the seL4 integrity proofs need to be strengthened to account for badged send authorities. We updated the definitions related to authority and authority confinement to account for this fact, and used the updated definitions to verify that untrusted Genode components cannot obtain capabilities intended for other components. We discussed these extensions with the seL4 team at Data61, and our initial analysis is that the extensions are consistent with (and implied by) the original definitions in seL4, but that rigorous proofs of the integrity/authority confinement theorems with the new definitions of authority and policy would take substantial effort beyond our current resources, and are therefore left to future work.

Our models of Genode and seL4, plus the definitions of integrity and authority confinement, and the verification commands added up to about 3000 lines of Alloy code.

7 Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Additionally, this research was developed in cooperation with Critical Technologies, Incorporated, of Utica, NY.

References

[1] FESKE, N. Operating System Framework 17.05.

[2] FESKE, N. Porting Genode to seL4. `https://github.com/genodelabs/genode/blob/master/repos/base-sel4/doc/core.txt`, 2015. [Online; accessed 5/5/2017].

[3] GENODELABS. Genode repository. `https://github.com/genodelabs/genode`.

[4] JACKSON, D. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM) 11*, 2 (2002), 256–290.

[5] KLEIN, G., DERRIN, P., AND ELPHINSTONE, K. Experience report: sel4: formally verifying a high-performance microkernel. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 91–96.

[6] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 207–220.

[7] KLEIN, G., SEWELL, T., AND WINWOOD, S. Refinement in the formal verification of the sel4 microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, pp. 323–339.

[8] SEWELL, T., WINWOOD, S., GAMMIE, P., MURRAY, T., ANDRONICK, J., AND KLEIN, G. sel4 enforces integrity. In *Proceedings of the Second International Conference on Interactive Theorem Proving* (Berlin, Heidelberg, 2011), ITP'11, Springer-Verlag, pp. 325–340.