# Reasoning About Information Flow Security of Separation Kernels with Channel-based Communication

Yongwang Zhao[1,2], David Sanán[1], Fuyuan Zhang[1], and Yang Liu[1]

[1] School of Computer Engineering, Nanyang Technological University, Singapore
[2] School of Computer Science and Engineering, Beihang University, Beijing, China

**Abstract.** Assurance of information flow security by formal methods is mandated in security certification of separation kernels. As an industrial standard for separation kernels, ARINC 653 has been complied with by mainstream separation kernels. Security of functionalities defined in ARINC 653 is thus very important for the development and certification of separation kernels. This paper presents the first effort to formally specify and verify separation kernels with ARINC 653 channel-based communication. We provide a reusable formal specification and security proofs for separation kernels in Isabelle/HOL. During reasoning about information flow security, we find some security flaws in the ARINC 653 standard, which can cause information leakage, and fix them in our specification. We also validate the existence of the security flaws in two open-source ARINC 653 compliant separation kernels.

## 1 Introduction

Separation kernels [26] create a secure environment by providing temporal and spatial separation of applications and ensure that there are no unintended channels for information flows between partitions other than those explicitly provided. Separation kernels decouple the verification of applications in partitions from the verification of the kernels themselves. They are often sufficiently small and straightforward to allow formal verification of their correctness. Assurance of information flow security [28] by formal methods is mandated in Separation Kernel Protection Profile (SKPP) [21] and certifying separation kernels to highest Common Criteria evaluation levels (EAL 6 or 7) is always accomplished by formally verifying information flow security.

Traditionally, security and safety of critical systems are assured and certified by using two kinds of separation kernels respectively, such as VxWorks 653 [3] for safety-critical systems and VxWorks MILS [4] for security-critical systems. A trend in this field is to integrate safe and secure functionalities into one separation kernel. For instance, PikeOS [2], LynxSecure [1] and open-source XtratuM [16] are designed to support both safety critical and security critical solutions. As an industrial standard for safety-critical separation kernels, ARINC 653 [5] aims at improving safety and certification process of safety-critical systems, which has been complied with by the mainstream separation kernels such as PikeOS, VxWorks 653 and XtratuM. Therefore, in order to develop ARINC 653 compliant secure separation kernels, it is necessary to assure security of the functionalities defined in ARINC 653. A security verified specification and its mechanically checked proofs of ARINC 653 are significant for the development and certification of separation kernels.

In separation kernels, Inter-Partition Communication (IPC) is a major mechanism to implement controlled information flows, but if the mechanism is not well designed, IPC can also contain covert channels [18] to leak information between applications. ARINC 653 defines the functionalities and services of a *channel-based communication* mechanism for IPC. Although formal specification [8, 30–32] and verification [13, 25, 33, 12, 19, 9, 29] of information flow security on separation kernels have been widely studied in academia and industry, information flow security of separation kernels with ARINC 653 channel-based communication has not been studied to date. To the best of our knowledge, this paper is the first effort on this topic.

In this paper, we present a formal specification and its security proofs[3] of separation kernels with ARINC 653 channel-based communication in Isabelle/HOL [22]. In detail, the technical contributions of this work are as follows.

1. We provide a mechanically checked formal specification which comprises a generic execution model for separation kernels and an event specification for ARINC 653. We introduce two security domains: a *scheduler* and a *message transmitter*, and their security policies according to the characteristics of scheduling and IPC of separation kernels. The event specification models all IPC services defined in ARINC 653 (Section 3).
2. We define a set of information flow security properties and an inference framework to sketch out the implications between security properties. We provide the security proofs to indicate information flow security of the specification (Section 4).
3. We find some security flaws, i.e., covert channels to leak information, in the ARINC 653 standard when proving our original specification that is completely compliant with ARINC 653, and fix them by a redesign of the specification. We also validate the existence of the security flaws in two open-source ARINC 653 compliant separation kernels, i.e., XtratuM and POK [10]. The cost of this work is in total 8 person-months (Section 5).

## 2 Challenges and Approach Overview

This section introduces the challenges in this work and the overview of our approach.

**Challenges** The challenges of this work are as follows.

1. *High complexity of the ARINC 653 standard*: the standard specifies the system functionality of separation kernels using more than 40 pages of informal descriptions and standardized services using more than 60 pages. As the core part for channel-based communication, the IPC takes more than 20 pages and defines a complicated communication mechanism including queuing and sampling modes, channel buffers and port control.
2. *Enormous efforts needed by formal verification of information flow security*: As a sort of hyperproperties [6], it is difficult to automatically verify information flow security on separation kernels so far and formal verification needs an exhausting effort. There exist different sorts of information flow security (e.g., in [27, 28, 23, 20]) and relationship of them on ARINC 653 separation kernels has to be clarified for security assurance and certification to reduce the verification effort.

[3] The specification and proofs are available at "http://securify.sce.ntu.edu.sg/skspecv1/"

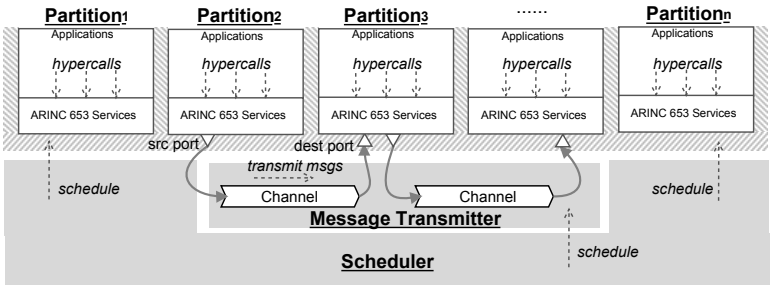


**Fig. 1.** Architecture of the Target System

**Analysis of the Target System** In order to address *Challenge 1*, we are more concerned on basic functionalities of separation kernels and reduce components not related to information flow security, such as hardware interface in ARINC 653. ARINC 653 uses the inter-partition flow policy [15] in which communication ports and channels are associated with partitions, and all processes in a partition can access the ports configured for this partition. Moreover, some hypervisor based separation kernels, such as XtratuM, manage partitions, but processes in a partition are invisible to the kernel. Thus, we omit the concept of "process" and intra-partition communication between processes in ARINC 653 in the formal specification. The target system to be formally specified and verified is illustrated in Fig. 1.

