NICTA

Final Report for AOARD Grant #FA2386-12-1-4022

Formal System Verification - Extension 2

June Andronick, Gerwin Klein

8 August 2012

Principal Investigators: Gerwin Klein and June Andronick

Email: gerwin.klein@nicta.com.au, june.andronick@nicta.com.au
Institution: NICTA
Mailing Address: 223 Anzac Parade, Kensington NSW 2052, Australia
Phone: +61 2 8306 0578
Fax: +61 2 8306 0406

Period of Performance: 9 Feb, 2012 - 8 Aug, 2012
Reporting Period: 9 Feb, 2012 - 8 Aug, 2012

Abstract

The aim of AOARD project #FA2386-12-1-4022 ("Formal System Verification - Extension 2", running from 9 February 2012 to 8 August 2012) is to provide an initial framework prototype for efficiently performing formal proofs of targeted security or safety properties about large, complex software systems. The framework is meant to be generic in terms of the targeted property for the system and to minimise the verification effort while providing high-assurance guarantees at the source code level.

This document is the final report of the project, presenting our initial framework, formalised in the theorem prover Isabelle/HOL [7]. The framework takes as input the concrete implementation (translated into formal logic) of any system made of a set of components running on top of an OS microkernel. The framework explicitly identifies and formally states all theorems required for a given property to hold about the system. In particular, the framework assumes that the system follows the strategy of a formally verified, minimal computing base, i.e. that the system is made of a minimal set of trusted components, isolated from untrusted ones by an OS kernel which we can formally reason about. The framework therefore requires as input a proof of the kernel's correctness and isolation properties, and a proof that the trusted components satisfy the targeted property. The former proof can be performed once for any system built on a given kernel. The latter proof is specific to the system and its trusted components' behaviour and must be provided for each instance. The framework then combines these proofs to provide a formal proof that the property holds at the source code level of the whole system.

1 Introduction

The larger research vision which this project is part of aims at building truly trustworthy software systems, i.e. systems whose safety or security properties can be guaranteed by means of formal proofs holding at the source code level of the whole software system. The target is real-world systems, i.e. large, complex software systems, typically comprising millions of lines of code. The approach taken here to provide formal guarantees about such large code base is to minimise the *trusted computing base* by designing the system as a set of components such that the targeted property only relies on the behaviour of a small number of components, so-called *trusted components*. In other terms, trusted components can potentially violate the property —and we will need to prove that they do not— whereas untrusted components are not given enough rights to violate the property, therefore they can be as large and complex as needed and their behaviour can be arbitrary. This MILS-style of security architecture [1] enables us to concentrate the verification effort on the trusted components.

However, this approach relies on the fact that untrusted components can be isolated from trusted ones. This isolation or control of communications between components has to be ensured by the underlying kernel. The kernel is the only component running in the privileged mode of the hardware, known as *kernel mode*. In this mode, the hardware allows free access to all resources. The kernel is therefore the most critical part of the system. Every other component runs in *user mode* and needs to perform a kernel call to be able to communicate with other components. User mode execution essentially only admits free access to the currently mapped memory and registers. All other interaction can be restricted via kernel access control. Which memory is currently mapped for each user component is again controlled by the kernel and thereby subject to the security policy of the system. If a user component tries to access memory it is not authorised to access, the hardware generates a page fault which leads to a kernel call. The kernel can then react appropriately. If the user component would like to invoke any other form of communication such as interrupts, synchronous or asynchronous message passing, it needs to invoke the corresponding kernel primitive. These are again subject to the access control policy. This strictly enforced separation by policy allows us to reason about the execution of untrusted components without a specification of their behaviour.

The key differentiator in our research is that not only are the systems designed and implemented following this approach, but all the arguments need to be supported by formal proofs: a proof that the kernel enforces this isolation, a proof that the trusted components do not violate the targeted property, a proof that untrusted components behaviour can be ignored, and finally a proof that all these separate proofs can be combined into one overall theorem.

The project reported here focuses on this last critical step: investigating how proven theorems (about the kernel, the trusted components, the isolation) can be combined into a single theorem stating that the targeted property holds at each (small) step of any possible execution of the whole system at the source code level. It follows AOARD project #104105 [3] where we presented an informal, yet detailed plan for achieving the decomposition of the overall theorem into proofs about the various part of the kernel-based, componentised system. Here we formalise such a vision by developing, in a common framework in the Isabelle/HOL [7] theorem prover, a simplified representation of all the theorems involved, which are then combined in the overall system proof.

Namely, the project goal was to:

> *“develop a simplified representation of all the theorems involved in the initial framework and prove that the targeted property can be derived for the set of these simplified theorems.”*

The framework we developed identifies the intermediate properties needed to prove that a given property, derived from an invariant I, holds for a given system described by a state machine. In other words, the user of the framework is given a set of statements to prove (*proof obligations*), which are then automatically combined in the framework into a theorem that I is preserved by the code of the whole system.

The aim of the framework is to minimise the set of proof obligations as much as possible (i.e. weaken the properties to be proved by the user), by following a defined proof strategy, and discharging (i.e. automatically proving) any intermediate lemma needed to weaken the properties. In particular, our framework assumes the system follows a strategy of verified, minimal computing base, and therefore requires as input a proof of the kernel’s correctness and isolation properties, and a proof that the trusted components satisfy the targeted property.

Our framework is generic in that it refers to any property that can be derived for a system invariant and to any kernel which can provide isolation and functional correctness guarantees. Note that these can be considered to be very strong requirements. However, they are realistic thanks to our previous achievements in this area: a first step in our vision of truly trustworthy systems has been to provide a formally verified microkernel basis. We have previously developed the seL4 microkernel, together with a formal proof (in the theorem prover Isabelle/HOL) of its functional correctness [6]. This means that all the behaviours of the seL4 C source code are included in the high-level, formal specification of the kernel. This work enabled us to provide further formal guarantees about seL4, in particular for isolation: we formally proved that seL4 enforces integrity and authority confinement [8], and are currently working on confidentiality.

This means that immediate next steps of the project presented here will be to instantiate the generic theorems about the kernel used in the framework with the actual existing theorems about seL4. We have already started on this task and first results are promising in being able to discharge all theorems specific to the kernel alone. This means that only the proof obligations specific to the targeted property or the trusted components will be required from the user of the framework.

