Formally Verified System Initialisation

Andrew Boyton[1,2], June Andronick[1,2], Callum Bannister[1,2], Matthew Fernandez[1,2], Xin Gao[1], David Greenaway[1,2], Gerwin Klein[1,2], Corey Lewis[1], and Thomas Sewell[1,2]

[1] NICTA, Sydney, Australia[⋆]
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia

{first-name.last-name}@nicta.com.au

Abstract. The safety and security of software systems depends on how they are initially configured. Manually writing program code that establishes such an initial configuration is a tedious and error-prone engineering process. In this paper we present an automatic and formally verified initialiser for component-based systems built on the general-purpose microkernel seL4. The construction principles of this tool apply to capability systems in general and the proof ideas are not specific to seL4. The initialiser takes a declarative formal description of the desired initialised state and uses seL4-provided services to create all necessary components, setup their communication channels, and distribute the required access rights. We provide a formal model of the initialiser and prove, in the theorem prover Isabelle/HOL, that the resulting state is the desired one. Our proof formally connects to the existing functional correctness proof of the seL4 microkernel. This tool does not only provide automation, but also unprecedented assurance for reaching a desired system state. In addition to the engineering advantages, this result is a key prerequisite for reasoning about system-wide security and safety properties.

Keywords: System Initialisation, seL4, Isabelle

1 Introduction

Verification and validation of embedded software systems usually concentrate on the operational running state of the system. For example, the recent proof of non-interference for the seL4 microkernel [7] assumes the presence of a system state that corresponds to a high-level access control and information flow policy. It then shows that all executions from this state satisfy the non-interference property. Clearly, this operational running state is the interesting case for such proofs, but the question remains how to satisfy the initial assumption that the system is in a well-known state that corresponds to some specific policy.

[⋆] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work was in part funded by AOARD grant #FA2386-11-1-4070

More generally, this is the question of system initialisation: how does one bring a system from an empty power-off or boot state into a well-defined desired configuration from which it can operate normally, and how does one prove that this state is reached? For traditional operating systems, initialisation is mostly a question of initialising devices, loading binary code images, and running a manually created start-up script. For system security, the access control protection state of the initialised system is obviously critical. Such protection states can be large and intricate. Security policy descriptions in SELinux systems for instance, can have over 100,000 access control rules. Such policy descriptions are coarse grained compared to capability-based systems that control access to individual kernel objects. While the system is already constrained by a given security policy during operation, the initial startup code usually runs with elevated access and possesses the power to violate the policy. Its purpose is to bring the system into a state that conforms to the policy and then to relinquish its own access. Manually writing such a program for a sizeable policy is a daunting engineering task. Formally verifying that it does so correctly is even less appealing.

The main contribution of this paper is to demonstrate a technique for automatic and formally verified initialisation of capability-based systems. In particular, we show (a) an automatic initialiser for systems based on the formally verified seL4 microkernel, (b) a formal Isabelle/HOL model of this initialiser and its interaction with the kernel, (c) a formal Isabelle/HOL proof that this model leads to correctly initialised system states, (d) a formal connection of the system description that the initialiser takes as input to existing security policy formalisations and proofs for seL4 [7, 12], and (e) a formal proof that the seL4 invocations used by the initialiser model are refined by the exisiting functional specification of seL4 [4].

The initialiser takes as input a declarative description of the desired protection state in capDL [6], a capability distribution language, and automatically brings the system from the boot state into an initialised state that conforms with this desired protection state. While capDL descriptions can be large and complex, they can be generated from higher-level descriptions of the system, for instance from a component setup for MILS-style security architectures [1].

Our initialiser theorem is crucial for instantiating the existing seL4 security theorems [7, 12] that show the kernel enforces isolation of components running on top. This isolation allow us to establish that such MILS architectures are enforced correctly by the microkernel [3], which in turn enables us to reason modularly about user-level applications in the system.

The initialiser model directly connects to the seL4 API specification, which the binary of the kernel is proven to implement correctly [4, 11]. This is to our knowledge the first proof of a user-level model that directly links to a formally verified kernel implementation. This shows that such full realistic kernel API formalisations are usable for application-level proofs. The initialiser model has been implemented by straightforward translation into C code. We plan to prove that this translation is a correct implementation in future work. The focus in this paper is on the correctness of the algorithm and its use of the seL4 API.

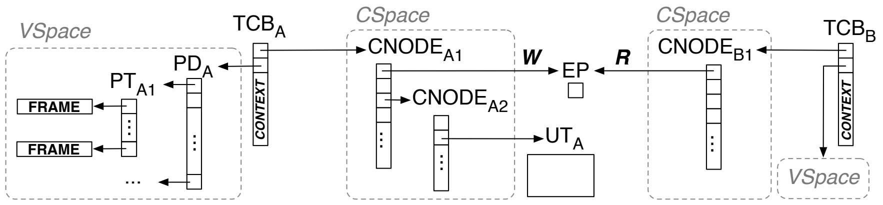

Fig. 1. seL4-based system with two threads that can communicate via an *endpoint*.