Since the latest version of ARINC 653 [5] is targeted at single-core processing environments, our work considers single-core separation kernels and assumes there is no in-kernel concurrency as the same as in [19]. Many separation kernel implementations only allow blocking partitions by means of invoking a "partition management" hypercall, we prohibit blocking partitions in communication events.

**Analysis of Information Flow Security** Traditionally, language-based information flow security [28] handles only two-level domain: *High* and *Low*. The data of programs are assigned either *High* or *Low* labels. Security hereby means that variations of *High*-level data should not cause a variation of *Low*-level data. When verifying information flow security of separation kernels, the only available information is the set of configured partitions, local configurations of partitions, and the set of possible events (hypercalls) partitions can invoke. There is not any concrete information about private data of partitions. Thus, it is not possible to classify the data as *High* or *Low*. Moreover, the inter-partition flow policy of ARINC 653 is an intransitive policy [27], which cannot be addressed by traditional language-based information flow security. This problem is solved in [27], where noninterference is defined following a state-event based approach that considers intransitivity. In order to clarify different definitions on separation kernels, we formalize language-based information flow security in a state-event style and reason about the relationship of them.

Traditional formulations in the state-event based approach for information flow security assume a static mapping from actions to domains, such that the domain of an action can be determined solely from the action itself [27]. However, in separation k-

ernels that mapping is dynamic. When a *hypercall* occurs, the kernel must consult the kernel scheduler to determine which partition is currently running, and the currently running partition is the domain of the hypercall. In our specification, we define the *scheduler* security domain for kernel scheduling, which cannot be interfered by any other domain to ensure that the scheduler security domain does not leak information via its scheduling decisions. Since ARINC 653 only defines the channel-based communication services using ports and leaves the implementation of message transmission on channels to underlying separation kernels, we define the *message transmitter* security domain, for message transmission. The transmitter also decouples message transmission from the scheduler to ensure that the scheduler is not interfered by partitions.

**Analysis of the Specification and Verification Approach** Since separation kernels usually support the deployment of partitions which are unknown in advance, it is well suited to use logical reasoning by induction for formal verification. By following the successful experiences of applying Isabelle/HOL in seL4 [19] and PikeOS [31, 32], we use Isabelle/HOL in this work.

The verification overview of our work is briefly shown in Fig. 2. In order to simplify the verification, we decompose the specification into two parts: an execution model for separation kernels with channel-based communication and an event specification for ARINC 653. The execution model defines basic components and a state machine of separation kernels. The event specification uses Isabelle/HOL functions to define the state changes when an event occurs. These concrete functions are invoked by the execution model. This decomposition leads to two-step proofs of information flow security. We first define a set of information flow security properties and provide an inference framework for them on the execution model. In the second step, we define a set of *concrete unwinding conditions* on the concrete functions. Satisfaction of the concrete unwinding conditions implies that the events satisfy the classical unwinding conditions, and thus shows information flow security of our specification. The decomposition of the specification and its proofs improves their reusability for subsequent specification refinement and development of implementations, and thus reduces the verification effort.

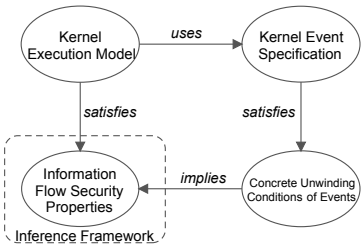


**Fig. 2.** Verification Overview

## 3 Formal Specification

In this section, we first introduce the kernel execution model including basic components and state-based kernel execution. Then, we present the event specification. Finally, we discuss the correctness of the formal specification.

### 3.1 Basic Components

According to Fig. 1, basic components include security domains, security policies and communication components. All these components are statically configured in ARINC 653 compliant separation kernels.

**Security Domains and Policies** As illustrated in bold and underlined in Fig. 1, the security domains are the scheduler, the transmitter, and the defined partitions. In order to discuss information flow policy, we assume a reflexive relation $\rightsquigarrow$ that specifies the allowable information flows between domains. If there is a channel from a partition `a` to a partition `b`, then `a` $\rightsquigarrow$ `transmitter` and `transmitter` $\rightsquigarrow$ `b` since we use the transmitter as the message intermediator. Since the scheduler can possibly schedule any domain, we define in the security policy that `scheduler` $\rightsquigarrow$ `d` for any domain `d`. The noninterference relation $\not\rightsquigarrow$ is the complement relation of $\rightsquigarrow$ that asserts no information flow outside of $\rightsquigarrow$.

**Communication Components** As illustrated in Fig. 1, IPC is conducted via messages on channels, which are defined by an abstract type `Message`. Partitions have access to channels via *ports* which are the endpoints of channels. A *channel* links partitions and is a logical link between one source port and one or more destination ports. It also specifies the mode of transferring messages, which can be *queuing* or *sampling* mode. The **datatype** `Channel_Type` and `Port_Type` define these two components.

**System Configuration** A significant characteristic of ARINC 653 compliant separation kernels is that partitions, policies and communication components are statically configured at built-time. In our specification, we use **record** `Sys_Config` to define the system configuration and **fixes** `sysconf :: "Sys_Config"` as a constant in the specification.

### 3.2 State-based Kernel Execution

**Event and State** We consider four types of events: *hypercalls*, *system events*, *exceptions*, and *actions in partitions*. Hypercalls cover all IPC services in ARINC 653. System events are the actions of the kernel itself and include kernel initialization, scheduling and message transmission. The other two types are abstract events that can be refined in a concrete specification. Events are illustrated in Fig. 1 as dotted line arrows and italics. Since there is no in-kernel concurrency, all these events execute atomically.

