Towards proving security in the presence of large untrusted components

June Andronick
NICTA, UNSW

David Greenaway
NICTA

Kevin Elphinstone
NICTA, UNSW

Abstract

This paper proposes a generalized framework to build large, complex systems where security guarantees can be given for the overall system's implementation. The work builds on the formally proven correct seL4 microkernel and on its fine-grained access control. This access control mechanism allows large untrusted components to be isolated in a way that prevents them from violating a defined security property, leaving only the trusted components to be formally verified. The first steps of the approach are illustrated by the formalisation of a multi-level secure access device and a proof in Isabelle/HOL that information cannot flow from one back-end network to another.

1 Introduction

Advances in machine-assisted theorem proving, and formal methods techniques in general, have pushed the limits of software verification to the point where it is possible to prove properties of real-world applications. The recently verified seL4 microkernel is one such example. Its 7500 lines of C code were formally proved to correctly implement a high-level abstract specification of its behaviour [7].

Formally verifying programs with sizes approaching 10 000 lines of code is a significant improvement in what formal methods was previously able to verify with reasonable effort. However, 10 000 lines of code is still a significant limit on the application of formal methods to the verification of contemporary software systems. Modern software systems, beyond very simple embedded systems, frequently consist of millions of lines of code. Thus the challenge remains as to how formal assurance can be given to real-world software systems of such size.

This paper presents our vision for how specifically targeted properties can be provably assured in very large and complex software systems. Our vision comes from the observation [1] that not all software in a large system necessarily contributes to a property of interest. For example, a game installed on a smartphone contributes nothing to the ability to make reliable phone calls. If one can assure the game is isolated from the phone call software, one can focus verification effort on the phone call software to assure reliability of calls.

The vision is to develop methodologies and tools that enable developers to systematically (*i*) isolate the software parts that are not critical to a targeted property, and prove that nothing more needs to be verified about them for the specific property; and (*ii*) formally prove that the remaining critical parts satisfy the targeted property. The key aspect of the vision is the *system-level* specification of the property of interest, and the incorporation of all critical code in an overall proof, including the kernel. A challenge will be to keep the security-critical parts or *trusted computing base* (TCB) as small and simple as possible to ensure that its verification remains tractable.

Our vision builds on, and is enabled by, the formal verification of the seL4 microkernel. Microkernel-based systems already componentise software into smaller, isolated, components for security, safety, or reliability. SeL4's verification will eventually enable provable isolation guarantees by providing correct kernel mechanisms for managing the hardware platform's memory protection mechanisms.

The remainder of the paper presents our first steps towards realising our vision for large, secure systems on seL4. We use a concrete case study of a *secure access controller* (SAC) as a representative example of a large complex system with a specific property requirement. Section 2 describes the SAC in more detail. Section 3 briefly overviews seL4, and presents a SAC design (and rationale) that is architected to minimise the TCB. Section 4 describes how to formally verify security prop-

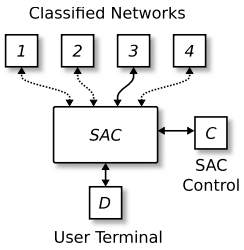

Figure 1: The SAC routes between a user's terminal and 1 of n classified networks.

erties on that architecture such that the properties still hold at the implementation level, and includes the formalisation of the information flow property targeted for the SAC and its proof using Isabelle/HOL. Finally, Section 5 looks at related work, while Section 6 concludes.

2 Case study overview

To illustrate some of the difficulties present in verifying large systems, we introduce a case study of a simple SAC device. In this scenario, a single user requires access to several independent networks of different security classifications. The user has a simple terminal connected to a network interface of the SAC. The SAC has additional network interfaces allowing it to be connected to each of the classified networks. The user only needs to access one network at a time, and selects the network through a web interface provided by the SAC on a control network interface. This setup is depicted in Figure 1.

The goal of the SAC is to route TCP/IP packets between the user's terminal and the currently selected network without allowing the information to be seen or manipulated by the other networks. The SAC must ensure that all data from one network is isolated from each of the other networks. While we assume that the user's terminal is trusted to not leak previously received information back to another network, we otherwise assume that all networks connected to the SAC are malicious and will collude.

Concrete applications of such a device can be found in the defence sector, where users frequently need to deal with data of several classifications, each of which is isolated on its own network. The traditional approach of having one terminal per classification level for each user, while clearly obeying the security requirements, is rather unwieldy.