We begin the remainder of this paper in Sec 2 with a short overview of the seL4 kernel. Sec 3 describes the capDL [6] language and associated proofs. Sec 4 presents the formalisation of the initialiser itself, while Sec 5 summarises the correctness proof. Sec 6 discusses experience and limitations.

2 seL4

The seL4 microkernel is a general-purpose operating system (OS) kernel designed as a secure and reliable foundation for a wide variety of applications. An OS kernel is the only software running in the *privileged* mode of the processor. The seL4 microkernel is formally verified for full functional correctness to the binary level [4, 11]. This means that there exists a machine-checked proof that the C code and binary of seL4 are a correct refinement of its functional, abstract specification.

As a microkernel, seL4 provides a minimal number of OS services: threads, inter-process communication, virtual memory, and capability-based access control. Fig 1 shows a trivial example system on seL4, composed of two threads, a sender A and a receiver B, communicating via an *endpoint* EP. Each thread is represented by its *thread control block* (TCB), which stores its context, virtual address space (VSpace) and capability space (CSpace). A VSpace defines the memory accessible to the thread; it is represented by a set of frames, generally organised in a hierarchical, architecture-dependent structure of page tables and page directories. CSpaces are kernel managed storage for *capabilities.* A capability is an unforgeable token that confers authority and that is stored in a graph of capability nodes (CNodes). In seL4, when a thread invokes an operation on an object, it needs to provide an index into its CSpace pointing to a capability for that object with sufficient authority. For instance, sender A needs a *write* capability to the endpoint, while receiver B needs a *read* capability to the same endpoint.

The allocation of kernel objects in seL4 is performed by retyping *untyped memory*, an abstraction of a region of physical memory. Possession of a capability to untyped memory confers the authority to allocate kernel objects in this region: sender A can request the kernel to transform $\mathsf{UT_A}$ into, say, a new CNode.

At boot time, seL4 first pre-allocates memory for itself and then gives the remainder to the initial user task in the form of capabilities to untyped memory. This user task is the initialiser we are targeting in this paper. Its aim is first to use these untyped capabilities for creating the required objects, such as $\mathsf{TCB_A}$, $\mathsf{TCB_B}$, $\mathsf{CNode_{A1}}$, and then to initialise them appropriately, e.g. to set $\mathsf{TCB_A}$'s CSpace field to $\mathsf{CNode_{A1}}$. This includes setting up communication channels, e.g. storing the *write* capability to EP in $\mathsf{TCB_A}$'s CSpace.

3 CapDL

We formally specify the desired initial system configuration as a *capDL system description*. The aim of the capability distribution language capDL [6] is to unify all aspects of the protection state of the system as explicit capabilities, allowing us to describe complete access control system configurations by capability distributions alone.

A capDL system description is both the input of our initialiser and its target: our initial user task must terminate in an initialised state corresponding to the description given as input. In the example of Fig 1, the description would be a formal and complete enumeration of all the kernel objects and the capabilities between them.

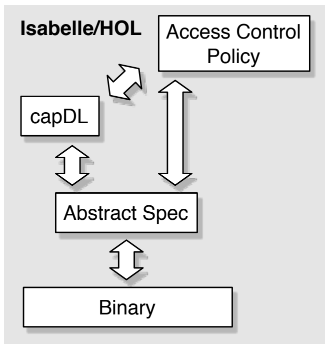

Fig. 2. CapDL model in the seL4 refinement chain (where arrows denote formal proof).

In addition to the language itself, which describes snapshots of system states, we have developed kernel semantics for this language that describes the effect of each kernel operation on such states, and showed that this *capDL kernel model* is a formally correct abstraction of existing models of seL4, with a complete refinement chain to the binary level, as shown in Fig 2. This ensures that the seL4 operations in the capDL model behave as the real kernel does.

We have also shown that capDL descriptions can be mapped to a corresponding access control policy: Let `s` be a kernel state from the abstract kernel specification level by Klein et al [4], `transform` be the state relation from abstract states to capDL states used in the refinement between these levels, `pas_refined P s` be the predicate that decides if `s` satisfies the access control policy `P` by Sewell et al [12], and `pcs_refined P c` be the predicate that decides whether a capDL state `c` satisfies the same policy `P`.

Theorem 1. *If the kernel invariants* `inv` *hold on* `s`*, then* `pas_refined P s = pcs_refined P (transform s)`*.*

The theorem implies that a capDL state description captures all information relevant to the protection state of an access control policy, i.e. instead of having to know the precise memory content of the machine, it is enough to reason about the information present in a capDL description.

```
init_system spec bootinfo obj_ids ≡
do (ut_cpts, free_cpts) ← parse_bootinfo bootinfo;
   (orig_caps, free_cpts) ← create_objs spec obj_ids ut_cpts free_cpts;
   dup_caps ← duplicate_caps spec orig_caps obj_ids free_cpts;
   init_vspace spec orig_caps obj_ids;
   init_tcbs spec orig_caps obj_ids;
   init_cspace spec orig_caps dup_caps obj_ids;
   start_threads spec dup_caps obj_ids
od
```

Fig. 3. The top-level definition of the system initialiser model.