It is not that all events are enabled in a state. We use a function `event_enabled` to indicate whether an event can execute in a state. The function `exec_event` executes an event in a state and changes the state when it is enabled. In the event specification, we define functions to implement concrete communication, scheduling and message transmission. The `exec_event` function here invokes the concrete functions.

The state is defined as **record** `State`, which consists of information about the current running partition, partition states, communication states, created ports and current value of local variables in domains. For a state `s::State` and a sequence of events `as`, `execute as s` denotes the final state reached by executing `as` from `s`.

**Domain of Events** Events have their own execution domains. The domain of the system events is static: the domain of the event *scheduling* is the scheduler; the domain of *message transmission* is the transmitter. On the other hand, the domain of hypercalls is dynamic and dependent on the current state of the kernel, defined as `domain_of_event s (hyperc h) = current s`, where `current s` returns the currently running partition in the state `s`.

**State Reachability** Since not all events are enabled in a state, some states in the type `State` are not reachable from the initial state `s0`. Let `reachable s` $\equiv$ $\exists$`as. s = execute as s0` denote that the state `s` is reachable from the initial state `s0`. According to the definition of `reachable` and `execute`, we have `reachable s0` and Lemma 1.

**Lemma 1.** $\forall$`s as. reachable s` $\wedge$ `s' = execute as s` $\longrightarrow$ `reachable s'`

**State Equivalence** A key concept for information flow security is that states are *identical* for a security domain. We define an equivalence relation $\sim$ `d` $\sim$ on states for each domain `d` such that `s` $\sim$ `d` $\sim$ `t` if and only if states `s` and `t` are identical for domain `d`, that is to say states `s` and `t` are indistinguishable for domain `d`. For a set of domains `D`, we define `s` $\approx$ `D` $\approx$ `t` $\equiv$ $\forall$`d` $\in$ `D. s` $\sim$ `d` $\sim$ `t`.

For a partition `d`, `s` $\sim$ `d` $\sim$ `t` if and only if `vpeq_part s d t`, where

```
vpeq_part s d t ≡ vpeq_vars s (the ((domv sysconf) d)) t
      ∧ (partitions s) d = (partitions t) d ∧ vpeq_part_comm s d t
```

It means that states `s` and `t` are equivalent for a partition `d`, when values of local variables, partition state, and communication abilities of `d` on these two states are the same. An example of the communication ability is that if a destination queuing port `p` is not empty in two states `s` and `t`, a partition `d` has the same ability on `p` in `s` as in `t`, because `d` has the ability to receive a message from `p` in these two states. The equivalence of communication abilities defines that partition `d` has the same set of ports, and that the number of messages is the same for all destination ports on states `s` and `t`.

Two states `s` and `t` are equivalent for the `scheduler` when the values of local variables of the scheduler and the current running partition on the two states are the same. The equivalence of states for the `transmitter` requires that all ports, states of the ports and values of local variable are the same.

### 3.3 Event Specification

The event specification defines the concrete functions to implement the execution of events. The functionalities of separation kernels in this paper include kernel initialization, scheduling, message transmission and hypercalls. The kernel initialization considers initialization of the kernel state. Since our specification does not define processes, we only consider the partition scheduling rather than the two-level scheduling on partition and process levels in ARINC 653. Because the execution of message transmission is also under the control of scheduling, we define an abstract partition scheduling that non-deterministically chooses one partition or the transmitter as the currently executing domain.

This subsection mainly discusses channel-based communication services in ARINC 653 and the message transmission. All events and their descriptions in the event specification are shown in Table 1.

**Channel-based Communication Services** ARINC 653 specifies the behavior of ports and the communication services via ports in detail. Programs in a partition could use IPC by invoking these services. ARINC 653 defines eleven services for sampling and queuing ports (No. 1 $\sim$ 11 in Table 1). The communication architecture is illustrated in Fig. 3.

**Table 1.** Events in Our Specification

| No. | Name | Description of Event Specification |
|---|---|---|
| **Hypercalls** | | |
| (1) | Create_Sampling_Port | Create a sampling port. An identifier is assigned by the kernel and returned. |
| (2) | Write_Sampling_Message | Write a message in the specified sampling port. The message overwrites the previous one. |
| (3) | Read_Sampling_Message | Read a message from the specified sampling port. |
| (4) | Get_Sampling_Portid | Return the sampling port identifier that corresponds to a sampling port name. |
| (5) | Get_Sampling_Portstatus | Return the current status of the specified sampling port. |
| (6) | Create_Queuing_Port | Create a queuing port. An identifier is assigned by the kernel and returned. |
| (7) | Send_Queuing_Message | Send a message in the specified queuing port. If there is sufficient space in the queuing port to accept the message, the message is inserted into the port buffer. If there is insufficient space, the message is lost. |
| (8) | Receive_Queuing_Message | Receive a message from the specified queuing port. If the queuing port is not empty, a message in the port buffer is removed and returned. If the queuing port is empty, `None` is returned. |
| (9) | Get_Queuing_Portid | Return the queuing port identifier that corresponds to a queuing port name. |
| (10) | Get_Queuing_Portstatus | Return the current status of the specified queuing port. |
| (11) | Clear_Queuing_Port | Discard any messages in the message buffer of the specified destination port. |
| **System events** | | |
| (12) | Schedule | Set one partition or the transmitter as the currently running domain. |
| (13) | Transfer_Sampling_Message | Copy the message in the source sampling port to all destination sampling ports of a sampling channel, if all ports of this channel have been created. |
| (14) | Transfer_Queuing_Message | Copy a message in the source queuing port to the destination queuing port of a queuing channel and remove the message from the source port, if the two ports of this channel have been created and the source port is not empty. If the destination port is full, the message is lost. |
| (15) | Init | Initialize the kernel state using the system configuration. |

In the first stage of this work, we design the event specification completely based on the service behavior specified in ARINC 653. When proving the unwinding conditions on these events, we find covert channels (Section 5 in detail) and change the service specification defined in ARINC 653 to avoid these covert channels. McCullough [17] provides three ways to avoid covert channels: unbounded buffer, process blocking and message loss. According to the discussion in Section 2, we do not allow partition blocking in communication services. Because unbounded buffer would lead to a bigger problem of denial of service (DoS), the feasible way for our specification is to allow message loss. In order to avoid covert channels, we allow message loss when sending a message to a queuing port and transmitting a message in a queuing channel.