While the requirements of the SAC are quite simple, it already presents several challenges to full system verification. In particular, the SAC requires code for (*i*) gigabit network card drivers; (*ii*) a secure web server; (*iii*) a TCP/IP stack for the web server; and (*iv*) IP routing code. Any one of these components would individually consist of tens of thousands of lines of non-trivial code that would give even the most seasoned verification engineer pause. Complicating matters further, each of the classified networks needs to both read and write data to the user's terminal at some point in time. Traditional data diodes or any security design that relies on statically partitioning resources would be incapable of providing the required functionality of the SAC. Despite these complications, our goal is to provide the required functionality while having a full system assurance that the data from the networks will remain isolated.

3 Designing for the Vision

For our case study, our property of interest is an access-control-based security property. Verifying such a property for the large body of code needed to implement the functionality required by the SAC is far beyond the abilities of current verification methods.

To overcome this we split the code of a large system into two classes: *trusted* code, implementing security-critical functionality, and *untrusted* code which we assume is malicious, avoiding the need to reason about its precise implementation. For such a split to be possible, we need some mechanism that allows such untrusted code to be securely isolated.

Our work uses the seL4 microkernel to provide such isolation. SeL4 is a small operating system kernel of the L4 family designed to be a secure, safe, and reliable foundation for a wide variety of application domains [11]. Its C implementation has been formally proved to match its functional specification [7], making it a key foundation of our goal for full system assurance. As a microkernel, it provides a minimal number of services to applications: abstractions for virtual address spaces, threads and inter-process communication (IPC).

SeL4 uses a capability-based access-control model. All memory, devices, and microkernel-provided services require an associated *capability* (access right) to utilise them [3]. The set of capabilities a component possesses determines what a component can directly access. SeL4 enforces this access control using the hardware's memory management unit (MMU). Additionally, seL4 allows device drivers to be isolated by using the I/O MMU functionality present on recent x86 processors. The I/O MMU allows the kernel to control what areas of physical memory each hardware device can access via *direct memory access* (DMA), preventing malicious hardware devices (or, more specifically, malicious software controlling such hardware devices) from bypassing seL4's access control mechanisms.

The access control mechanism of seL4 allows systems

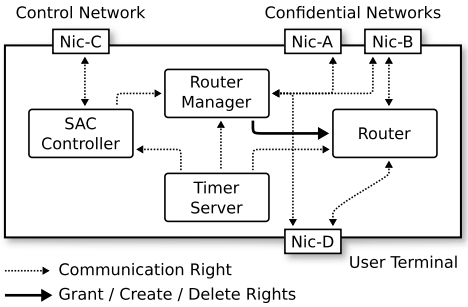

Figure 2: High-level component breakdown of the SAC design. The router manager is the only trusted component in the system, as no other component has simultaneous access to both NIC-A and NIC-B.

to be broken into smaller independent components, each with its own set of access rights. This split of components forms the system's high-level *security architecture*. The set of capabilities we provide to each component forms the system's *capability distribution* which precisely defines overt communication amongst components and hardware, and thus can be used as the basis of a security analysis of the system. Covert communication channels, such as timing channels, would have to be analysed by other means.

Components that do not possess any capabilities that may be used to violate the system's security policy need not be trusted, and can be implemented without requiring verification. Components that *do* have sufficient capabilities to violate the system security policy become part of the TCB (along with the seL4 kernel itself), and require verification. For example, in our SAC case study, any component that possesses a capability to a network card connected to a classified network while simultaneously having access to information from another classified network would need to be trusted, and hence verified.

In our experience, designing a secure system is an iterative process: (*i*) a high-level security architecture is proposed, coarsely breaking the system down into components; (*ii*) a capability distribution is determined by applying the principle of least privilege for the design [10]; and (*iii*) this capability distribution is analysed to determine which components have sufficient rights to violate the desired security policy (hence becoming part of the system's TCB). The resulting design may be further refined via re-iterating to reduce the size of the TCB, and thus ease verification effort.

We return to the case study to illustrate how this design process applies to the SAC. For simplicity of explanation, we assume that the SAC only needs to multiplex two classified networks, NIC-A and NIC-B. The user's terminal is connected to NIC-D, while the SAC is controlled through a web interface provided on NIC-C.