4 System Initialisation

In this section, we present an overview of our formal model of the system initialiser in Isabelle/HOL, and examine in detail the initialisation of the capability spaces as a representative example. Recall that the initialiser is the first user task to run after boot time, with access to all available memory. We model it as a sequence of high-level instructions, taking a capDL specification `spec` as input, and creating and initialising all objects and capabilities as specified by `spec`. Formally, `spec` has the type `cdl_state`, i.e. a full state of the capDL kernel model. Its most important component is the kernel `heap` of type `obj_id ⇒ cdl_object`. CapDL objects are formalisations of the TCBs, CNodes, Endpoint, and other objects mentioned in the previous section. They consist of a map from capability slots to capabilities and potentially additional payload such as further TCB data.

The top-level definition `init_system`, shown in Fig 3, is purposely divided into well-defined separate phases which simplifies reasoning as we will see in Sec 5. The additional input `bootinfo` is given by the kernel to the initial user task and specifies the location of untyped memory and free capability slots in the initialiser's CSpace. The final parameter to the initialiser is the list of object names `obj_ids` mentioned in `spec`. The `do x← f; g x od` notation is syntactic sugar for the monadic binding `f >>= (λx. g x)` where `f` is executed, potentially changing the underlying state, with its return value passed into `g`, bound to the variable `x`.

In the first phase of the initialiser, we extract from `bootinfo` the list `ut_cpts` of pointers to the untyped memory regions the initialiser has access to and can use to create new objects, as well as the list of free slots `free_cpts` in its CSpace it can use to store capabilities to these new objects.

In the second phase, we create all objects listed in the capDL specification by invoking seL4's *retype* operation on the provided untyped memory. During this operation, the kernel will create a capability to each object and store it in the provided free slot in the initialiser's CSpace. This original capability can then be given to other threads, either by moving it or by copying it (with full or diminished rights). Note that an original capability confers more authority than derived ones; it allows the revocation of derived capabilities and full destruction of the object. This creates a subtle dependency for the order in which the initialiser has to distribute capabilities: it eventually needs to give away original capabilities, and at the same time keep access to the objects to finish their initialisation.

For this reason, we duplicate, in a third phase, all original capabilities *orig_caps* into *dup_caps*, also stored in the initialiser's CSpace.

At this stage we can start the initialisation, per object type, including installing the capabilities into the capability storage objects. VSpaces are initialised by mapping in the required entries in page directories, and then page tables; TCBs are each initialised atomically; CSpaces are initialised similarly to VSpaces with the added complication of needing to distinguish between capabilities that are *moved* and the ones that are *copied*; Moreover, unlike VSpaces which are fixed, two-level data structures, CSpaces can be arbitrary directed acyclic graphs.

The final step of the initialisation is to set all threads to be runnable, from which point the initialiser becomes dormant and the system is ready to run.

We describe the initialisation of CSpaces more deeply and use it as a running example in this paper. The initialisation of CSpaces consists of putting the desired capability in every slot of every CNode appearing in *spec*. This occurs in two phases, depending on whether *spec* requires the capability to be the original one or not. For all capabilities that need not be originals, we *copy* the initialiser's original capability into the target CNode (we actually *mint* it, diminishing the access rights to those specified in *spec*). For the ones that need to be original, we *move* the initialiser's original capability (we actually *mutate* it with the appropriate rights specified in *spec*, except for endpoint capabilities which cannot be mutated in seL4). Each phase maps over the full list of all CNode slots, but does nothing to slots not concerned with that phase.

```
init_cspace spec orig_caps dup_caps obj_ids ≡
do cnode_ids ← return [obj←obj_ids. cnode_at obj spec];
   mapM (init_cnode spec orig_caps dup_caps Copy) cnode_ids;
   mapM (init_cnode spec orig_caps dup_caps Move) cnode_ids
od
```

The *return* function just returns its argument without modifying the state; *[a←xs. P a]* denotes a filter returning all elements of the list *xs* for which *P a* holds; *mapM f xs* is the standard map function over state monads, executing a monadic function *f* on each element of the list *xs* in order.

In each phase, we initialise the capability slots one by one.

```
init_cnode spec orig_caps dup_caps mode cnode_id ≡
do cnode_slots ← return $ slots_of_list spec cnode_id;
   mapM (init_cnode_slot spec orig_caps dup_caps mode cnode_id) cnode_slots
od
```