We begin the remainder of this report with an overview of the framework scope and strategy in Section 2. Then, in Section 3, we give some background on the proof techniques and artefacts used in the formalisation of the framework, which is itself presented in detail in Section 4. Finally, we draw conclusions and present future directions in Section 5.

2 Framework Overview

This section gives an informal overview over the framework and explains the main formalisation steps.

2.1 Scope and Assumptions

The goal of the framework we are building is to prove that a given property P is true about a given system S. In this project we will focus on:

- properties that can be derived from a system invariant I;
- systems built as a set of trusted and untrusted components running on a kernel.

The first assumption constrains the problem space to the class of formal safety properties. In practice, this means that the goal comes down to proving that

$$I \textit{ is preserved by any execution step } t \textit{ of the system.} \tag{1}$$

where the system is represented as a state machine with transitions t between system states. In this report we treat the (usually more difficult) step case during system execution. For the property P to hold over the entire system execution, we must additionally show that the initial state of the system also satisfies the invariant I.

The second assumption entails that state transitions can be either kernel transitions or user transitions, or transitions switching from one mode to another. Figure 1 presents a simplified version of a typical execution of the system: a user $u1$ runs, then it performs a kernel call, which traps into kernel mode where the kernel executes the command, and then schedules another user, say $u2$; this user then starts running until it in turn performs a kernel call, etc. The kernel may also schedule the *idle thread*, switching to idle mode.

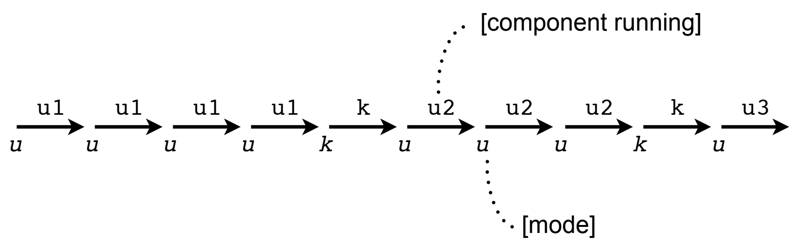

Figure 1: Example of system execution at the code level (simplified).

More precisely, the state automaton looks as in Figure 2, where there exist three types of transitions. Firstly, *user transitions* tu represent the execution of a user program that does not involve invoking the kernel. Such transitions go from a state in user mode to a state still in user mode. Secondly, *kernel transitions tk* start from a state already in kernel mode and execute the kernel call. Since the kernel is non-preemptible, such transitions are atomic, including the scheduling of a new user or the idle thread, and therefore end up in a state either in user mode or idle mode. Finally, there are transitions trapping to kernel mode, from either user mode or idle mode, and either due to a system call or an interrupt. These transitions do not change the state itself apart from the

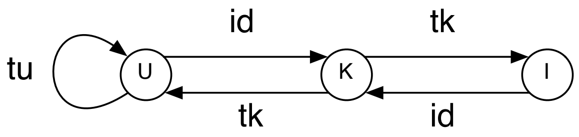

Figure 2: State automaton representing a kernel-based system.

mode, and are therefore represented by the identity function id. The goal (1) then becomes:

> *I is preserved by any execution step t of the system, where t is either a kernel transition tk or a user transition tu or t only changes the mode.* (2)

2.2 Proof Approach and Architecture

The decomposition of the final invariant preservation goal (2) into several proof obligations will use two main proof techniques:

- ***Abstraction***: here we first show that a specific kind of concrete transition tc can be abstracted to a simpler (abstract) transition ta where it is easier to reason about the invariant preservation; this will be done using forward simulation, illustrated in Figure 3, explained in Section 3; the invariant preservation is then shown at the abstract level. This means that each transition for

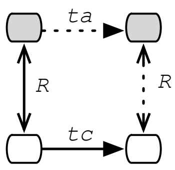

Figure 3: Forward simulation between concrete transition tc and abstract transition ta.

  which this technique is used can be proved at a different level of abstraction, enabling us to choose the level best suited to each particular problem.

- ***Locality of state changes***: here we use the fact that parts of the state may stay unchanged during given transitions, and if the invariant I does not depend on what might have changed, then I is trivially preserved. We will note $s =_{|M} s'$ when the state s' is equal to the state

s up to a given substate M. If I does not depend on M and $s =_{|M} s'$ where s (resp. s') is the state before (resp. after) a transition, then this transition preserves I. We will use this when reasoning about transitions on behalf of untrusted users: if isolation is enforced by the kernel, such transitions may only change what the untrusted components may modify. Now, by definition of untrusted components, the invariant I should not depend on the parts of the state that these components have access to. This leads us to place a reasonable assumption on the invariant to be discharged by the user of the framework.

Using these two techniques, we can decompose the proof that invariant I is preserved by transition t (where each step of t is either a kernel transition tk or a user transition tu or the identity over states) into proofs at a different level of abstraction or proof obligations that I is independent of some substate, as illustrated schematically on Figure 4.

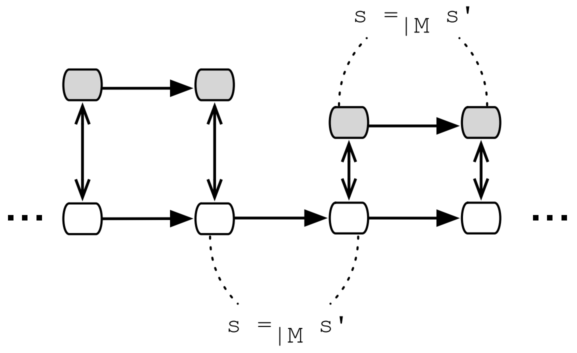

Figure 4: Schematic representation of proof decomposition.

2.3 Framework Formalisation Strategy

Since the framework is generic in the property to prove and the targeted system, it can never fully prove the property, but only present a list of proof obligations that will need to be satisfied once instantiated to a given property and a given system. An intermediate step will be to instantiate to a given kernel, seL4, and leave only the components and property generic.