To avoid trusting (and thus verifying) large bodies of code such as network stacks, we architect the system with an untrusted *router* component. This component is given access to NIC-D and either NIC-A or NIC-B, and is responsible for routing between the two networks. The component has two additional parts: read-only access to its own initialisation code and additional read-write memory required by it at run-time.

A second trusted component, the *router manager*, possesses capabilities to all three of NIC-A, NIC-B and NIC-D. When the SAC needs to switch between networks, the router manager first deletes any running router component, clears the router's read-write memory, and sanitises the hardware registers and buffers of NIC-D (to prevent any residual information from inadvertently being stored in it). Such sanitisation requires a detailed knowledge of the the network card hardware (to ensure that all potential storage channels are cleared), but is expected to be significantly simpler than an implementation of a full driver for the card. The router manager will then recreate the router, grant it access to its read-only code and read-write memory, and grant it access to NIC-D and either NIC-A or NIC-B as required. This allows the router to switch between NIC-A and NIC-B without being capable of leaking data between the two.

A third untrusted component, the *SAC controller*, provides a web interface to the control network on NIC-C. The router manager is given a read-only communication channel to the SAC controller, which is used to instruct the router manager to restart the router with rights to the other classified network.

Finally, to avoid components sharing the system's timing hardware (thus creating a communications channel between them), a fourth untrusted *timer server* component is granted access to the system clock and provided with a write-only communication channel to each of the other components. It broadcasts a regular timer tick to the other components, allowing each to internally track time, required by modern network card drivers and TCP/IP implementations.

This design, shown in Figure 2, only requires the router manager to ever have access to both NIC-A and NIC-B simultaneously. While this means that the router manager component becomes part of the system's TCB, it allows us to leave all other components in the system untrusted, significantly easing the burden of verification.

Our implementation of this design uses GNU/Linux to implement the router and SAC controller components. The SAC controller's webserver is implemented using '`mini_httpd`', while the Linux kernel itself provides functionality for routing, the TCP/IP networking stack and drivers for the network cards. The Linux kernel alone consists of millions of lines of code, much of which

would become part of the TCB if used directly. By utilising the access control features of seL4 and designing the system to isolate this functionality, we were able to reduce the run-time TCB of the SAC to just the router manager (approximately 1500 lines of code) and the seL4 kernel (approximately 7500 lines of code), just under 9000 lines in total.

4 Formal verification of security properties

While the previous section described informally how a secure system such as the SAC might be designed to reduce the size of its TCB, this alone does not provide any guarantees about our desired security property. This section describes a process that allows us to formally prove that the final system implementation obeys the property, and describes our progress on this vision by describing the first few steps of the proof on our SAC case study.

As in seL4's correctness proof, we focus on verification of the initialized C code, assuming the correctness of hardware, compiler, assembly and booter (the two latest being on-going work).

This verification approach is illustrated in Figure 3. Once a system has been broken into components with an initial capability distribution defined (labelled ① in the figure) and trusted components in the system have been identified (labelled ② in the figure), we must then:

1. Prove that this partition is sound. That is, we must prove that untrusted components are incapable of violating the targeted security property of the system. This is done by describing the behaviour of trusted components (labelled ③ in the figure), and modelling untrusted components as capable of carrying out any series of actions authorised by the set of capabilities they possess. If, under these assumptions, a proof of security succeeds, nothing further needs to be proven about the untrusted components.

2. Prove that the code of the run-time TCB (i.e., the trusted components and underlying kernel) correctly implements the security model used for the proof. This involves taking the simple model used to perform the proof in step 1, and then refining it (possibly via several increasingly more precise models), down to the final system's implementation.

The second step involves three tasks, most of them being on-going or future work. First, we need to prove that the kernel implementation refines its security model. Building on seL4's proof of correctness reduces this task to proving that the high-level specification the kernel implements refines the security model. This is on-going work. The second task is to prove that the trusted components' implementations refine their formal behavior. This has not been done for the SAC system but our experience from the kernel verification and the framework built for such refinement give us confidence that this task is feasible. The last task consists in proving that the initial capability distribution in the system implementation satisfies the abstract security architecture. We have defined a capability distribution language, called capDL [8], with a formal semantics that aims to be used to automatically and formally link a user-defined capability distribution description of the system to both an initial implementation state and an abstract security architecture.