The initialisation of a single capability slot *cnode_slot* of a CNode *cnode_id* is shown in Fig 4. This could for instance be the first slot of $\mathsf{CNode}_{\mathsf{A1}}$ in our example of Fig 1, which needs to contain a (not necessarily original) capability to the endpoint EP with a *write* right. In this definition, we first extract the target capability

```
init_cnode_slot spec orig_caps dup_caps mode cnode_id cnode_slot ≡
do target_cap ← assert_opt (opt_cap (cnode_id, cnode_slot) spec);
   target_cap_obj ← return (cap_object target_cap);
   target_cap_rights ← return (cap_rights target_cap);
   target_cap_data ← return (cap_data target_cap);
   is_orig_cap ← return (is_orig_cap spec (cnode_id, cnode_slot));
   dest_obj ← get_spec_object spec cnode_id;
   dest_size ← return (object_size_bits dest_obj);
   dest_root ← assert_opt (dup_caps cnode_id);
   dest_index ← return cnode_slot;
   dest_depth ← return dest_size;
   src_root ← return seL4_CapInitThreadCNode;
   src_index ← assert_opt (orig_caps target_cap_obj);
   src_depth ← return 0x20;
   if target_cap = NullCap then return True
   else if mode = Move ∧ is_orig_cap
       then if ep_related_cap target_cap
            then seL4_CNode_Move dest_root dest_index dest_depth
                  src_root src_index src_depth
            else seL4_CNode_Mutate dest_root dest_index dest_depth
                 src_root src_index src_depth target_cap_data
       else if mode = Copy ∧ ¬ is_orig_cap
           then seL4_CNode_Mint dest_root dest_index dest_depth
                 src_root src_index src_depth target_cap_rights
                 target_cap_data
           else return True
od
```

Fig. 4. Initialisation of a single capability slot.

target_cap that *spec* requires in *cnode_slot*. The function *opt_cap* returns an *option* type, i.e. either *Some cap* or *None*. The function *assert_opt* asserts that this value is of the form *Some cap* and returns *cap*; otherwise it fails. From *target_cap*, we extract the target object *target_cap_obj* (say EP), the desired rights *target_cap_rights* (say *write*), and additional data *target_cap_data* (e.g. for endpoints, a so-called *badge*). We store in *is_orig_cap* whether *spec* requires the capability to be original for that slot.

In order to be able to invoke seL4's *seL4_CNode_Move*, *seL4_CNode_Mutate* and *seL4_CNode_Mint* operations, the initialiser needs to hold, in its CSpace, both the target capability to be moved or copied, and a capability to the destination slot. We compute the destination information (*dest_root*, *dest_index* and *dest_depth*) from the (duplicated) capability that the initialiser holds for the destination slot. (We use the duplicate capabilities in case the original capability to the destination slot has already been given away.) We compute the source information (*src_root*, *src_index* and *src_depth*) from the (original) capability that the initialiser holds for the target capability. We then can invoke the appropriate seL4 operation depending if the target capability needs to be original or not. These operations directly connect to the capDL-level API model of the kernel.

5 Correctness

In this section, we summarise the proof of system initialiser correctness. We present the separation logic we instantiated for this proof, state the top-level theorem, show how the proof is decomposed, and describe how it is connected to the seL4 kernel proofs.

5.1 Separation Logic Instance

We begin by setting up the basic reasoning framework we use in the proof of the initialiser. As described in Sec 4, we initialise each object in isolation, and within an object, each capability slot in isolation. Ideally, the proof about these executions follows the same pattern. Separation logic [10] is a good fit for this style of reasoning, at least if the specific flavour of separation logic allows us to decompose heaps, and objects within heaps, across exactly these boundaries. In addition to the local state of the user-level initialiser, we will have to reason about the internal state of the kernel comprising the various objects we create. This means, if we want to use a specific style of separation logic, we will have to retro-fit it onto the existing capDL-level kernel model.

Usually, a separation logic is defined in terms of a heap and the concepts of disjointness and separating conjunction. We can shortcut this stack of definitions by building on an Isabelle type class for abstract separation algebra [5]. This previous development allows us to merely define the concepts of a heap, heap addition, and heap disjointness. After proving their basic axioms, we get the development of separation logic for free, including basic Isabelle/HOL automation.

As mentioned, we want the heap in our separation algebra to be fine-grained enough to split objects into individual capability slots and other object data. The heap in the existing kernel model had no such requirement and therefore has no such concept. Instead of enriching the state space of the existing model with partial object ownership as described in [5] (and then having to re-prove refinement to the code), we *lift* the existing state space into a larger one that allows us to perform the desired decomposition easily. Our lifted heap is of type

$$\mathit{sep\_state} = \mathit{obj\_id} \times \mathit{cdl\_component} \Rightarrow \mathit{sep\_entity\ option}$$

The *obj_id* is the same as in the heap mentioned in Sec 4. The *cdl_component* specifies which part of an object we are addressing — a single capability slot of an object, or the fields (non-capability data) of an object. A *sep_entity* is either a single capability, or an object with payload only. In the lifting from *obj_id* ⇒ *cdl_object* to *sep_state*, we also set the so-called *intent* field of TCBs to a default value. The *intent* models the contents of a thread's IPC buffer storing the parameters of the next kernel call it is about to make — a detail we can ignore, because the initialiser is the only thread that can make system calls.

In terms of separation algebra, the resulting *sep_state* is a standard heap structure and is instantiated in the standard manner [5]. The tradeoff is that we are not reasoning about the real state of the system, but about the lifted state. The lifted state space is larger than the original state space, and due to resetting the *intent* field, the mapping is not even injective. Neither effect impacts our verification: our predicates only talk about well-formed states that exist in reality and the states that are identified only differ in TCB intent which we can ignore.

To apply this lifting to the kernel state and phrase separation predicates about it, we use the syntax *<P>*. To apply separation predicates to the user-level initialiser state, which subsumes the kernel state, we use the syntax $\ll P \gg$.

