Refinement-based Specification and Security Analysis of Separation Kernels

Yongwang Zhao, David Sanán, Fuyuan Zhang, Yang Liu

Abstract—Assurance of information-flow security by formal methods is mandated in security certification of separation kernels. As an industrial standard for improving safety, ARINC 653 has been complied with by mainstream separation kernels. Due to the new trend of integrating safe and secure functionalities into one separation kernel, security analysis of ARINC 653 as well as a formal specification with security proofs are thus significant for the development and certification of ARINC 653 compliant Separation Kernels (ARINC SKs). This paper presents a specification development and security analysis method for ARINC SKs based on refinement. We propose a generic security model and a stepwise refinement framework. Two levels of functional specification are developed by the refinement. A major part of separation kernel requirements in ARINC 653 are modeled, such as kernel initialization, two-level scheduling, partition and process management, and inter-partition communication. The formal specification and its security proofs are carried out in the Isabelle/HOL theorem prover. We have reviewed the source code of one industrial and two open-source ARINC SK implementations, i.e. VxWorks 653, XtratuM, and POK, in accordance with the formal specification. During the verification and code review, six security flaws, which can cause information leakage, are found in the ARINC 653 standard and the implementations.

Index Terms—Separation Kernels, ARINC 653, Refinement, Formal Specification, Information-flow Security, Common Criteria, Theorem Proving.

1 Introduction

In recent years, a trend in embedded systems is to enable multiple applications from different vendors and with different criticality levels to share a common set of physical resources, such as IMA [1] and AUTOSAR [2]. To facilitate such a model, resources of each application must be protected from other applications in the system. The separation kernel [3] and its variants (e.g. partitioning kernels and partitioning operating systems [4], [5]) establish such an execution environment by providing to their hosted applications spatial/temporal separation and controlled information flow. Separation kernels, such as PikeOS, VxWorks 653, INTEGRITY-178B, LynxSecure, and LynxOS-178, have been widely applied in domains from aerospace and automotive to medical and consumer electronics.

Traditionally, safety and security of critical systems are assured by using two kinds of separation kernels respectively, such as VxWorks 653 for safety-critical systems and VxWorks MILS for security-critical systems. In order to improve the safety of separation kernels, the ARINC 653 standard [5] has been developed to standardize the system functionality as well as the interface between the kernel and applications. ARINC 653 is the premier safety standard and has been complied with by mainstream separation kernels. On the other hand, the security of separation kernels is usually achieved by the Common Criteria (CC) [6] and Separation Kernel Protection Profile (SKPP) [7] evaluation, in which formal verification of information-flow security is mandated for high assurance levels.

A trend in this field is to integrate the safe and secure functionalities into one separation kernel. For instance, PikeOS, LynxSecure, and XtratuM are designed to support both safety-critical and security-critical solutions. Therefore, it is necessary to assure the security of the functionalities defined in ARINC 653 when developing ARINC 653 compliant Separation Kernels (ARINC SKs). Moreover, a security verified specification compliant with ARINC 653 and its mechanically checked proofs are highly desirable for the development and certification of ARINC SKs. The highest assurance level of CC certification (EAL 7) requires comprehensive security analysis using formal representations of the security model and functional specification of ARINC SKs as well as formal proofs of correspondence between them. Although formal specification [8], [9], [10], [11] and verification [12], [13], [14], [15], [16], [17], [18] of information-flow security on separation kernels have been widely studied in academia and industry, information-flow security of ARINC SKs has not been studied to date. To the best of our knowledge, our work is the first effort on this topic in the literature.

There exist three major challenges to be addressed in this work. First, the ARINC 653 standard is highly complicated. It specifies the system functionality and 57 standard services of separation kernels using more than 100 pages of informal descriptions. Second, as a sort of hyperproperties [19], it is difficult to automatically verify information-flow security on separation kernels so far. Formal analysis of information-flow security of separation kernels is difficult and needs

- *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.*
- *Y. Zhao is with the School of Computer Science and Engineering, Beihang Univerisity, Beijing, China (Email: zhaoyw@buaa.edu.cn) and the School of Computer Science and Engineering, Nanyang Technological University, Singapore (Email: ywzhao@ntu.edu.sg).*
- *D. Sanán, F. Zhang and Y. Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore (Email: {sanan, fuzh, yangliu}@ntu.edu.sg).*

exhausting manual efforts. Moreover, there exist different definitions of information-flow security (e.g. in [20], [21], [22], [23]) and the relationship of them on ARINC SKs has to be clarified. Third, reusability of formal specification and proofs is important. They should be extensible and easy to be reused for subsequent development and certification.