The remainder of this section describes the first step of the two listed above in detail, illustrating them with our SAC case study.

4.1 Notation

We briefly introduce the notation used for the remainder of this paper. Our meta-language Isabelle/HOL conforms for the most part with normal mathematical notation.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ. The option type **datatype** $'a\ option = None \mid Some\ 'a$ adjoins a new element *None* to a type $'a$. Function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$ and $f(x \mapsto y)$ stands for $f\ (x := Some\ y)$.

Isabelle supports tuples with named components. For instance, we write **record** $point = x :: nat,\ y :: nat$ for the type *point* with two components of type *nat*. If p is a *point*, a possible value for p is notated $(\!|x=5,y=2|\!)$. The term $x\ p$ stands for the x-component of p. Updating p from a current value $(\!|x=5,y=2|\!)$, with the update notation $p(\!|x:=4|\!)$, gives $(\!|x=4,y=2|\!)$. Finally, the keyword **types** introduces a type abbreviation.

4.2 Underlying access control model

From a security point of view, the operations provided by the kernel can be reduced to seven possible operations: read, write, create, delete, remove, grant, revoke and four corresponding access rights: read (r), write (w), create (c) and grant (g). The seL4 kernel supports more operations, but purely from the security perspective, each of them can be reduced to a sequence of these seven. For instance an IPC receive can be reduced to a read, while an IPC send can be reduced to a write.

The first five operations allow a component to read or write from another component, to create a new component, to delete an existing component, or to remove an

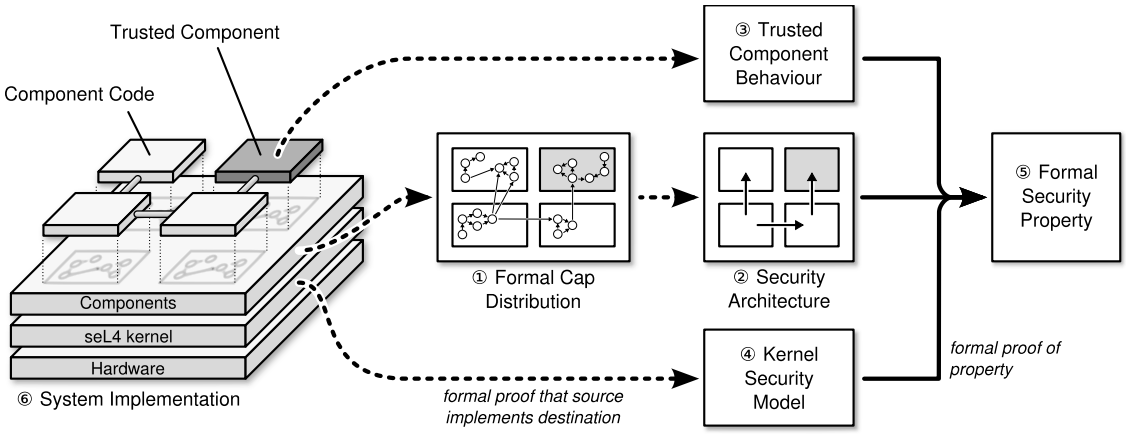

Figure 3: Full-system verification approach for seL4-based system

existing capability. All of these operations require the acting component to hold the correct capability to the target entity. The last two operations, grant and revoke, enable a component to delegate one of its capabilities to another component or to withdraw one or more capabilities from other components. Whereas the grant operation requires an explicit capability for authorisation as above, the revoke operation is authorised implicitly. Each component may revoke any capability it has created and all copies it has granted as long as it still holds the original capability. The kernel internally tracks this create/grant relationship in an internal book-keeping mechanism, and authorises revoke and delete operations accordingly.

In the following, we define the state space the application level entities will operate in, together with the transitions on this state space that the seL4 kernel allows.

4.2.1 State Space Model

Our model is largely inspired by seL4's security model developed in previous work [4, 5, 2]. In this previous work, all the kernel objects (active and inactive) are modelled as *entities*, and the state space only stores the capabilities each entity in the system has access to. In other words, it abstracts away from all application-local or kernel-internal storage, and instead concentrates on how capabilities—and therefore access to information—are distributed throughout the system. The only extension made here is regarding storage: for certain security properties we may need to track additional state.