Given this lifted heap and the automatic setup for separating conjunction $\wedge^*$ etc, we can define the classic maps-to predicates of separation logic. For instance, we define the predicate $ptr \mapsto_o object$ that specifies that the state consists of the object *object* at position *ptr*. We can divide it into two predicates that extract the fields and capability slots of an object separately:

$$ptr \mapsto_o object = (ptr \mapsto_f object \wedge^* ptr \mapsto_S object)$$

Following our heap structure, we can further divide the predicate $\mapsto_S$ on the capability storage of an object into predicates $\mapsto_c$ about its individual capability slots.

One key reasoning principle in separation logic is the *frame rule*. Because the kernel API model is a shallow Isabelle/HOL embedding, we cannot prove the frame rule as a generic rule of the logic. We can, however, bake-in the frame rule to any statement we make about the kernel API, proving it about the leaf functions and then passing it up through to the API top level. For instance, we proved the following Hoare triple for the kernel-internal operation *set_cap*, which bakes in the frame rule by adjoining $\wedge^*$ *R*. The rule states that *set_cap* changes only a single capability slot of a single object, and leaves everything else unchanged, including other parts or other capability slots of the same object.

$$\{\!|<ptr \mapsto_c old\_cap \wedge^* R>|\!\}\ set\_cap\ ptr\ cap\ \{\!|<ptr \mapsto_c cap \wedge^* R>|\!\}$$

Making sure that our flavour of separation logic matches the verification problem gives us a convenient tool for reasoning about loops — a common operation in the system initialiser. We can reduce a global map of an operation such as *set_cap* over a list of capability slots to local reasoning about each slot simply by using the following rule.

$$(\bigwedge R\ x.\ x \in xs \Longrightarrow \{\!|\ll P\ x \wedge^* R \gg|\!\}\ f\ x\ \{\!|\ll Q\ x \wedge^* R\gg|\!\}) \Longrightarrow$$
$$\{\!|\ll \bigwedge^* map\ P\ xs \wedge^* R\gg|\!\}\ mapM\ f\ xs\ \{\!|\ll \bigwedge^* map\ Q\ xs \wedge^* R\gg|\!\}$$

In this rule, the big $\bigwedge$ is universal quantification and the big $\bigwedge^*$ is separating conjunction over a list of predicates in the usual way.

In the following sections we will see how these predicates and rules are applied to make statements about the initialiser behaviour.

5.2 Top-Level Statement

The correctness statement for the system initialiser is that, at the end of the initialisation, all objects in the system either belong to the initialiser itself and are inactive, or are initialised in conformance with the capDL specification.

In capDL specifications, systems are described as a mapping from object identifiers to objects. An object, identified by *spec_object_id* in the capDL specification *spec*, is said to be initialised in conformance to *spec*, defined by the predicate *object_initialised spec* φ *spec_object_id*, if the object *spec_object* it points to in the resulting state is the one *spec* requires it to point to.

```
object_initialised spec φ spec_object_id ≡
λs. ∃kernel_object_id spec_object.
       φ spec_object_id = Some kernel_object_id ∧
       (kernel_object_id ↦ₒ spec2s φ spec_object) s ∧
       cdl_objects spec spec_object_id = Some spec_object
```

The injection φ captures the subtlety that the kernel decides memory addresses at runtime. It maps names in *spec* to these memory addresses. Therefore the predicate *object_initialised* requires that *spec_object_id* maps to a *kernel_object_id* via the injection φ and that *kernel_object_id* points to *spec_object* where all object identifiers have been renamed by φ — including those within the capabilities of each object, defined by *spec2s*. The function *cdl_objects spec* extracts the mapping from *obj_id* to object from *spec*.

The top-level theorem of the system initialiser is:

Theorem 2. *If* *well_formed spec* *and* *obj_ids = dom (cdl_objects spec)* *and* *distinct obj_ids* *then*

```
⦃≪valid_boot_info bootinfo spec ∧* R≫⦄
init_system spec bootinfo obj_ids
⦃λs. ∃φ. ≪⋀* map (object_initialised spec φ) obj_ids ∧*
            si_objects spec φ ∧* R≫ s ∧
          injective φ ∧ dom φ = obj_ids⦄
```

It states that, given a *well_formed* capDL specification, the system initialiser, if it terminates, transforms a boot state described by *boot_info*, into a state containing (a) each object in the specification correctly initialised and (b) the data structures of the initialiser.[3] Additionally, it states that the mapping φ is injective and covers all specification objects.

The assumptions about the capDL system description encoded in the predicate *well_formed* exclude infeasible specifications by describing constraints on seL4-based system configurations. The formal definition of these constraints is too long for this paper but are summarised as follows.

- There is only a finite number of objects in the system.
- Every object is of the correct size, with the correct number of capability slots.

[3] The initialiser presently does not delete the capabilities it duplicated.

– Every capability points to an object, and there is a capability in the system for every object. The types of the object and corresponding capability match.
– Each object only possess capabilities of the right type, e.g. page tables only store Frame capabilities, whereas CNodes can contain most capability types.
– Capability rights are well formed, e.g. frames cannot have write without read permissions.
– Each capability has a unique original capability that it is derived from.
– Page tables cannot be shared.
– Page tables must be empty, or mapped in a page directory.