We use a set of functions to implement one service. For instance, the `Send_Queuing_Message` service is implemented by function `send_queuing_message_maylost` as follows and a set of related functions invoked by this function.

**definition** `send_queuing_message_maylost :: "Sys_Config ⇒ State ⇒ port_id ⇒ Message ⇒ (State × bool)"` **where**

```
"send_queuing_message_maylost sc s p m ≡
   (if(¬ is_a_queuingport s p ∨ ¬ is_source_port s p
             ∨ ¬ is_a_port_of_partition s p )  then  (s, False)
    else if is_full_portqueuing sc s p then  (s, True)
    else  (insert_msg2queuing_port s p m, True))"
```

As specified in the `Send_Queuing_Message` service in ARINC 653, when sending a message via a queuing port, it fails if either the specified port does not exist, or it is not a source port, or it is not in current partition. When the port is full, the calling process is blocked. Since blocking is not considered in this paper, we just discard the message.

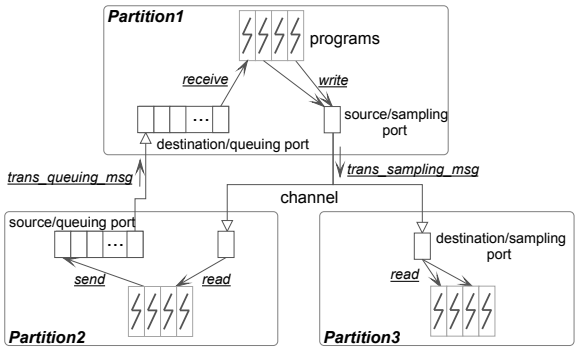


**Fig. 3.** Channel-based Communication in ARINC 653

**Message Transmission on Channels** ARINC 653 does not define the functionalities of message transmission and leaves its implementation to underlying separation kernels. We design a basic specification of the message transmission in this paper.

The message transmission on channels is shown in Fig. 3. ARINC 653 has two modes of channel-based communication: sampling and queuing mode. The multicast message that is sent from a single source to more than one destination is supported in sampling mode. The queuing mode only supports the unicast message. In sampling mode, a message transmission on a channel copies the message in the source sampling port of the channel to the buffers of all destination sampling ports of the channel. Whilst in queuing mode, a message transmission on a channel copies a message in the buffer of the source queuing port, removes it from this buffer and stores the message into the buffer of the destination queuing port of the channel. When the buffer of the destination queuing port is full, the message is discarded.

For instance, the message transmission in queuing mode is defined as follows. If the source and destination port have been created and there are messages in the buffer of the source port, a message in the buffer is removed and inserted into the buffer of the destination port. When the buffer of the destination port is full, the message is discarded.

```
primrec transf_queuing_msg_maylost :: "Sys_Config ⇒ State ⇒ Channel_Type
⇒ State" where
  "transf_queuing_msg_maylost sc s (Channel_Queuing _ sn dn) =
    (let sp = get_portid_by_name s sn; dp = get_portid_by_name s dn in
      if sp ≠ None ∧ dp ≠ None ∧ has_msg_inportqueuing s (the sp) then
        let sm = remove_msg_from_queuingport s (the sp) in
            if is_full_portqueuing sc (fst sm) (the dp) then s
            else
              insert_msg2queuing_port (fst sm) (the dp) (the (snd sm))
      else s )" |
  "transf_queuing_msg_maylost sc s (Channel_Sampling _ _ _) = s"
```

### 3.4 Correctness of Formal Specification

To assure the correctness of our specification, beside the manual validation by inspecting the Isabelle/HOL specification, we prove that functionalities of the specified ser-

vices are correct w.r.t. the ARINC 653 informal description [5] by means of 33 lemmas for events and invariants. Due to the atomicity of event execution, the correctness of an event can be specified and proved by pre- and post-conditions of the event in Hoare logic [14], i.e., $\{P\}\ C\ \{Q\}$, where $C$ is the Isabelle function implementing the event, $P$ and $Q$ are the pre- and post-conditions respectively. Since the execution of events always terminate, our specification is a total correctness specification. Termination is ensured by using the **primrec** and **definition** in Isabelle/HOL to define the functions in our specification and proved automatically in Isabelle/HOL. For instance, the correctness lemma for the event `Create_Sampling_Port` is as follows. The pre-condition is that the port named `p` is configured, has not been created and is a port of the currently running partition. Under the pre-condition, the execution of `create_sampling _port` returns a pair of the new state and the assigned identifier of the created port. The post-condition ensures that the identifier (`the (snd r)`) is stored in the `ports` in the new state (`ports (comm (fst r))`).

**Lemma 2 (Correctness of Create_Sampling_Port).**

```
{ get_samplingport_conf sysconf p ≠ None ∧ get_portid_by_name s p = None ∧
p ∈ get_partition_cfg_ports_byid sysconf (current s) }
r = create_sampling_port sysconf s p
{ (ports (comm (fst r))) (the (snd r)) ≠ None }
```

Functional correctness requires to prove invariants on the data structures defining the state. An invariant is a safety property and defined on states as a predicate $\psi$ `s`. It is preserved in all reachable states by proving the invariant theorem: `reachable s` $\Longrightarrow$ $\psi$ `s`. A typical invariant is the predicate `port_consistent s`. We use a set to store created ports. The port state (e.g., the messages currently in the port) is defined as `Ports = "port_id ⇀ Port_Type"`. Ports belong to different partitions that is defined as `part_ports :: "port_id ⇀ partition_id"`. The `port_consistent s` requires that the created port set and the domains of these two partial functions are the same in any reachable states. The invariant theorem is proved by Lemma 1 and other two lemmas: (1) $\psi$ `s0` and (2) $\forall$`s as.` $\psi$ `s` $\wedge$ `s' = execute as s` $\longrightarrow$ $\psi$ `s'`.