For instance, in our case study, the property we are interested in for the SAC is the absence of explicit information flow, i.e., confidential data being explicitly read by an external entity. For simplicity, we only aim to prove that there is no information flow from NIC-A to NIC-B (the property being symmetric). The *confidential data* is therefore the data coming from NIC-A and the *external entity* that should not obtain any information from this confidential data is NIC-B. The approach taken is to tag the data coming from NIC-A as confidential. This means that we give any entity with storage (memory and network cards) a flag denoting whether it could possibly contain data from NIC-A. Entities that have this flag set are called *contaminated*. In their initial state, no entity, other than NIC-A, is contaminated. Each time an entity reads from a contaminated entity, it too becomes contaminated. Likewise, if a contaminated entity performs a write operation to another entity, the target becomes contaminated. The goal then is to prove that the whole SAC can never reach a state where NIC-B is contaminated.

Entities are therefore represented by the set of capabilities they hold and their "contamination status":

record *entity* =
  caps :: *cap set*
  contam :: *bool*

Each capability contains a reference to an entity it grants rights to, and the set of access rights it provides:

datatype *right* = *Read* | *Write* | *Grant* | *Create*
record *cap* =
  entity :: *entity-id*
  rights :: *right set*

Both the capability set and each entity's contamination state can dynamically change. The state of the system at a given point is a function from entity identifiers to entities. We model the fact that not all entity identifiers are mapped to entities by using the option type:

datatype *entity-id* =
  SacController | *NicA* | *NicB* | *NicC* | *NicD*
  | *RouterManager* | *Router* | *RouterMem* | *RouterCode*
  | *Timer* | *TimerChip* | *UnknownEntity nat*.

types *state* = *entity-id* ⇒ *entity option*

Note that the entity's contamination status can be gen-

eralised to other kinds of storage information required by a label-based security property.

4.2.2 System operations

The possible basic transitions on this state space are described by the kernel operations available to components.

We do not model the revoke operation in its general case here, but instead represent it by the specific sequence of remove operations that eventually take place. The operations' formalization is the following:

datatype *sys-op* =
  SysRead cap | *SysWrite cap bool*
| *SysCreate cap* | *SysGrant cap cap*
| *SysDelete cap* | *SysRemoveSet cap* (*cap set*)

All of the operations take a capability pointing to the targeted entity. In the case of *SysRead c* for instance, the entity performing the operation is reading from the entity referred to by the capability *c*. The operation will only be allowed by the kernel if the capability *c* is held by the entity performing the operation and includes at least the read right. *SysGrant* also takes the capability to be granted and *SysWrite* takes a boolean flag which is true if the write operation is a *flush* operation, removing an entity's contamination flag. Such an operation is required to model the router manager's sanitisation of NIC-D when the network is being switched. The final operation *SysRemoveSet* removes a set of capabilities.

The authorisation check for all the system operations is summarised in the function *legal s e sysop* defining the conditions for entity *e* to perform operation *sysop* in state *s*. For instance:

legal s e (*SysRead c*) =
  (*is-entity s e* ∧ *is-entity s* (*entity c*) ∧
  c ∈ *entity-caps-in-state s e* ∧ *Read* ∈ *rights c*)

where *is-entity* ensures that entity *e* is defined in *s* and *entity-caps-in-state* retrieves the capabilities held by *e*.

4.2.3 State transitions

We now look at how the state changes for each operation. This is modelled by the function *step s e sysop* defining the resulting state after the entity *e* has performed operation *sysop* on state *s*. For instance, we model both reads and writes as a *write-operation* (with the direction of the write switched for reads):

step s e (*SysWrite c b*) = *write-operation e* (*entity c*) *b s*
step s e (*SysRead c*) = *write-operation* (*entity c*) *e False s*

where *write-operation* is defined as follows:

write-operation source target is-flush ss ≡
  (*case ss target of*
    Some target-entity ⇒
      ss(*target* ↦ *target-entity*(| *contam* :=
        ((*is-contam ss target* ∨ *is-contam ss source*)
          ∧ ¬*is-flush*)|))
  | - ⇒ *ss*)

The other operations are defined similarly and are used to define *step*. A *legal-step* is defined as a step of the system that only takes place if it is legal. If the operation can not be performed (because the thread attempting the operation doensn't have an appropriate capability, for instance), the operation silently fails and the system state remains unchanged:

legal-step s e-id sysop ≡
  if legal s e-id sysop then (*step s e-id sysop*) *else s*

This model, for simplicity, does not allow threads to determine if an operation failed; trusted threads need to ensure that they have the correct resources before attempting any security-critical operation.