This paper presents a specification development and security analysis method based on stepwise refinement [24], [25] for ARINC SKs with regard to the EAL 7 of CC certification. A separation kernel is an event-driven system reacting to hypercalls or exceptions, which is alternatively called a reactive system. Therefore, we borrow design elements from existing formalisms such as superposition refinement [26] of reactive systems and Event-B [27]. We start from a generic security model of ARINC SKs that could be instantiated as functional specifications at different abstract levels. The refinement in this paper concretizes an abstract functional specification by incorporating additional design elements and by transforming existing ones. The stepwise refinement addresses the first challenge by modeling complicated requirements of ARINC 653 in a modular and hierarchical manner. The refinement provides formal proofs of the correspondence between functional specifications for CC certification. Information-flow security is proven on an abstract specification and it is preserved in a concrete specification by means of refinement. Moreover, system functionalities and services of ARINC 653 are modeled as the event specification in the functional specification. In such a design, information-flow security is proven in a compositional way, i.e. the security of ARINC SKs is implied by the satisfaction of local properties (*unwinding conditions*) on events. Thus, the second challenge is resolved. Using the correctness-by-construction method based on the refinement, well-structured specification of ARINC SKs is developed together with their correctness and security proofs, which helps to resolve the third challenge.

The challenges mentioned above are not well addressed in the literature. In formal verification of the seL4 microkernel [28] and its separation extension [16], and the ED separation kernel [12], refinement methods have been applied. First, due to the post-hoc verification objective of these projects, refinement is not a technique to develop the specification in a stepwise manner, but to prove the conformance between formalizations at different levels. Therefore, they have few levels of specification and the refinement is coarse-grained. Second, ARINC 653 is not the emphasis of these works and the formal specification are not compliant with ARINC 653. Information-flow security verification has been enforced on PikeOS [10], [11], INTEGRITY-178B [13], and an ARM-based separation kernel [17]. However, refinement is not considered in these works. Correctness-by-construction methods have been used to create formal specification of separation kernels [8], [9], [15], [29]. However, information-flow security is not the emphasis of them. Formalization and verification of ARINC 653 have been studied in recent years, such as the formal specification of ARINC 653 architecture [30], modeling ARINC 653 for model-driven development of IMA applications [31], and formal verification of application software on top of ARINC 653 [32]. In [29], the system functionalities and all service in ARINC 653 have been formalized in Event-B, and some inconsistencies have been found in the standard. These works aim at the safety of separation kernels and applications. Our work is the first to conduct a formal security analysis of ARINC 653.

We have used Isabelle/HOL [33] to formalize the security model, the refinement method, and functional specifications as well as to prove information-flow security. All specifications and security proofs are available at "http://securify.scse.ntu.edu.sg/skspecv3/". In detail, the technical contributions of this work are as follows.

1) We define a security model, which is a parameterized abstraction for the execution and security configuration of ARINC SKs. A set of information-flow security properties are defined in the model. An inference framework is proposed to sketch out the implications of the properties and an unwinding theorem for compositional reasoning.
2) We propose a refinement framework for stepwise development of ARINC SKs. Functional specifications of ARINC SKs at different levels are instantiations of the security model, and thus all properties of the security model are satisfied on the specifications. We define a security extended superposition refinement for ARINC SKs, which supports introducing additional design elements (e.g. new state variables and new events). We also show the security proofs of the refinement, i.e. the preservation of security properties during the refinement.
3) We develop a top-level specification for ARINC SKs which covers kernel initialization, partition scheduling, partition management, and inter-partition communication (IPC) according to ARINC 653. A second-level specification is developed by refining the top-level one. We add processes, process scheduling, and process management according to ARINC 653. Security is proven by refinement.
4) We conduct a code-to-spec review required by CC certification on one industrial and two open-source ARINC SK implementations, i.e. VxWorks 653, XtratuM [34], and POK [35], in accordance with the formal specification. During the verification and code review, six covert channels to leak information [36] have been found in the ARINC 653 standard and the implementations. The approaches to fixing them are also provided.