## 4 Information Flow Security and Proofs

This section first presents a set of information flow security properties defined on the execution model, which includes the original definitions of noninterference [27], nonleakage [23] and noninfluence [23], and their variants. Nonleakage is language-based information flow security and noninfluence is the combination of noninterference and nonleakage. Then, we present an overview of our proof structure and the proofs which include an inference framework of these properties and the security proofs of our event specification.

### 4.1 Formalizing Noninterference

Since intransitive policies could be used to specify channel control policies [27], we consider intransitive noninterference in this paper. The essence of noninterference on separation kernels is that a partition `d` cannot distinguish the final states between executing a sequence of events `as` and executing its purged sequence from the initial state.

In the purged sequence, the events of partitions that are not allowed to pass information to d directly or indirectly are removed.

In order to express the allowed information flow for intransitive policies, we employ the function sources [27], which takes a sequence of events as and a target domain d and yields the set of domains that are allowed to pass information to d when as occurs. Due to the dependency of event domains on states, the sources function in our specification depends on the current state s. The sources function is used to define the classical purge function, ipurge, in terms of which security properties are formulated. The ipurge as s d yields the sequence of events as, where all events that are not allowed to pass information to d directly or indirectly when as is executed from s are removed.

We use the abbreviation s ◁ as ≅ t ◁ bs @ d for the observational equivalence. It denotes that d is identical in the two final states after executing as from s (by execute as s) and executing bs from t. Traditionally, this equivalence is defined using a projection function output which returns the observed results on a state by a domain. In this paper, we have combined the output in the state equivalence presented in Subsection 3.2. This allows us to avoid the unwinding condition of *output consistent*. We define the classical nontransitive noninterference [27] on our execution model as follows.

```
noninterference ≡ ∀d as. (s0 ◁ as ≅ s0 ◁ (ipurge as s0 d) @ d)
```

In the definition of noninterference, the ipurge function only deletes all unsuitable events. A strong version of noninterference is introduced in [23] to handle arbitrary insertion and deletion of secret events. Oheimb [23] says that the strong noninterference and the original one are equivalent in deterministic cases. We define this strong version of noninterference on the execution model as weak_noninterference, since noninterference implies weak_noninterference on our execution model.

The above definitions of noninterference are based on the initial state s0, but separation kernels usually support *warm* or *cold start* and they may start to execute from a non-initial state. Therefore, we define a more general version of noninterference as follows based on the reachable function. This general noninterference requires that the system starting from any reachable state is secure. It is obvious that this noninterference implies the classical noninterference due to the lemma: reachable s0.

```
noninterference_r ≡ ∀d as s. reachable s ⟶
                    (s ◁ as ≅ s ◁ (ipurge as s d) @ d)
```

### 4.2 Formalizing Nonleakage and Noninfluence

Language-based information flow security is generalized to arbitrary multi-domain policies in [23] as a new notion *nonleakage*. *Nonleakage* and *noninterference* are also combined in [23] as a new notion *noninfluence*. Murray et al. [20] have extended the original definition of nonleakage and noninfluence and defined the general forms of them for operating systems based on the scheduler. We use Murray's definitions and define them on our execution model as follows.

```
nonleakage ≡ ∀d as s t. reachable s ∧ reachable t ⟶
        (s ∼ (scheduler sysconf) ∼ t) ⟶ (s ≈ (sources as s d) ≈ t)
        ⟶ (s ◁ as ≅ t ◁ as @ d)
```

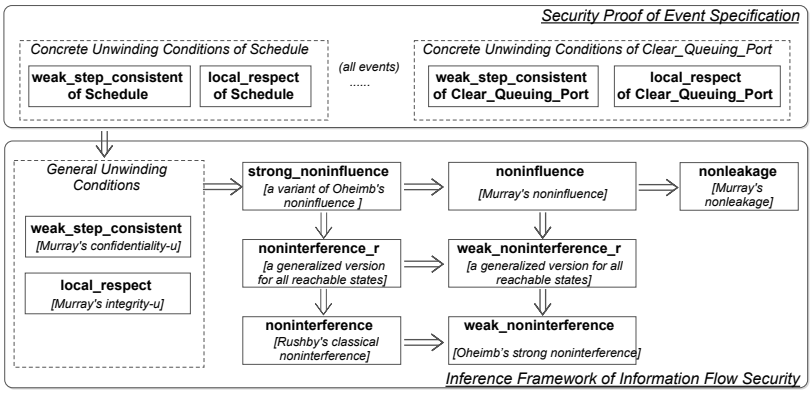


**Fig. 4.** Proof Structure

```
noninfluence ≡ ∀ d as bs s t . reachable s ∧ reachable t ⟶
   (s ≈ (sources as s d) ≈ t) ⟶ (s ∼ (scheduler sysconf) ∼ t) ⟶
   ipurge as s d = ipurge bs s d ⟶ (s ◁ as ≅ t ◁ bs @ d)
```

The intuitive meaning of nonleakage is that if the secret data is not leaked initially, the secret data should not be leaked during executing a sequence of events. Separation kernels are said to preserve *nonleakage* when for any pair of reachable states `s` and `t` and observing domain `d`, if (1) `s` and `t` are equivalent for all domains that may (directly or indirectly) interfere with `d` during the run of `as`, i.e., `s ≈ (sources as s d) ≈ t`, and (2) the same domain is currently running in both states, i.e., `s ∼ (scheduler sysconf) ∼ t`, then `s` and `t` are observationally equivalent for `d` when executing `as`. Murray's definition of noninfluence is a weak one, we propose a strong one according to the Oheimb's noninfluence by extending the scheduler and state reachability as follows.

```
strong_noninfluence ≡ ∀ d as s t . reachable s ∧ reachable t ⟶
        (s ≈ (sources as s d) ≈ t) ⟶ (s ∼ (scheduler sysconf) ∼ t)
        ⟶ (s ◁ as ≅ t ◁ (ipurge as t d) @ d)
```

### 4.3 Proof Structure

As discussed in Section 2, proofs of information flow security on our specification comprise two parts: an inference framework of information flow security properties on the execution model and security proofs of the event specification. The proof structure of this work is shown in Fig. 4, where an arrow means the implication between properties. In the next two subsections, we discuss the two parts of proofs in turn.