There are further constraints in *well_formed* that encode current limitations of the initialiser, not fundamental constraints. We describe these in Sec 6.

5.3 Decomposition

The key to this proof is the ability to decompose it along the functionality of the initialiser. There are two aspects to this decomposition — decomposing the proof itself along function boundaries and decomposing predicates about objects such as *object_initialised* into smaller predicates. The former is provided by the frame rule, the latter by our heap structure.

The proof of the system initialiser is divided into three sections. The first part ensures that parsing the kernel provided *bootinfo* structure correctly extracts information about untyped memory and free capability slots in the boot state. The second part ensures that the *create_objs* function creates all objects described by the specification in their default state and stores the corresponding capabilities in the slots that later parts of the initialiser expect. This involves some internal book-keeping and looping over the collection of untyped capabilities.

In the last, most complex part of the proof, we show that each object is transformed from this default state (*object_empty*) to its fully initialised state (*object_initialised*). We further divide this last part into separate proofs about the initialisation of each type of object by showing the following rewrite rule.

```
⟦well_formed spec; obj_ids = dom (cdl_objects spec)⟧
⟹ ≪⋀* map P obj_ids ∧* R≫ =
    ≪⋀* map P [obj←obj_ids. table_at obj spec] ∧*
      ⋀* map P [obj←obj_ids. tcb_at obj spec] ∧*
      ⋀* map P [obj←obj_ids. cnode_at obj spec] ∧*
      ⋀* map P [obj←obj_ids. stateless_at obj spec] ∧* R≫
```

Expanding this map of an arbitrary predicate over all objects into maps by type allows us to use the frame rule for looking at each type in isolation. As an example, consider the rules in Fig 5 for *init_tcbs* and *init_cspace*. It is not important to understand these predicates in detail. However, we can note that each of them talk about a separate part of the overall object map, both mention some side conditions about the presence of capabilities in the initialiser itself (*si_cap_at* φ *caps spec obj_id* and *si_cspace*), and both have a frame condition *R* than can be suitably instantiated to join them up.

```
⟦well_formed spec; obj_ids = dom (cdl_objects spec); distinct obj_ids⟧
⟹ ⦃≪objects_empty spec φ [obj←obj_ids. tcb_at obj spec] ∧*
      ⋀* map (si_cap_at φ orig_caps spec) obj_ids ∧* si_cspace ∧* R≫⦄
    init_tcbs spec orig_caps obj_ids
    ⦃≪objects_initialised spec φ [obj←obj_ids. tcb_at obj spec] ∧*
      ⋀* map (si_cap_at φ orig_caps spec) obj_ids ∧* si_cspace ∧* R≫⦄

⟦well_formed spec; obj_ids = dom (cdl_objects spec); distinct obj_ids;
 distinct free_cptrs; orig_caps = map_of (zip obj_ids free_cptrs);
 length obj_ids ≤ length free_cptrs⟧
⟹ ⦃≪objects_empty spec φ [obj←obj_ids. cnode_at obj spec] ∧*
      ⋀* map (si_cap_at φ orig_caps spec) obj_ids ∧*
      ⋀* map (si_cap_at φ dup_caps spec)
           [obj←obj_ids. cnode_or_tcb_at obj spec] ∧*
      si_cspace ∧* R≫⦄
    init_cspace spec orig_caps dup_caps obj_ids
    ⦃≪objects_initialised spec φ [obj←obj_ids. cnode_at obj spec] ∧*
      ⋀* map (λcptr. (si_cnode_id, cptr) ↦c NullCap)
           (take (length obj_ids) free_cptrs) ∧*
      ⋀* map (si_cap_at φ dup_caps spec)
           [obj←obj_ids. cnode_or_tcb_at obj spec] ∧*
      si_cspace ∧* R≫⦄
```

Fig. 5. Individual rules for *init_tcbs* and *init_cspace*.

Continuing in this trend, we decompose the problem of initialising a single object into the separate parts of an object, namely its fields and its individual capability slots. This is embodied in the following rule.

```
⟦dom (slots_of obj_id spec) = slots; distinct slots⟧
⟹ object_initialised spec φ obj_id =
    (object_fields_initialised spec φ obj_id ∧*
     ⋀* map (object_slot_initialised spec φ obj_id) slots ∧*
     object_empty_slots_initialised spec φ obj_id)
```

Such a decomposition is not necessarily true for any separation logic and any concept of partly initialised object. Being able to prove the rule above as an equality was one of the design goals of our separation logic. In particular, the definition of *object_initialised* (see Sec 5.2) contains an existential quantifier over *kernel_object_id* and *spec_object* which needs to be well-behaved enough to lift over the separating conjunction on the right hand side of the rule.