In any case, the usefulness of the framework can be measured in terms of the "quality" of the proof obligations it produces, i.e. how *weak* they are. For instance, the framework could produce a single proof obligation, requiring the user to prove that t preserves I, without decomposing anything at all. This is of course the strongest obligation it can produce and is of no use. Following the strategy described previously, the proof obligations can be gradually decomposed and weakened, with the decomposition proved correct in the framework.

The strategy to formalise the framework is to mimic this gradual decomposition and weakening of the proof obligations, using Isabelle's *locale* mechanism. Locales will be explained in detail in Section 3; the basic idea is that a locale l_1 contains a set of definitions, assumptions and theorems, and we can declare another locale l_2 as being a *sublocale* of l_1. This requires us to prove that all the assumptions (in our case proof obligations) of l_1 can be derived from l_2 ones. This in turn enables us to replace a strong proof obligation in l_1 by a weaker or a set of smaller proof obligations in l_2. What is automatically granted in the process is that all theorems proved in the context of l_1 are automatically inherited in l_2, because they were proved using l_1 assumptions, which can be derived from l_2 ones.

Our framework will therefore consists in a hierarchy of sublocales, whose root contains the overall theorem derived from strong, obvious proof obligations, and the "last" sublocale contains proof obligations that are as weak as possible.

3 Background

3.1 Isabelle Notation

Isabelle/HOL [7] is an interactive proof assistant — a program that supports the user in interactively developing formal proofs in Higher-Order Logic. The assistance consists of a) machine-checking proofs for soundness and admitting only correctly derived theorems, and b) automated proof tactics, definition and notation support, a human-readable proof language and further features that make it possible to interactively produce large-scale formal developments.

Isabelle's notation mostly conforms to everyday mathematical notation and the notation used in functional programming. For instance the type of total functions is denoted by ⇒. The notation `nat ⇒ bool` stands for a function from natural numbers to booleans. Type variables are written `'a`, `'b`, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ. The type of pairs is written as `'a × 'b`.

Definitions are introduced by the keyword **definition** followed by a name, an optional type annotations, the keyword **where** and the definition itself. For example, the statement

definition `successor :: "nat ⇒ nat"`
  where
  `"successor n ≡ n + 1"`

defines a new constant `successor` of type `nat ⇒ nat` which adds one to its argument.

Lemmas and proof rules are written `⟦A; B⟧ ⟹ C` where `A` and `B` are assumptions and `C` is the conclusion of the proof rule. Lemmas and theorems are introduced by the keywords **lemma** and **theorem** followed by a name and the theorem statement. This is usually followed by a potentially lengthy formal proof, which we elide in this document. Instead we describe the main reasoning idea of the proof informally. Since this document is automatically generated directly from the Isabelle proof sources, the lemma statements are formally machine-checked for soundness.

An example lemma with elided proof is shown below.

lemma `example: "A ∧ B ⟹ B ∧ A"` ⟨*proof*⟩

The proof of this lemma is obvious for humans, and automatic for Isabelle.

3.2 Isabelle Locales

The Isabelle/HOL proof assistant provides a mechanism for modularising formal developments and for conducting a formal development in a named local context. These contexts can then later be combined and/or instantiated. They allow abstract reasoning from a group of assumptions and provide facilities for establishing formal relationships between different contexts.

The command **locale** `L = parent +` introduces such a local context with name `L` and parent locale `parent`, which may be empty. This declaration is followed by a list of declarations and assumptions that make up the locale.

For example, the following declaration introduces a locale `L` that assumes the existence of a constant `f` of type `nat ⇒ bool`. It additionally assumes that `f n` for all `n < 10` and gives that assumption the name *n*.

```
locale L =
  fixes f :: "nat ⇒ bool"
  assumes n: "∀n < 10. f n"
```

In this context, new properties can now be derived. For instance:

```
context L begin

lemma f5: "f 5" ⟨proof⟩

end
```

An instantiation of this locale would provide a concrete implementation of the function *f* and would have to show that the assumption *n* is true about this function. As a consequence all lemmas proved in the context *L* would then automatically become available.

In our formal system verification framework in Section 4 we will not use such full instantiations. Instead we will instantiate locales partially to achieve a step-wise more refined formalisation. We can achieve such a partial instantiation by defining an initially independent, more detailed example locale *L2*:

```
locale L2 =
  fixes f :: "nat ⇒ bool"
  assumes n: "∀n < 20. f n"
```

We can now prove that the assumptions of locale *L2* already satisfy the assumptions of locale *L*. This will make all facts previously proved in context *L* available in *L2* as well. The **sublocale** command tells Isabelle about this relationship and establishes the required proof obligations.

```
sublocale L2 < L ⟨proof⟩
```

We can now make use of the fact *f5* as if we had proved it in context *L2*.

We will use this mechanism in our framework to first establish a very strong context in which the final overall system property can be proved easily. We then develop a new, more detailed context with easier-to-prove assumptions and show that — usually because of the added detail and structure — we can establish the assumptions of the previous context and therefore retain the truth of our overall system property. Repeating this step will give us an increasingly sophisticated setup using incremental proof steps that work on one aspect of the system at a time.

3.3 Refinement and Forward Simulation

In our framework below we will make heave use of formal refinement [4] and the corresponding proof technique of forward simulation.

Generally speaking, formal refinement is the concept that a system is described at two different levels of abstraction — an abstract and a concrete one. Usually, the abstract description allows easier reasoning, and the concrete description easier validation that the description actually represents the system being reasoned about. In the extreme case, the concrete description is the source code or binary code of the system itself, or an automated translation thereof into the theorem prover, while the abstract description is high-level and easy to reason about.

A proof of refinement entails that Hoare logic properties, expressed in pre/post conditions or invariants, can be transferred from the abstract to the concrete level. That means, when refinement is proved once, any Hoare logic property of the concrete system can be reduced to an equivalent Hoare logic property of the abstract system. In previous work we have demonstrated that the reduction of proof effort induced by this technique can be an order of magnitude [8].