### 4.4 Inference Framework of Information Flow Security

In order to clarify different properties of information flow security on our specification, we provide an inference framework on the execution model as shown in the lower part of Fig. 4. We have proven all implication relations between these properties on the execution model. We could see that the property `strong_noninfluence` is the strongest one and if this property is satisfied, so are all other properties.

The standard proof of information flow security properties is discharged by proving a set of unwinding conditions [27] that examine individual execution steps of the system. Our work follows this approach. In order to prove strong_noninfluence, we define two general unwinding conditions, weak_step_consistent and local_respect, as follows. As there is no output function in our specification, we do not define the classical unwinding condition of *output consistent*.

```
weak_step_consistent ≡ ∀ d a s t . reachable s ∧ reachable t ⟶
      (s ∼ d ∼ t) ∧ (s ∼ (scheduler sysconf) ∼ t) ∧
      ((domain_of_event s a) ⇝ d) ∧ (s ∼ (domain_of_event s a) ∼ t)
      ⟶ ((exec_event s a) ∼ d ∼ (exec_event t a))
```

```
local_respect ≡ ∀ a d s s'. reachable s ⟶
((domain_of_event s a) \⇝ d) ∧ (s' = exec_event s a) ⟶ (s ∼ d ∼ s')
```

The weak_step_consistent means that for any pair of reachable states s and t, and any observing domain d, the next states after executing any event a on s and t are indistinguishable for d, i.e., (exec_event s a) ∼ d ∼ (exec_event t a), if s and t are indistinguishable for d, the same domain is currently running in s and t, the domain of event a in state s can interference with d, and s and t are indistinguishable for the domain of event a. The weak_step_consistent is the same as confidentiality-u proposed in [20]. The local_respect is the same as integrity-u in [20], which means that an event a that executes in some state s can affect only those domains to which the domain executing event a is allowed to send information.

### 4.5 Security Proofs of Event Specification

The second step of proofs is to show security of the event specification. From definitions of the two general unwinding conditions, we could see that in order to prove the satisfaction of the two conditions on our specification, we can induct on each type of events in separation kernels and prove that each concrete event satisfies the two conditions. Therefore, we define a set of *concrete unwinding conditions* for all events. Satisfaction of the concrete unwinding conditions of one event implies that the event satisfies the general unwinding conditions. For instance, Lemma 3 and 4 show the concrete unwinding conditions for event Create_Queuing_Port.

**Lemma 3 (Local_respect of creating_queuing_port).**

```
reachable s ∧ is_a_partition sysconf (current s) ∧ (current s) \⇝ d ∧
s' = fst (create_queuing_port sysconf s pname) ⟹ s ∼ d ∼ s'
```

**Lemma 4 (Weak_step_consistent of creating_queuing_port).**

```
is_a_partition sysconf (current s) ∧ reachable s ∧ reachable t ∧
s ∼ d ∼ t ∧ s ∼ (scheduler sysconf) ∼ t ∧ (current s) ⇝ d ∧
s ∼ (current s) ∼ t ∧ s' = fst (create_queuing_port sysconf s pname) ∧
t' = fst (create_queuing_port sysconf t pname) ⟹ s' ∼ d ∼ t'
```

Finally, we conclude the satisfaction of strong_noninfluence on our specification and all other information flow security properties according to the inference framework.

**Table 2.** Specification and Proofs Statistics

| **Specification** | | | | **Proofs** | | | |
|---|---|---|---|---|---|---|---|
| Item | # of function /definition | LOC | PM | Item | # of lemma /theorem | LOP | PM |
| Execution model | 32 | $\sim$ 200 | 2 | Inference Framework | 61 | $\sim$ 1000 | 6 |
| Event Specification | 68 | $\sim$ 800 | | Correctness | 33 | $\sim$ 6000 | |
| | | | | Security | 123 | | |
| **Total** | 100 | $\sim$ 1000 | 2 | **Total** | 217 | $\sim$ 7000 | 6 |

## 5 Results and Discussion

**Evaluation** We use Isabelle/HOL as the specification and verification system for separation kernels. The proofs of information flow security in our specification are conducted in the structured proof language *Isar* in Isabelle, allowing for proof text naturally understandable for both humans and computers. All derivations of our proofs have passed through the Isabelle proof kernel.

The statistics for the effort and size of the specification and proofs are shown in Table 2. We use 100 functions/definitions and $\sim$ 1000 lines of code (LOC) of Isabelle/HOL to specify the execution model and event specification. 217 lemmas/theorems in Isabelle/HOL are proved using $\sim$ 7000 lines of proof (LOP) of Isar to ensure the information flow security of our specification. The work is carried out by a total effort of roughly 8 person-months (PM).

**Validating and Fixing Covert Channels in ARINC 653** When proving the satisfaction of unwinding conditions on the events, we find some security flaws, i.e., covert channels to leak information, in ARINC 653.