The proof of CNode initialisation is representative of the proofs of other object types. Capability slots in CNodes are initialised in a two-step process as described in Sec 4. We define a predicate *cnode_half_initialised* to describe this intermediate state and use the above decomposition rule for *object_initialised*, decomposing *cnode_half_initialised* in a similar way, combined with the *mapM* rule described in Sec 5.1 to reduce reasoning about loops to single capability slots. Arriving at the leaf kernel calls of the *init_cnode* function, the initialiser extracts the capabilities that authorise it to make these calls. These capabilities are mentioned in Fig 5 as *si_cap_at φ caps spec obj_id* in a map over all such capabilities. The following rule allows us to extract the specific one we need.

$$\llbracket x \in xs;\ distinct\ xs;\ \bigwedge R.\ \{\!|\ll P \wedge^* I\ x \wedge^* R\gg|\!\}\ f\ \{\!|\ll Q \wedge^* I\ x \wedge^* R\gg|\!\}\rrbracket$$
$$\Longrightarrow \{\!|\ll P \wedge^* \bigwedge^* map\ I\ xs \wedge^* R\gg|\!\}\ f\ \{\!|\ll Q \wedge^* \bigwedge^* map\ I\ xs \wedge^* R\gg|\!\}$$

We join the initialiser proof with formal specifications of the seL4 API that are explained in the next section.

5.4 seL4 API specification

It is a universal hazard in formal specification that the specification does not meet requirements, is inconsistent or does not match the code. We narrow the requirements gap by proving a high-level correctness statement. We address the latter two by the formal connection of the system initialisation proof to the capDL model of the seL4 kernel, which formally abstracts the seL4 binary as illustrated in Fig 2.

While we re-use the existing capDL kernel model, we did have to prove new properties about it. Existing proofs about seL4 mostly concerned global invariants and all possible, potentially malformed or malicious inputs. Exercising the kernel API from a user-level proof, however, requires a different perspective: given a specific good pre-state for an API call, show the effect of the API call on this state, and determine which other parts of the kernel state are (not) affected. Separation logic proved a good match for this kind of specification. This style of proof does not tell us anything about invariant preservation within the kernel, but it gives us the information we need for user-level proofs. We expect the separation logic triples we proved about the seL4 API to be useful in other user-level proofs as well; they were not specific to the initialiser. These triples are typically large, around 30–50 lines each, because they capture the precise conditions needed for a specific kernel call to succeed.

The system model we use to connect to the kernel formalisation is somewhat simplistic: it assumes that only one thread in the system can make kernel calls and affect the system state. This allows the initialiser model to treat the kernel as a library that embeds the kernel state in the initialiser state. It also allows us to avoid reasoning about interleaved user executions. This works for our one-thread initialiser, but obviously would have to be generalised for more complex systems.

6 Experience, Limitations, and Assumptions

This section describes some of the lessons learnt in this verification and discusses the limitations and assumptions of the current version of the proof.

The size of the initialiser model is relatively small: roughly 400 lines of Isabelle definitions. It connects to the seL4 capDL-level kernel model of about 4,150 lines. This connection to the fully realistic kernel model is the main source of complexity in the proof. As can be seen in Sec 4, the initialiser has to deal with a full kernel API with all its real-life complexities and wrinkles.

The proof specific to the user-level initialiser measures 8,100 lines of Isabelle, the separation logic proofs about kernel functions another 10,500 lines, coming to a total of 18,500 lines overall. This compares to 25,300 lines for the refinement proof between the capDL kernel model and the functional specification, and 200,000 lines for the functional specification to the C code of the kernel [4].

Previous verifications based on seL4 were either refinement proofs where one full specification layer is connected to another such layer [4], or proofs of global security properties [7, 12]. The use of specific API functions in our setting, combined with local separation logic statements about them, gives our proofs a distinctly different flavour. While previous proofs had to show global invariants that are inconvenient to express in separation logic, we could get away without stating any global kernel-level invariants at all. The local separation logic specifications were sufficient.

Our compositional state space and the corresponding separation algebra was crucial for reasoning about individual components of each object separately. Another difference between the initialiser and previous seL4 proofs was the heavy use of nested loops. Again, our separation logic setup enabled us to decompose these loops into local steps without stating complex invariants.

These benefits of separation logic reasoning were not an accident. We expended considerable effort fine-tuning and designing the underlying separation algebra instance such that the higher-level proofs would later fall out relatively conveniently. This design was iterative, leading through a number of rather complex instantiations with the simple state space lifting presented here as the final result. The Isabelle-enforced abstraction layer, which the separation algebra type class brought, enabled us to change the algebra instantiation and heap structure several times underneath the kernel proofs. We only had to re-prove the basic axioms and frame properties of the leaf functions.

We share a frequent experience in separation logic proofs: more automation would have been welcome. While the generic separation logic setup in Isabelle/HOL provided basic proof tactics, higher-level automation such as frame computation/matching would have improved productivity. We are currently investigating how such support can be implemented.

Despite our comprehensive top-level correctness statement, there are still a number of limitations in the initialiser and corresponding proof.

A general limitation of this style of proof is that it shows a safety, not a liveness property: only if the initialiser finishes successfully do we know that the resulting state is correctly initialised. Since almost all loops in the initialiser are (potentially nested) maps over finite lists, termination is not an issue. However, the initialiser may legitimately fail, e.g. because of insufficient memory. This limitation could be lifted with further explicit assumptions about the *bootinfo* structure provided by the kernel. An assumption of the original seL4 verification was that the kernel boots correctly. We further assume that the *bootinfo* structure provides correct information about the layout of memory and capabilities.