Refinement, defined as the ability to transfer such properties over all traces of possible inputs, is hard to establish directly. The prevalent method is to exploit the well-known implication to forward simulation instead, which reduces a proof of refinement to a separate proof about each local transition of the system. Figure 5 that we have already seen in Section 2.2 illustrates the resulting proof obligation:

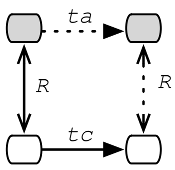

Figure 5: Forward simulation between concrete transition tc and abstract transition ta.

We must show that there exists a refinement relation $\mathtt{R}$ between concrete and abstract states, such that for each concrete transition $\mathtt{tc}$, if the initial states are in the relation $\mathtt{R}$, then there will be an abstract transition $\mathtt{ta}$ such that the final states again are in this transition $\mathtt{R}$.

Properties can then be transported between $\mathtt{tc}$ and $\mathtt{ta}$ modulo this refinement relation $\mathtt{R}$.

4 Formal Definitions and Framework

This section presents the formalisation of the framework in the theorem prover Isabelle/HOL. We begin in Section 4.1 with basic definitions and the key theorems that support refinement steps. Section 4.2 then presents the increasingly more sophisticated instantiations of the framework that, while becoming larger, lead to proof obligations that are simpler and easier to satisfy.

4.1 Definitions

Let us first present the formal definitions of invariant preservation and forward simulation that will be used in the framework. We also provide several lemmas enabling to prove invariant preservation in various ways, including our *abstraction* proof technique described in Section 2.2, namely using forward simulation to prove invariant preservation at an abstract level.

4.1.1 Notion of Forward simulation

The formalisation of forward simulation in Isabelle follows precisely the definition given in Section 3:

```
definition fw_sim_step
  where
  "fw_sim_step R tc ta ≡
     ∀s1 s2'. (∃s1'. R (s1,s1') ∧ tc  (s1',s2')) ⟶
               (∃s2 . ta (s1,s2) ∧ R (s2,s2'))"
```

4.1.2 Notion of Invariant Preservation over Transitions

Invariant preservation of an invariant `I` by a transition `t` is also defined in a straightforward formalisation of the fact that if `I` is true about the state before the transition then it is true about the state after the transition.

```
definition inv_preserved_by_trans
  where
  "inv_preserved_by_trans I t ≡ ∀s1 s2. t (s1,s2) ⟶ I s1 ⟶ I s2"
```

Invariant preservation could be subject to a condition, for instance that the currently running component is trusted. We therefore define a variant of invariant preservation under some condition `c`, and prove that the previous definition is equivalent to having a condition which is always true.

```
definition inv_preserved_by_trans_cond
  where
  "inv_preserved_by_trans_cond c I t ≡
   ∀s1 s2. c s1 ⟶ t (s1,s2) ⟶ I s1 ⟶ I s2"

lemma inv_preserved_by_trans_cond_true:
  "inv_preserved_by_trans_cond (λ_. True) I t =
    inv_preserved_by_trans I t"
  ⟨proof⟩
```

4.1.3 Ways of proving invariant preservation

Now we provide three ways of proving invariant preservation:

- by case distinction on a condition: to show that an invariant is unconditionally preserved, we can show that it is preserved under a condition `C` and it is also preserved on the negation of `C`;

- by case distinction on the type of transition: to show that an invariant is preserved by a transition `t`, where `t` is either `t1` or `t2`, we can show that it is preserved by both `t1` and `t2`;

- by using forward simulation: to show that an invariant is preserved by a concrete transition `tc`, where `tc` is in forward simulation with an abstract transition `ta` with respect to a state relation `R`, we can show that the *lifted* invariant `lift R I` is preserved by `ta`. The lift operation is defined by `lift R I ≡ λs. ∃s'. R (s, s') ∧ I s'`. In other words, the lifted invariant is true about an abstract state `s` if there exist a concrete state `s'` related to it by the relation `R` such that `I` is true about `s'`. This approach requires us to prove two additional conditions about the state relation. First we need to be able to abstract the concrete initial state which satisfies `I`. We therefore need the state relation to be total on all concrete states satisfying `I`:
  ```
  total_rel_on_concrete_under_cond R I ≡
    ∀s'. I s' ⟶ (∃s. R (s, s'))
  ```
  Secondly, once we have established that the lifted invariant is preserved by the resulting abstract state, we need to conclude that the concrete resulting state satisfies `I`. In other words we need the state relation to *preserve* the invariant from abstract states to concrete ones:
  ```
  rel_preserves_inv_down R I ≡
    ∀s s'. R (s, s') ⟶ lift R I s ⟶ I s'
  ```

In Isabelle, this is expressed in the following lemmas below.

```
lemma inv_preserved_by_trans_cases:
  "⟦ inv_preserved_by_trans_cond C I t;
     inv_preserved_by_trans_cond (λs. ¬ C s) I t ⟧ ⟹
     inv_preserved_by_trans I t"
  ⟨proof⟩

lemma inv_preserved_by_trans_disj:
  "⟦ ∀p. t p = t1 p ∨ t2 p;
     inv_preserved_by_trans I t1;
     inv_preserved_by_trans I t2  ⟧ ⟹
     inv_preserved_by_trans I t"
  ⟨proof⟩
```

```
lemma fw_sim_step_preserves_inv_cond:
  "⟦ total_rel_on_concrete_under_cond R I;
     rel_preserves_inv_down R I;
     fw_sim_step R tc ta;
     inv_preserved_by_trans_cond (lift R C) (lift R I) ta  ⟧ ⟹
     inv_preserved_by_trans_cond C I tc"
  ⟨proof⟩

lemma fw_sim_step_preserves_inv:
  "⟦ total_rel_on_concrete_under_cond R I;
     rel_preserves_inv_down R I;
     fw_sim_step R tc ta;
     inv_preserved_by_trans (lift R I) ta  ⟧ ⟹
     inv_preserved_by_trans I tc"
  ⟨proof⟩
```

4.2 The Framework

As explained in Section 2, the framework will be defined as a hierarchy of locales, in which proof obligations are gradually decomposed and weakened. Each locale introduces an additional layer of structure and instantiation into the overall system transition system. This structure can then be used to export simpler proof obligations to the user. These are the assumptions of the locale. The formal sublocale proofs show that the refined structure is a formal instance of the previous one, and therefore all consequences derived from the previous locale remain true. This includes in particular the formal statement that the transition system satisfies the whole-system invariant at the source-code level.