4.3 Component-level model

So far we have described the states and transitions of the underlying kernel which the components will run on. This model is used to describe the components' behavior as sequences of instructions, where each instruction is a *step* modifying the global state of the system. As explained earlier, we model only the trusted components' behavior. No restriction at all is placed on the untrusted components, and they will be correctly implemented by any concrete program code. Their behaviour is only constrained by the authority they are given via capabilities. In our case study this means that the router instance, when it has received the capabilities to its network cards, will be able to attempt any behaviour, but the kernel will only allow access to the two network cards (NIC-A and NIC-D, say) it possesses capabilities to at this point. If we can show that the system is secure with this unconstrained behaviour, it will also be secure with any specific implementation of the router components.

While the specification of untrusted components is simple, the specification of trusted components requires more care. We rely on specific behaviour of the trusted component for the security of the overall system. In our case study, we rely on the router manager to execute specific operations in a specific order, such as creating the router instance, granting capabilities, revoking capabilities, flushing the network cards, etc.

We model the program of such trusted entities as a list of instructions, each of which either performs a kernel operation (*SysOp*) or changes the program counter of the entity (*Jump*). To avoid needing to reason about implementation details of trusted entities, flow control (such

as 'if' and 'case' statements) is modelled by non-deterministic choice, which itself is modelled by having the *Jump* instruction accept a list of targets. Untrusted entities may perform any operation they wish, so are modelled with a program consisting of an *AnyOp* instruction, representing any legal kernel operation:

datatype *instruction* =
  SysOp sys-op
 | *Jump nat list*
 | *AnyOp*
types *program* = *entity-id* ⇒ *instruction list*

To model the behavior of the whole system, we need to represent the fact that entities run concurrently. We model this behaviour by considering all possible interleavings of instructions between entities. For this, we keep track of a program counter for each component in an additional program counter state:

record *sys-state* =
 sys-entity-st :: *state*
 sys-pc-st :: *entity-id* ⇒ *nat*

One execution step of the whole system consists of non-deterministically choosing any existing active entity and running its current instruction (specified by its program counter). This models the seL4 kernel scheduler. If the current instruction is *AnyOp*, then we pick any arbitrary operation that is legal in the current state for this entity and execute it. We thus get a safe over-approximation of all possible execution traces of the system.

To model a single step of a particular entity *e-id*, we look at the instruction at that entity's program counter. If it is a *SysOp* or *AnyOp* operation, it is executed using *legal-step* producing a new state of the system. If the instruction is a *Jump* operation, the model non-deterministically updates the current entity's program counter to one of the values in the list *loffset*. In both cases, the model requires that the entity performing the instruction exists and that the entity's program counter is within the bounds of its program:

inductive
 entity-operation :: *entity-id* ⇒ *sys-state* ⇒ *sys-state* ⇒ *bool*
where
 entitySysOp:
 ⟦ *ss* = (| *sys-entity-st* = *s*, *sys-pc-st* = *pc* |);
  is-entity s e-id; *pc e-id* = *e-pc*;
  sys-program e-id = *e-prog*; *e-pc* < *length e-prog*;
  e-prog ! *e-pc* = *SysOp oper* ∨ *e-prog* ! *e-pc* = *AnyOp*;
  s' = *legal-step s e-id oper*;
  new-pc = (*e-pc* + *1*) *mod* (*length e-prog*);
  ss' = (|*sys-entity-st*=*s'*,*sys-pc-st*=*pc*(*e-id*:=*new-pc*)|)⟧
  ⟹ *entity-operation e-id ss ss'*
 | *entityJump*:
 ⟦ ... (∗ *as above* ∗)

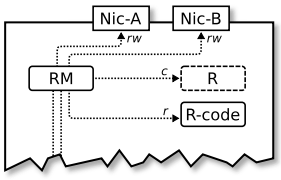

Figure 4: SAC initial state (partial)

 e-prog ! *e-pc* = *Jump loffset*;
 new-pc ∈ *set loffset*;
 ss' = (|*sys-entity-st*=*s*,*sys-pc-st*=*pc*(*e-id*:=*new-pc*)|)⟧
 ⟹ *entity-operation e-id ss ss'*

A single execution step of the whole system is then modelled by the relation $ss \rightarrow ss'$ which is true if ss' is a possible resulting system state after executing the current instruction of any existing active entity in the system state *ss*. An execution $ss \rightarrow^* ss'$ is then defined as a sequence of execution steps.