*Covert Channel 1: queuing mode channel-based communication*. If there is a queuing mode channel from partition `a` to `b` and no other channels exist, then it is secure that `a` $\rightsquigarrow$ `transmitter`, `transmitter` $\rightsquigarrow$ `b`, `transmitter` $\not\rightsquigarrow$ `a` and `b` $\not\rightsquigarrow$ `transmitter`. In fact, these security policies are violated in ARINC 653. Firstly, when `a` sends a message by invoking `Send_Queuing_Message` service of ARINC 653, the service returns `NOT_AVAILABLE` or `TIMED_OUT` when the buffer is full, and returns `NO_ERROR` when the buffer is not full. However, the full/empty status of the buffer in the port can be changed by message transmission executed by the transmitter. Thus, the `local_respect` property is not preserved on `Send_Queuing_Message` service, and `transmitter` $\not\rightsquigarrow$ `a` is violated. Secondly, due to no message loss required by ARINC 653, the transmitter cannot transmit a message on a channel when the destination queuing port is full. However, the full status of the destination port can be changed by `Receive_Queuing_Message` service executed by partition `b`. Thus, the `local_respect` property is not preserved on the event of message transmission, and `b` $\not\rightsquigarrow$ `transmitter` is violated. To avoid this covert channel, we allow message loss when sending messages to a queuing port or transmitting message on a queuing mode channel.

*Covert Channel 2: Create_Sampling_Port and Create_Queuing_Port services*. This is a potential covert channel. It is dependent on the concrete implementation of ARINC 653 and can be avoided by careful designs. In ARINC 653, the service `Create_Sampling_Port` and `Create_Queuing_Port` create a port and return a new unique identifier assigned by the kernel to the new port. In the initial specification, we use a

natural number to maintain this new identifier. This number is initially assigned to one and increased by one after each port creation. We find in this design that the number becomes a covert channel that can flow information from any partition to another, and the two events do not preserve the `weak_step_consistent` property. This covert channel can be avoided by assigning the port identifier to each port during system initialization or in the system configuration.

**Validating and Fixing Covert Channels in Open-source Implementations** We have manually validated the found covert channels in two open-source separation kernels, i.e., XtratuM and POK. Covert channels are found when we validate these two implementations.

The version of XtratuM we validate is v3.7.3 for SPARC v8 architecture. Unlike that there is one buffer for each queuing port in ARINC 653, XtratuM uses one shared buffer between the source port and the destination port of a queuing mode channel as a transmitter. If the buffer is not full, the hypercall *SendQueuingPort* inserts the message into the buffer and notifies the receiver; whilst if the buffer is full, *SendQueuingPort* immediately returns *XM_OP_NOT_ALLOWED*. The hypercall *ReceiveQueuingPort* has a similar design. Thus, the found covert channel 1 exists in XtratuM. The way to avoid this security flaw is to redesign the hypercall *SendQueuingPort* to lose the message and return *XM_OK* when the buffer is full.

The version of POK we validate is the latest one released in 2014. Different from XtratuM, POK has a transmitter to transfer messages from a source port to a destination port of a channel. POK blocks processes to wait for resources. If the buffer is not full, the syscall *pok_port_queueing_send* inserts the message into the buffer; whilst if the buffer is full and $timeout = 0$, it immediately returns *POK_ERRNO_FULL*. *pok_port_transfer* responds for transmitting messages from a source port to a destination one and returns *POK_ERRNO_SIZE* when the destination port has no available space to store messages. Thus, the found covert channel 1 exists in POK. The way to avoid this security flaw is to allow message loss or block the calling process until the port buffer is not full in the syscall *pok_port_queueing_send*.

When creating a port, XtratuM and POK use the index of the port in the port array as the new identifier. Thus, they do not have the covert channel 2.

**Discussion** The reusability of formal specification and proofs can largely alleviate the enormous efforts needed when others enforce information flow security on separation kernels. Our formal specification can be refined to the concrete specification of separation kernels. In the concrete specification, new variables and events may be introduced and some events in this paper may be refined. The state equivalence in our specification is sufficient for the abstract and concrete specification of the channel-based communication. Therefore, the new variables in the concrete specification do not change the definition of state equivalence, and thus the new variables and new events manipulating these variables do not break the information flow security of the concrete specification. Information flow security properties in this paper can be preserved on refinement of events of the channel-based communication according to the conclusion in [20]. Due to the reusability of the formal specification, the inference framework and the security proofs in this work are also reusable for the concrete specification.

## 6 Related Work and Conclusions

**Information Flow Security** Information flow security [28] has attracted many research efforts in recent years. State-event based noninterference [27] is usually chosen for verifying general purpose operating systems and separation kernels [20]. Language-based information flow security was generalized to arbitrary multi-domain policies in [23] as a new state-event based notion nonleakage. Oheimb [23] also combined the classical noninterference and nonleakage as the notion noninfluence. These properties have been instantiated for operating systems in [20] and formally verified on seL4 [19]. In our work, all of these properties and their variants are defined in our specification. We also propose an inference framework to clarify the implications between these properties.

**Formal Specification and Verification of Separation Kernels** Formal methods have been widely applied on separation kernels in recent years [8, 30–32, 13, 25, 33, 12, 19, 9, 29]. An overview is available in [34]. An Isabelle/HOL specification for a generic separation kernel was published by EURO-MILS project [31]. They provided an abstraction specification for Controlled Interruptible Separation Kernels (CISK), instantiated it to a separation kernel model, and then applied them on the PikeOS separation kernel [32]. The Isabelle/HOL specification of seL4 was extended to a separation kernel specification in [19]. Formal specification in our work provides a detailed model for ARINC 653 channel-based communication, which is not covered in related work. In particular, there is no concrete communication actions in specification of [31]. The IPC syscalls in seL4 [19] and PikeOS [32] are very different from ARINC 653 channel-based communication.

**Formalization and Verification of ARINC 653** Formalization and verification of ARINC 653 have been considered in recent years, such as formal specification of ARINC 653 architecture [24], modeling ARINC 653 for model driven development of IMA applications [11], and verification of application software on top of ARINC 653 [7]. In [35], the system functionalities and all service requirements in ARINC 653 have been formalized in Event-B, and some inconsistencies have been found in the standard. These works aim at safety of separation kernels or applications. Our work is the first to conduct a formal security analysis of the ARINC 653 standard.