In this section we describe this set of locales, starting with the base locale stating the invariant preservation for the overall system. Figure 6 gives a high-level overview of which parts of the overall transition system is refined by a particular locale instantiation step.

The figure starts at the top with the base-locale that almost trivially satisfies invariant preservation. Each arrow in the figure represents are refinement or instantiation step. The source of the arrow indicates which part of the transition system is refined.

4.2.1 The base locale, containing the final theorem (Step 1)

This locale contains the final targeted theorem, stating that a given system invariant `givenI` over concrete states of type `'cstate` is preserved by a state transition `global_transition` describing the whole system. Here we already take our scope into account (see Section 2.1), namely that we are targeting kernel-based system. This means that we define our global transition as being either a kernel transition or a user transition or the identity over states. We also start our decomposition, with proving our final theorem using two assumptions: invariant preservation over kernel transitions

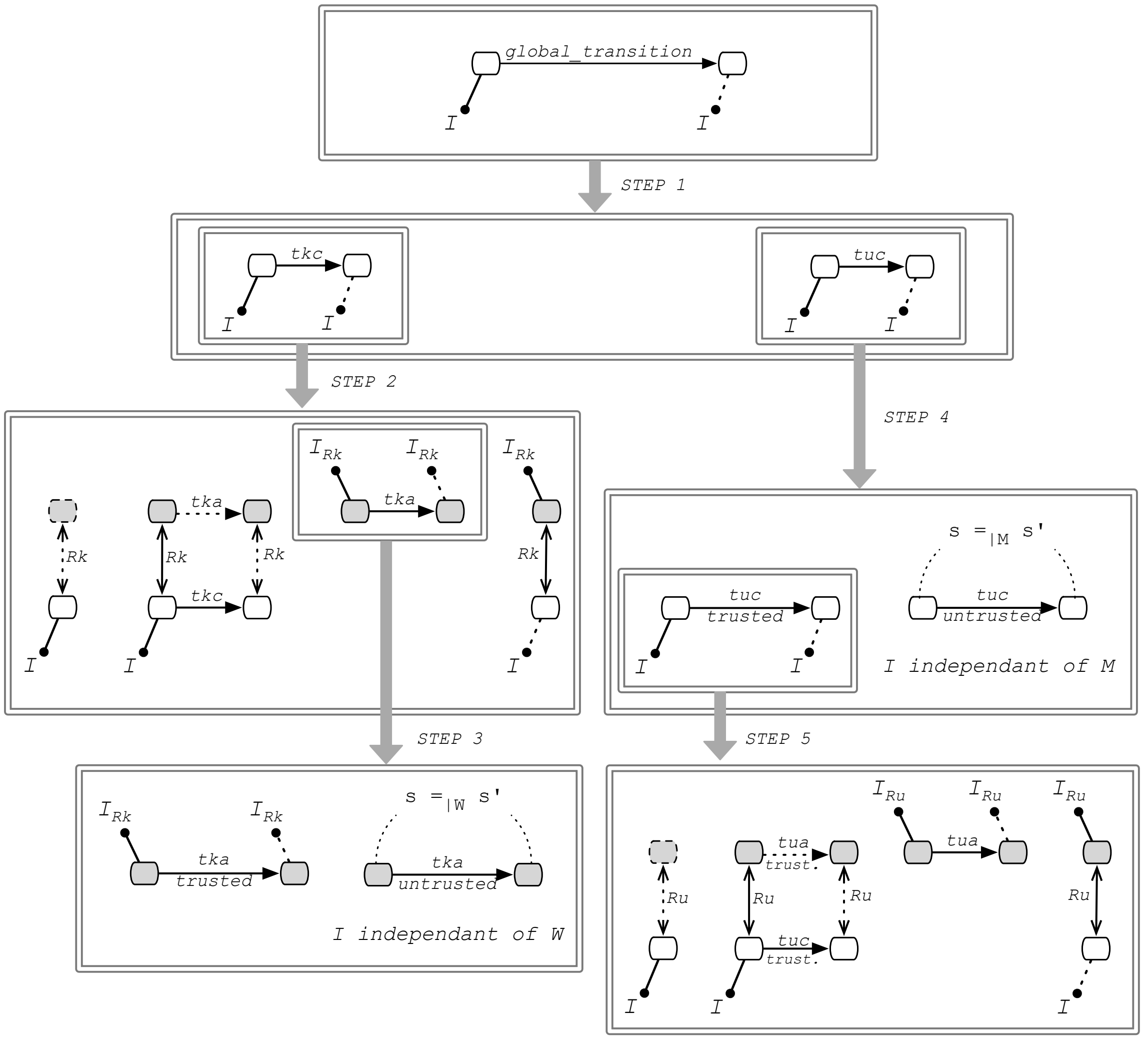

Figure 6: Structure of the locales hierarchy and proof overview.

and user transitions respectively. This proof uses the `inv_preserved_by_trans_disj` lemma described in Section 4.1.3.

```
locale framework_final =
  fixes tkc:: "'cstate × 'cstate ⇒ bool"
  fixes tuc:: "'cstate × 'cstate ⇒ bool"
  fixes givenI:: "'cstate ⇒ bool"
  assumes tkc_pres_inv: "inv_preserved_by_trans givenI tkc"
  assumes tuc_pres_inv: "inv_preserved_by_trans givenI tuc"

context framework_final
begin

definition global_transition
  where
  "global_transition p ≡ tkc p ∨ tuc p ∨ (split op =) p"

theorem final_theorem:
  "inv_preserved_by_trans givenI global_transition"
 ⟨proof⟩

end
```