Sec 5.2 mentions fundamental constraints on system configurations formalised in the predicate `well_formed` on the input specification. We also use this predicate to encode specific limitations of our current initialiser model. We currently do not allow the system configuration to mention untyped capabilities, IRQ capabilities and ASID pool capabilities. This corresponds to static system configurations as used in a separation-kernel setting [7]. With the basic reasoning framework set up, we think these limitations can be lifted easily in future work.

At present, capDL only models the protection state of the system, not its memory content. This means we also do not model the loading of program code. This limitation is less severe than it may sound, because in the envisioned application space, the system image loaded from disk already contains all application binaries. That is, loading program code is reduced to mapping the right memory frames into the right virtual address spaces, which we do model.

Finally, we do not model the kernel scheduler in our proofs, because the capDL kernel model does not provide enough detail on it. Our underlying execution model implicitly assumes that the initialiser is the only running thread in the system. Since the initialiser runs with highest priority and only produces threads with lower priority this assumption is trivially satisfied until the initialiser terminates. We do prove that the initialiser always remains runnable.

7 Conclusion and Related Work

Initialising systems according to a given configuration, and guaranteeing that the initialisation is correct, are both hard and critical tasks. Security requirements for high-assurance certification of separation kernels (SKPP) for instance include providing evidence that the initialisation function establishes the system in a secure state consistent with the configuration data [8]. Configuration data describes high-level partitions and authorised information flows between partitions.

SELinux policies allow fine grained MLS security, but the richness of these policies makes it difficult to understand them. Hicks et al. [2] developed a formal semantics for SELinux policies in Prolog and demonstrated that it was possible to show information flow properties of SELinux policies.

The EROS kernel partially side-steps the initialisation problem by persistence; it simply restarts at the last saved checkpoint. The initial system image is constructed by hand and the creation and instantiation of confined subsystems uses *constructors* that are part of the trusted computing base [13]. The proof of the correctness of these constructors is with respect to a high-level model of EROS only, not formally linked to the EROS code.

The OKL4 microkernel [9] moves the initialisation problem almost entirely to offline processing and runs the initialisation phase once, before the system image is built. Similarly to EROS, when the machine starts, it loads a fully pre-initialised state. While this makes it possible to inspect the initialised state offline, a full assurance case must be made for each system. In our approach, assurance about system initialisation is now reduced to reasoning about static, formal capDL system descriptions.

In this paper we presented the formalisation and correctness proof of a generic, automatic system initialiser that brings an seL4-based system from boot state into a desired access control configuration. From such a configuration, we can then reason with confidence about the security of the resulting system.

We have shown a general separation logic framework that can be used to reason about such user-level systems, we have produced a proof framework to reason about user-level executions on top of a formally verified microkernel API, and we have applied it to show the correctness of the initialiser model.

While the initialiser we present here is specific to seL4, we think that the general principle and pattern of reasoning would generalise to other capability-based systems. Future work includes the functional correctness proof down to the C code level of the initialiser. We expect this proof to be simpler than the seL4 correctness, because the initialiser code itself is much simpler. Its complexity lies in the interaction with the kernel, which we have treated here.

Acknowledgements We are grateful to Mark Staples and Toby Murray for their feedback on drafts of this paper.

References

1. J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.
2. B. Hicks, S. Rueda, L. S. Clair, T. Jaeger, and P. D. McDaniel. A logical specification and analysis for SELinux MLS policy. In V. Lotz and B. M. Thuraisingham, editors, *SACMAT*, pages 91–100. ACM, 2007.
3. G. Klein. From a verified kernel towards verified systems. In K. Ueda, editor, *8th APLAS*, volume 6461 of *LNCS*, pages 21–33, Shanghai, China, Nov 2010. Springer.
4. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220. ACM, 2009.
5. G. Klein, R. Kolanski, and A. Boyton. Mechanised separation algebra. In *3rd ITP*, volume 7406 of *LNCS*, pages 332–337. Springer, Aug 2012.
6. I. Kuz, G. Klein, C. Lewis, and A. Walker. capDL: A language for describing capability-based systems. In *1st APSys*, pages 31–36, New Delhi, India, Aug 2010.
7. T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symp. Security & Privacy*, Oakland, CA, May 2013.
8. National Security Agency. U.S. government protection profile for separation kernels in environments requiring high robustness, version 1.3, Jun 2007.
9. Open Kernel Labs. OKL4 microkernel, reference manual. `http://wiki.ok-labs.com/downloads/release-3.0/okl4-ref-manual-3.0.pdf`, Sep 2008.
10. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.
11. T. Sewell, M. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proc. 34th PLDI*, pages 471–481. ACM, Jun 2013.
12. T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *2nd ITP*, volume 6898 of *LNCS*, pages 325–340, 2011.
13. J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *IEEE Symposium on Security and Privacy*, pages 166–176. IEEE Computer Society, 2000.