We have now defined execution and implicitly with the relation above all possible execution traces of the system.

Instantiation to a given system. This formalisation of seL4-based systems' behavior can be instantiated to a specific componentised system like the SAC. This is done by defining the initial capability distribution for this system and the program of each of its trusted components. For instance, the initial state for the SAC system (partially illustrated in Figure 4) is modelled as:

SAC-startup≡ (|*sys-entity-st*=*SAC-init-state*, *sys-pc-st*=λ*x*. *0*|)

where *SAC-init-state* defines the initial capability set for each component, together with their contamination status (of which all components other than NIC-A are initially uncontaminated). For instance, the router manager's initial state looks like:

RM0 ≡ (| *caps* = { *cap-RW-to-NIC-A*, ...}, *contam* = *False* |)

where we take the convention that the name of each cap is of the form: *cap-*<*rights*>*-to-*<*target-entity*>, as in:

cap-RW-to-NIC-A ≡ (| *entity*=*NicA*, *rights* = {*Read*, *Write*} |)

Each trusted component's behaviour is modelled as a sequence of instructions. For instance, the router manager in our case study will be formalized as follows.

RM-prg ≡
 [(∗ *00*: *Wait for command*, *delete router manager*. ∗)
 SysOp (*SysRead cap-R-to-SAC-C*),
 SysOp (*SysRemoveAll cap-C-to-R*),
 SysOp (*SysDelete cap-C-to-R*),
 SysOp (*SysWriteZero cap-RW-to-NIC-D*),
 ...

Jump [*0*, *10*, *19*],
(∗ *10*: *Setup router between NIC−A and NIC−D*. ∗)
SysOp (*SysCreate cap-C-to-R*),
SysOp (*SysNormalWrite cap-RWGC-to-R*),
SysOp (*SysGrant cap-RWGC-to-R cap-RW-to-NIC-A*),
SysOp (*SysGrant cap-RWGC-to-R cap-RW-to-NIC-D*),
SysOp (*SysGrant cap-RWGC-to-R cap-R-to-R-code*),
...]

The *sys-program* function that associates a program to each component (used *entity-operation*) is defined as:

sys-program eid ≡
if (*eid* = *RouterManager*) *then RM-prg*
else if (*eid* ∈ *untrusted-entities*) *then* [*AnyOp*] *else* []

where *untrusted-entities* for the SAC consist of *SacController*, *Router*, *Timer* and where the inactive entities (such as the network cards) are associated with empty programs.

4.4 Security property proof

With the model described, we can now formally state the security property we are targeting for our SAC case study. The property we are interested in is the absence of explicit information flow. As explained earlier, we model the fact that NIC-B cannot read information from NIC-A in a given state as NIC-B not being contaminated. In particular, we state that in any state that the SAC can reach starting from its initial state, NIC-B is not contaminated with data from NIC-A:

lemma *sacSecurity*: ⟦ *SAC-startup* $\rightarrow^*$ *ss'* ⟧ ⟹
¬ *is-contaminated* (*sac-entity-st ss'*) *NicB*

The proof relies on showing an invariant always holds on the state of the SAC. The invariant insists that: (*i*) Only NIC-A, the router (and associated components) and NIC-D ever become contaminated; (*ii*) The capabilities held by each component is limited to a small, secure set; (*iii*) The router doesn't have capabilities to both NIC-A and NIC-B at the same time; (*iv*) The router doesn't have a capability to NIC-B while any component it can access is contaminated; (*v*) The capabilities held by the router manager at every point of time is sufficient to allow it perform its job of deleting the router and sanitising NIC-D; (*vi*) All entities other than the router always exist; and finally (*vii*) That certain conditions about the state of the router hold when the router manager's program counter is at particular values.

The last invariant is the most intricate, and is required to show that the system remains in a well-known state while the router manager is mid-way through deleting or creating the router instance. For instance, when the router manager's program counter points to an instruction granting the router access to NIC-B, we must know that the router is in an uncontaminated state, which can only be established because the router manager earlier deleted the router, and hasn't provided it with any caps to NIC-A since.