4.2.2 Using Abstraction for Kernel Transitions (Step 2)

Now the goal is to weaken the assumption as much as it is possible while remaining generic on `givenI` and the system. Let us first look at the kernel transitions. Here we can prove the invariant preservation at a more abstract level, assuming we have a verified kernel, i.e. a kernel for which we have a proof of forward simulation. This is possible only if the state relation satisfies the two conditions mentioned in Section 4.1.3. Formally, this means defining a new locale `framework_kernel_fw_sim`, containing two new variables (the abstract kernel transition and the kernel state relation) and replacing the assumption about invariant preservation by concrete kernel transitions with four new assumptions: the preservation by abstract kernel transitions, the forward simulation statement and the two conditions on the state relation. Although we increase the number of assumptions, these should be easier to prove, except for the forward simulation one that is meant to be established once for the kernel and that we already have for the seL4 microkernel.

```
locale framework_kernel_fw_sim =
  fixes tkc:: "'cstate × 'cstate ⇒ bool"
  fixes tuc:: "'cstate × 'cstate ⇒ bool"
  fixes givenI:: "'cstate ⇒ bool"

  fixes tka:: "'astate × 'astate ⇒ bool"
  fixes Rk:: "'astate × 'cstate ⇒ bool"

  assumes Rk_cond1: "total_rel_on_concrete_under_cond Rk givenI"
  assumes Rk_cond2: "rel_preserves_inv_down Rk givenI"
```

assumes *tk_fw_sim: "fw_sim_step Rk tkc tka"*
assumes *tka_pres_inv: "inv_preserved_by_trans (lift Rk givenI) tka"*

assumes *tuc_pres_inv: "inv_preserved_by_trans givenI tuc"*

We now state that this locale is a sublocale of the base locale *framework_final*. This requires us to prove that each assumption of locale *framework_final* follows from the assumptions of *framework_kernel_fw_sim*, which is done using the *fw_sim_step_preserves_inv* lemma described in Section 4.1.3. This gives us that *final_theorem* now holds for this new locale.

sublocale *framework_kernel_fw_sim < framework_final*
⟨*proof*⟩

4.2.3 Using Locality (Integrity) for Untrusted Kernel Invocations (Step 3)

Given that the system is meant to be made of trusted and untrusted components, we can decompose the kernel transition case further, by distinguishing kernel calls on behalf of trusted users from kernel calls on behalf of untrusted ones.

For this first prototype of the framework, we are keeping the assumption that *trusted* kernel calls preserve the invariant without further decomposition. In future work, we would like to explore how to further weaken this statement by using a notion of *rely-guarantee* [5] where we can establish some common minimal condition that any trusted user transition would guarantee when doing a system call and that would be sufficient to prove invariant preservation for any such system call.

Presently, we can already simplify the *untrusted* system calls further, using the *locality* principle explained in Section 2.2. When the kernel is running on behalf of an untrusted user, the only thing that can change in the state is what this user is allowed to modify: $s =_{|W} s'$, where W is defined using a (to-be-instantiated) predicate *curr_user_authorized_writes* which extract the authority of the currently running user in a state. This relies on the kernel correctly enforcing access control on what users are allowed to do. This means that when the kernel executes on behalf of a user, it should not modify anything that this user has no write authority to. This is called the *integrity* property and is one of the properties that has been formally proven about seL4. Note that it typically only makes sense to exploit the integrity property for kernel executions on behalf of untrusted components. Trusted components typically have enough access control authority to violate the system invariant via kernel calls. This is the main reason they are trusted and that their behaviour needs to be formally verified to achieve a whole-system theorem.

Now, by definition, untrusted components are considered untrusted because they cannot violate the targeted property. This means that the global invariant *givenI*, which the property can be derived from, should not depend on anything that the untrusted user can modify. In other words, it should be reasonably easy for the user of the framework to prove that *givenI* is independent of what untrusted users can modify, which we introduce as a new, weak assumption of the framework:

indep_of_substate equal_up_to (lift Rk givenI) W where *indep_of_substate* is defined by:

```
  indep_of_substate eq_up_to P S=
    ∀s1 s2. eq_up_to S s1 s2 ⟶ P s1 ⟶ P s2
```

Here the equality of state up to a set of memory locations is defined in a separate locale, so that it can be reused later on a different state space:

```
locale locale_mem_locs =
  fixes equal_up_to :: "'memory_locs ⇒ 'a ⇒ 'a ⇒ bool"
```

For our framework, this means that we can replace the assumption about invariant preservation by *tka* by only assuming it when the current user is trusted, and adding two new assumptions: the integrity property and the fact that *givenI* does not depend on what untrusted users can modify.

```
locale framework_kernel_trusted_untrusted =
  locale_mem_locs equal_up_to
  for equal_up_to :: "'memory_locs ⇒ 'astate ⇒ 'astate ⇒ bool" +

  fixes tkc:: "'cstate × 'cstate ⇒ bool"
  fixes tuc:: "'cstate × 'cstate ⇒ bool"
  fixes givenI:: "'cstate ⇒ bool"

  fixes tka:: "'astate × 'astate ⇒ bool"
  fixes Rk:: "'astate × 'cstate ⇒ bool"

  fixes is_trusted_subj_c:: "'cstate ⇒ bool"

  fixes curr_user_authorized_writes :: "'astate ⇒ 'memory_locs"

  assumes Rk_cond1: "total_rel_on_concrete_under_cond Rk givenI"
  assumes Rk_cond2: "rel_preserves_inv_down Rk givenI"
  assumes tk_fw_sim: "fw_sim_step Rk tkc tka"

  assumes tka_trusted_pres_inv:
    "inv_preserved_by_trans_cond
       (lift Rk is_trusted_subj_c) (lift Rk givenI) tka"

  assumes integrity:
    "⟦ tka (s1,s2);
       ¬ is_trusted_subj s1;
       W = curr_user_authorized_writes s1 ⟧
     ⟹ equal_up_to W s1 s2"
  assumes givenI_indep_of_what_untrusted_can_modify:
    "⟦ ¬ is_trusted_subj s1;
       W = curr_user_authorized_writes s1⟧
      ⟹ indep_of_substate equal_up_to (lift Rk givenI) W"

  assumes tuc_pres_inv: "inv_preserved_by_trans givenI tuc"
```

Again, this locale is a sublocale of the previous one. The proof uses the case distinction lemma `inv_preserved_by_trans_cases` to distinguish the case of trusted kernel calls from untrusted ones. Then, unfolding the definition of `indep_of_substate` and `integrity` gives us that the invariant is trivially preserved by untrusted kernel calls.

sublocale *framework_kernel_trusted_untrusted < framework_kernel_fw_sim*
⟨*proof*⟩