**Conclusions and Future Work** In this paper, we applied Isabelle/HOL to formally specify and verify separation kernels with ARINC 653 channel-based communication. We provided a formal specification with mechanically checked proofs that is totally free of covert channels and therefore provided information flow security for high assurance systems. We revealed covert channels in ARINC 653 and validated their existence in XtratuM and POK. Our specification is reusable for subsequent specification refinement and development of implementations. The proofs in this work can alleviate the verification efforts on information flow security. In the next step, we will develop a formal specification of separation kernels supporting multi-core and the specification in this paper will be revised. Due to the kernel concurrency between cores, we will find a feasible way to verify multi-core separation kernels. The long-term goal of our project is to construct a compositional approach of building security verified system, which includes verification of the functional and noninterference correctness for a separation/partitioning microkernel for a multi-core architecture, and verification of the functional correctness of the underlying hardware.

***Acknowledgement*** We would like to thank Gerwin Klein and Ralf Huuck of NICTA, Australia for their suggestions. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

## References

1. Lynxsecure separation kernel hypervisor. http://www.lynx.com/products/hypervisors/, accessed: 2015-07
2. Sysgo pikeos hypervisor. https://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/, accessed: 2015-07
3. Wind river vxworks 653 platform. http://www.windriver.com/products/vxworks/certification-profiles/, accessed: 2015-07
4. Wind river vxworks mils platform. http://www.windriver.com/products/vxworks/certification-profiles/, accessed: 2015-07
5. Aeronautical Radio, Inc.: ARINC Specification 653: Avionics Application Software Standard Interface, Part 1 - Required Services (November 2010)
6. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security 18(6), 1157–1210 (2010)
7. de la Cmara, P., Castro, J.R., Gallardo, M.d.M., Merino, P.: Verification support for arinc-653-based avionics software. Software Testing, Verification and Reliability 21(4), 267–298 (2011)
8. Craig, I.D.: Formal Refinement for Operating System Kernels, chap. 5. Springer (2007)
9. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple arm-based separation kernel. In: Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13). pp. 223–234. ACM, New York, NY, USA (2013)
10. Delange, J., Lec, L.: Pok, an arinc653-compliant operating system released under the bsd license. In: Proceedings of the 13th Real-Time Linux Workshop (2011)
11. Delange, J., Pautet, L., Kordon, F.: Modeling and validation of arinc653 architectures. In: Proceedings of Embedded Real-time Software and Systems Conference (ERTS'10) (2010)
12. Freitas, L., McDermott, J.: Formal methods for security in the xenon hypervisor. International journal on software tools for technology transfer 13(5), 463–489 (2011)
13. Heitmeyer, C.L., Archer, M.M., Leonard, E.I., McLean, J.D.: Applying formal methods to a certifiably secure software system. IEEE Transactions on Software Engineering 34(1), 82–98 (2008)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
15. Levin, T.E., Irvine, C.E., Weissman, C., Nguyen, T.D.: Analysis of three multilevel security architectures. In: Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSA'07). pp. 37–46. ACM (2007)
16. Masmano, M., Ripoll, I., Crespo, A., Metge, J.: Xtratum: a hypervisor for safety critical embedded systems. In: Proceedings of the 11th Real-Time Linux Workshop. pp. 263–272 (2009)
17. McCullough, D.: Noninterference and the composability of security properties. In: Proceedings of IEEE Symposium on Security and Privacy (S&P'88). pp. 177–186. IEEE Computer Society (1988)
18. Millen, J.: 20 years of covert channel modeling and analysis. In: Proceedings of IEEE Symposium on Security and Privacy (S&P'99). pp. 113–114. IEEE (1999)

19. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: sel4: from general purpose to a proof of information flow enforcement. In: Proceedings of IEEE Symposium on Security and Privacy (S&P'13) (2013)
20. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Klein, G.: Noninterference for operating system kernels. In: Certified Programs and Proofs, pp. 126–142. Springer (2012)
21. National Security Agency: U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness (2007)
22. Nipkow, T., Wenzel, M., Paulson, L.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag, Berlin, Heidelberg (2002)
23. von Oheimb, D.: Information flow control revisited: Noninfluence= noninterference+ nonleakage. In: Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS'04). pp. 225–243. Springer (2004)
24. Oliveira Gomes, A.: Formal Specification of the ARINC 653 Architecture Using Circus. Master's thesis, University of York (2012)
25. Richards, R.J.: Modeling and security analysis of a commercial real-time operating system kernel. In: Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 301–322. Springer (2010)
26. Rushby, J.: Design and verification of secure systems. ACM SIGOPS Operating Systems Review 15(5), 12–21 (1981)
27. Rushby, J.: Noninterference, transitivity, and channel-control security policies. Tech. rep., SRI International, Computer Science Laboratory (1992)
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
29. Sanán, D., Butterfield, A., Hinchey, M.: Separation kernel verification: The xtratum case study. In: Verified Software: Theories, Tools and Experiments, pp. 133–149. Springer (2014)
30. Velykis, A., Freitas, L.: Formal modelling of separation kernel components. In: Proceedings of the 7th International Colloquium Theoretical Aspects of Computing (ICTAC'10). pp. 230–244. Springer (2010)
31. Verbeek, F.F., Tverdyshev, S.S., etc.: Formal specification of a generic separation kernel. Archive of Formal Proofs (2014)
32. Verbeek, F., Havle, O., Schmaltz, J., Tverdyshev, S., Blasum, H., Langenstein, B., Stephan, W., Wolff, B., Nemouchi, Y.: Formal api specification of the pikeos separation kernel. In: NASA Formal Methods, pp. 375–389. Springer (2015)
33. Wilding, M.M., Greve, D.A., Richards, R.J., Hardin, D.S.: Formal verification of partition management for the aamp7g microprocessor. In: Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 175–191. Springer (2010)
34. Zhao, Y.: Formal specification and verification of separation kernels: An overview. ArXiv e-prints (2015), http://arxiv.org/abs/1508.07066
35. Zhao, Y., Yang, Z., Sanan, D., Liu, Y.: Event-based formalization of safety-critical operating system standards: An experience report on arinc 653 using event-b. In: Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE'15). pp. 281–292. IEEE Computer Society (2015)