The final security property follows directly from the invariant, which states that NIC-B will always be uncontaminated.

5 Related Work

The idea of using system architectures to ensure security by construction, relying on basic kernel mechanisms to separate trusted from untrusted code is widely explored in the MILS (multiple independent levels of security and safety) space [1]. In the context of formal analyses of capability-based software, Spiessens [12] developed the formal language *Scoll* to model the behaviour of trusted components, together with a model-checker for that language to check capability-based software.

Murray [9] builds on Spiessens' concepts, but uses a CSP, for which model-checking tools already exist. Its main contribution is to extend the kind of properties that can be expressed to also include noninterference style information flow properties (under the assumption that the capability system that the software is running on does not expose covert channels between unconnected objects) and liveness properties under fairness assumptions.

Both Spiessens' and Murray's work explicitly prove that it is safe to take multiple entities and model them as a single entity that possesses the union of their capabilities and exhibits the union of their behaviours. This idea is also part of our vision, with the addition of a capability abstraction (the capability distribution in Figure 3 to reason about typed capabilities upon kernel objects, whereas at the security architecture level, the simple model of read, write, create, grant capabilities between components is used). However, the proof that it is safe to aggregate entities in this way is part of our future work.

To the best of our knowledge, our work is the first to use interactive theorem proving rather than model-checking to verify capability based systems by modelling trusted components' behaviour. We also modelled the SAC using the SPIN model checker [6] for comparison. Although the proof effort was reduced from around six weeks (by an inexperienced Isabelle/HOL user) to less than one day, our model quickly reached a size that was beyond the abilities of SPIN to verify in a reasonable amount of time and memory. In particular, we could only verify our final system design by making simplifying assumptions in the SPIN model. The other main benefit of using a theorem prover is being provided with

a framework to prove the refinement between the security architecture and trusted components' behaviour, and the system implementation. Assuring label-based security properties about the actual implementation of real-world system is the real added value of the framework we propose. Investigating how interactive theorem proving and model checking can be combined in a way that gives the flexibility of the former with the ease-of-use of the latter is part of our future work.

6 Conclusion

In this paper we have presented our vision of how large software systems consisting of millions of lines of code can still have formal guarantees about certain targeted properties. This is achieved by building upon the access control guarantees provided by the verified seL4 microkernel and using it to isolate components such that their implementation need not be reasoned about.

We have demonstrated in our SAC case study how careful design and componentisation of a large system can be used to reduce the run-time TCB from millions of lines of code to just under 9000. Additionally, we have modelled the design of the SAC and shown that the modelled system fulfills its security goal of isolating data between different networks.

What still remains is connecting the model used to prove security of the system with the actual implementation. In particular, we must still show that (*i*) the C implementation of trusted components in the SAC refine the behaviour modelled in the security proof; and (*ii*) that the kernel operations in the security proof correctly model the actual behaviour of the seL4 kernel. The verification success of the seL4 kernel, with its C code shown to implement its functional specification, gives us confidence that both of these tasks are feasible. Carrying out this verification effort forms part of our ongoing work.

Acknowledgements The authors would like to thank Gerwin Klein, Toby Murray and Simon Winwood for their advice, feedback and contributions to this paper, and Xin Gao for his contributions to the proof.

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 material is in part based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-09-1-4160. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

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] A. Boyton. A verified shared capability model. In G. Klein, R. Huuck, and B. Schlich, editors, *4th SSV*, volume 254 of *ENTCS*, pages 25–44, Aachen, Germany, Oct 2009. Elsevier.

[3] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

[4] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. Technical Report NRL-1474, NICTA, Oct 2007. Available from http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf.

[5] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *VSTTE 2008*, volume 5295 of *LNCS*, pages 99–114, Toronto, Canada, Oct 2008. Springer.

[6] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[7] 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, Big Sky, MT, USA, Oct 2009. ACM.

[8] I. Kuz, G. Klein, C. Lewis, and A. Walker. capDL: A language for describing capability-based systems. In *1st APSys*, New Delhi, India, Aug 2010. To appear.

[9] T. Murray. *Analysing the Security Properties of Object-Capability Patterns*. PhD thesis, University of Oxford, 2010.

[10] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[11] seL4 Website. http://ertos.nicta.com.au/research/sel4/, Jun 2010.

[12] A. Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, February 2007.