4.2.4 Using Locality for Untrusted User transitions (Step 4)

At this stage we look at decomposing the user transitions case. Here is where we want to use the key approach of not having to prove anything about untrusted user transitions. We decompose again into trusted and untrusted cases, and we use the locality principle for untrusted user transitions: for the same reasons as before, the invariant `givenI` should not depend on untrusted user memory, which is the only thing that changes during an untrusted user transition. Our new locale therefore has two new assumptions, one stating that user transition only modify user memory in the state, and one stating that the invariant is independent of untrusted user memory. The first one can be proved once and for all when the framework is instantiated to a particular kernel, like seL4. The second one is a proof obligation that is specific to the targeted property and that should be reasonably straightforward to discharge.

```
locale framework_user_trusted_untrusted =
  locale_mem_locs equal_up_to +
  locale_mem_locs equal_up_to_c
  for equal_up_to :: "'memory_locs ⇒ 'astate ⇒ 'astate ⇒ bool"
  and equal_up_to_c :: "'memory_locs ⇒ 'cstate ⇒ 'cstate ⇒ bool" +

  fixes tkc:: "'cstate × 'cstate ⇒ bool"
  fixes tuc:: "'cstate × 'cstate ⇒ bool"
  fixes givenI:: "'cstate ⇒ bool"

  fixes tka:: "'astate × 'astate ⇒ bool"
  fixes Rk:: "'astate × 'cstate ⇒ bool"

  fixes is_trusted_subj_c:: "'cstate ⇒ bool"

  fixes curr_user_authorized_writes :: "'astate ⇒ 'memory_locs"

  fixes curr_user_mem :: "'cstate ⇒ 'memory_locs"

  assumes Rk_cond1: "total_rel_on_concrete_under_cond Rk givenI"
  assumes Rk_cond2: "rel_preserves_inv_down Rk givenI"
  assumes tk_fw_sim: "fw_sim_step Rk tkc tka"

  assumes tka_trusted_pres_inv:
    "inv_preserved_by_trans_cond
       (lift Rk is_trusted_subj_c) (lift Rk givenI) tka"

  assumes integrity:
    "⟦ tka (s1,s2);
       ¬ is_trusted_subj s1;
       W = curr_user_authorized_writes s1 ⟧
     ⟹ equal_up_to W s1 s2"
  assumes givenI_indep_of_what_untrusted_can_modify:
    "⟦ ¬ is_trusted_subj s1;
       W = curr_user_authorized_writes s1⟧
      ⟹ indep_of_substate equal_up_to (lift Rk givenI) W"

  assumes tuc_trusted_pres_inv:
    "inv_preserved_by_trans_cond is_trusted_subj_c givenI tuc"

  assumes tu_only_changes_user_mem:
  " tuc(s1',s2') ⟹
    equal_up_to_c (curr_user_mem s1') s1' s2' "
  assumes givenI_independent_of_untrusted_memory:
   "⟦ ¬ is_trusted_subj_c s1';
      untrusted_mem = curr_user_mem s1'⟧ ⟹
      indep_of_substate equal_up_to_c givenI untrusted_mem"
```

Proving that this locale is a sublocale of the previous one again uses case distinction to distinguish trusted and untrusted kernel calls, and then unfolds the definition of `indep_of_substate`.

```
sublocale framework_user_trusted_untrusted <
            framework_kernel_trusted_untrusted
⟨proof⟩
```

4.2.5 Using Abstraction for Trusted User Transitions (Step 5)

Now we are left with dealing with trusted user transitions. Here we speculate that we would have a forward simulation proof for each concrete user transition step. This could be at a different abstraction level than the kernel one (i.e. different state relation), and even different state space. The new locale then just replaces the invariant preservation at the concrete level by the preservation at the abstract level.

```
locale framework_trusted_user_fw_sim =
  locale_mem_locs equal_up_to +
  locale_mem_locs equal_up_to_c
  for equal_up_to :: "'memory_locs ⇒ 'astate ⇒ 'astate ⇒ bool"
  and equal_up_to_c :: "'memory_locs ⇒ 'cstate ⇒ 'cstate ⇒ bool" +
  fixes tkc:: "'cstate × 'cstate ⇒ bool"
  fixes tuc:: "'cstate × 'cstate ⇒ bool"
  fixes givenI:: "'cstate ⇒ bool"

  fixes tka:: "'astate × 'astate ⇒ bool"
  fixes Rk:: "'astate × 'cstate ⇒ bool"

  fixes is_trusted_subj_c:: "'cstate ⇒ bool"

  fixes curr_user_authorized_writes :: "'astate ⇒ 'memory_locs"

  fixes curr_user_mem :: "'cstate ⇒ 'memory_locs"

  fixes tua:: "'aastate × 'aastate ⇒ bool"
  fixes Ru:: "'aastate × 'cstate ⇒ bool"

  assumes Rk_cond1: "total_rel_on_concrete_under_cond Rk givenI"
  assumes Rk_cond2: "rel_preserves_inv_down Rk givenI"
  assumes tk_fw_sim: "fw_sim_step Rk tkc tka"

  assumes tka_trusted_pres_inv:
    "inv_preserved_by_trans_cond
       (lift Rk is_trusted_subj_c) (lift Rk givenI) tka"

  assumes integrity:
    "⟦ tka (s1,s2);
       ¬ is_trusted_subj s1;
       W = curr_user_authorized_writes s1 ⟧
     ⟹ equal_up_to W s1 s2"
  assumes givenI_indep_of_what_untrusted_can_modify:
    "⟦ ¬ is_trusted_subj s1;
       W = curr_user_authorized_writes s1⟧
      ⟹ indep_of_substate equal_up_to (lift Rk givenI) W"

  assumes Ru_cond1: "total_rel_on_concrete_under_cond Ru givenI"
  assumes Ru_cond2: "rel_preserves_inv_down Ru givenI"
  assumes tu_fw_sim: "fw_sim_step Ru tuc tua"
  assumes tua_trusted_pres_inv:
    "inv_preserved_by_trans_cond
       (lift Ru is_trusted_subj_c) (lift Ru givenI) tua"

  assumes tu_only_changes_user_mem:
  " tuc(s1',s2') ⟹
    equal_up_to_c (curr_user_mem s1') s1' s2'"
  assumes givenI_independent_of_untrusted_memory:
   "⟦ ¬ is_trusted_subj_c s1';
```

```
    untrusted_mem = curr_user_mem s1'⟧ ⟹
    indep_of_substate equal_up_to_c givenI untrusted_mem"
```

Proving that this locale is a sublocale of the previous one is completely analogous to the case where we abstracted kernel transitions, except that we use the conditional version of the lemma (*fw_sim_step_preserves_inv_cond*).

sublocale *framework_trusted_user_fw_sim < framework_user_trusted_untrusted* ⟨*proof*⟩

We have now reached the current state of our initial framework prototype. In the next section we summarize what the framework provides, its limitations and the future steps to validate and improve it further.

5 Conclusion and Future Directions

Summary. Our prototype framework provides, for microkernel-based, componentised systems, and for any targeted system invariant, a list of proof obligations which, once proved by the user of the framework, will imply that the invariant is preserved at the source code level of the whole system. In its current state, the framework contains three types of proof obligations. Firstly, we have three proof obligations about the kernel or system automaton, that do not depend on the specific system built on top:

- proof of functional correctness for the kernel;
- proof that the kernel enforces integrity;
- proof that user transitions may only change user memory.

These three proof obligations can be discharged once for a given kernel and the immediate next step of our project is to partially instantiate the framework with the seL4 microkernel and discharge those assumptions.

The next kind of proof obligations are assumptions about the invariant that should be relatively easy to discharge given the definition of untrusted components and targeted property:

- proof that the invariant is independent of what untrusted users can modify and their memory (i.e. that untrusted users really need not be trusted);
- proof that the invariant satisfies the totality conditions about the state relations used in the kernel's and the trusted user's functional correctness proof. These are necessary to lift the invariant preservation to an abstract level.

Finally, the last category of proof obligations are the more involved proofs about the behaviour of trusted components. By definition, trusted components can potentially violate the targeted property and hence the system invariant. We therefore need a formal model of their behaviour and a proof that they preserve the invariant, both during purely user-level transitions and during kernel calls that they initiate:

- proof of functional correctness for trusted, concrete, small-step, user transitions;
- proof that trusted kernel calls preserves the invariant

These two last assumptions are more speculative and lead to further work in improving or validating the framework, as we expand on below.

Current limitations and Future work. In terms of the framework itself, the following are the current limitations that will need to be addressed in future work:

- *initial state*: invariant preservation only makes sense if the invariant can be established once in the initial state of the system; this is part of another ongoing project (AOARD project #114070 [2]) about providing a provably correct initialiser of componentised systems: taking a formal description of a desired configuration in terms of components and communications channels and provide a initialiser code together with a proof that this code sets up the system as required. Future work thus includes using this formalised initial state of the system to prove that it satisfies the invariant.

- *trusted kernel calls*: proving that every specific kernel transition on behalf of a trusted component preserves the invariant remains a strong proof obligation. As explained in Section 4, we believe this statement could be simplified further by using a notion of *rely-guarantee* [5] style proof where we can establish some common minimal condition that any trusted user transition would guarantee when doing a system call and that would be sufficient to prove invariant preservation for any such system call. We aim to explore this as future work to improve the framework.

- *small step forward simulation for trusted components*: as already mentioned, the assumption about being able to abstract trusted user transitions for each small, concrete step, in such a way that the abstraction allows an easier proof of the invariant preservation, is still speculative. Validating that this is feasible in practice is part of our next step of exercising this framework on concrete small example systems.

Besides the current limitations of the framework, the assumptions of the integrity proof will lead to restrictions on the system policy. These include that not all policies are valid for secure systems. This is to be expected: for instance, the policy that just gives everyone access to everything is not a secure policy. The main notable assumptions of the seL4 integrity proof are that we do not permit memory sharing between security domains and make restrictions on which communication primitives are permitted between domains. In future work we are looking to relax this assumption and allow limited forms of shared memory communication. A first formal set of assumptions is detailed in our paper on the integrity proof [8]. We expect further policy restrictions to arise from our work on confidentiality and from validating our framework prototype on real system and targeted invariant.

Finally, the line of reasoning presented in this formal verification does not include timing or other side channels of the machine that user-level components may try to use for communication. By definition, side channels are those channels that are not visible in the formal model of the system. In our case, our highest-fidelity model includes the full functional behaviour of the kernel at the C-code level as well as the memory, register, and most virtual memory subsystem behaviour of the machine. It does not include caches, the TLB, or the timing behaviour of the machine. It currently also does not yet include external devices. Any of these could potentially be used as communication side channels. How far that is possible depends mainly on the hardware and the implementation of the kernel. For these we still need to rely on traditional security analyses.

Conclusion. The formal, machine-checked reasoning framework for whole-system properties presented in this report is the first concrete step in pulling together a number of key theorems about microkernel-based systems on different levels of abstraction. The framework shows that kernel-enforced integrity and isolation properties and a system design that exploits a suitable division of an architecture into trusted and untrusted components can drastically reduce the number and complexity of the proof obligations that need to be satisfied per system. The theorems in this report are automatically generated from the corresponding formal, machine-checked Isabelle/HOL theories.

References

[1] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *International Journal on Embedded Systems*, 2:239–247, 2006.

[2] June Andronick, Gerwin Klein, and Andrew Boyton. Formal system verification — extension, annual report AOARD 114070. http://www.nicta.com.au/pub?doc=5926 ISSN 1833-9646-5926, NICTA, Sydney, Australia, May 2012.

[3] June Andronick, Gerwin Klein, and Toby Murray. Final report option 1 AOARD 104105, formal system verification for trustworthy embedded systems. http://www.nicta.com.au/pub?doc=5617 ISSN 1833-9646-5617, NICTA, Sydney, Australia, November 2011.

[4] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[5] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[6] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[7] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[8] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *2nd International Conference on Interactive Theorem*

Proving, volume 6898 of *Lecture Notes in Computer Science*, pages 325–340, Nijmegen, The Netherlands, August 2011. Springer-Verlag.