In this paper, we have extended our previous work [37] by introducing the security model, the refinement framework, the second-level specification, the code review of VxWorks 653, and four new covert channels. The rest of this paper is organized as follows. Section 2 introduces preliminaries of this paper. Section 3 presents the overview of our method. The next four sections present the security model, refinement framework, top-level and second-level specifications, respectively. Then, Section 8 discusses the security flaws found in ARINC 653 and the implementations. Finally, Section 9 gives the conclusion and future work.

2 Preliminaries

2.1 Information-flow Security

The notion *noninterference* is introduced in [38] in order to provide a formal foundation for the specification and analysis of information-flow security policies. The idea is that a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2672983, IEEE Transactions on Dependable and Secure Computing

security domain u is noninterfering with a domain v if no action performed by u can influence the subsequent outputs seen by v. Language-based information-flow security [21] assigns either *High* or *Low* labels to program variables and ensures the data confidentiality by preventing information leakage from *High*-level data to *Low*-level data. Transitive noninterference is too strong and thus is declassified as intransitive one [39], [40]. That is, the system should allow certain flows of information from *High* domains to *Low* domains, if that flow traverses the appropriate declassifier. In [20], intransitive noninterference is defined in a state-event manner and concerns the visibility of *events*, i.e. the secrets that events introduce in the system state. Language-based information-flow security is generalized to arbitrary multi-domain policies in [22] as a new state-event based notion *nonleakage*. In [22], nonleakage and intransitive noninterference are combined as a new notion *noninfluence*, which considers both the data confidentiality and the secrecy of events. Intransitive noninterference [20], [22], [41] in state-event manner has been applied to verify separation kernels, such as seL4 [16], PikeOS [11], INTEGRITY-178B [13], and AAMP7G [14].

2.2 ARINC 653

The ARINC 653 standard is organized in six parts. Part 1 [5] in Version 3 specifies the baseline operating environment for application software used within IMA. It defines the *system functionality* and requirements of 57 *services*. ARINC SKs are mandated to comply with this part. The six major functionalities specified in ARINC 653 are partition management, process management, time management, inter- and intra-partition communication, and health monitoring.

ARINC 653 establishes and separates multiple partitions in time and space except the controlled information flows along communication channels among partitions. The security policy used by ARINC SKs is the *Inter-Partition Flow Policy* (IPFP) [42], which is intransitive. It is expressed abstractly in a partition flow matrix $\mathbf{partition\_flow} : partition \times partition \rightarrow mode$, whose entries indicate the mode of the flow. For instance, $\mathbf{partition\_flow}(P_1, P_2) = QUEUING$ means that partition P_1 is allowed to send information to partition P_2 via a channel with a message queue. Another feature of ARINC 653 is its two-level scheduling, i.e. partition scheduling and process scheduling. For the purpose of temporal separation, the partition scheduling in ARINC 653 is a fixed, cycle based scheduling and is strictly deterministic over time. Process scheduling in a partition is priority preemptive.

2.3 Isabelle/HOL

Isabelle is a generic and tactic-based theorem prover. We use Isabelle/HOL [33], an implementation of higher-order logic in Isabelle, for our development. The keyword **datatype** is used to define an inductive data type. The list in Isabelle is one of the essential data type in computing and is defined as **datatype** $'a\ list = Nil\ (\text{“[]”}) \mid Cons\ 'a\ \text{“}\ 'a\ list\ \text{”}(infixr\ \text{“\#”})$, where $[\,]$ is the empty list and $\#$ is the concatenation. The polymorphic option type is defined as **datatype** $'a\ option = None \mid Some\ (the : 'a)$. A function of type $'b \rightharpoonup 'a$ models a partial function, which is equal to $'b \Rightarrow ('a\ option)$. Record types may be defined, for example **record** $point = xcoord :: int \quad ycoord :: int$ with the elements like $p = (\!|xcoord = 10, ycoord = 10|\!)$ and projections $xcoord\ p$ and $ycoord\ p$. Records may be updated, such as $p(\!|xcoord := 20, ycoord := 20|\!)$, and extended, such as **record** $cpoint = point + col :: colour$. Nonrecursive definitions can be made with the **definition** command and the **primrec** function definition is used for primitive recursions. *Locales* are Isabelle's approach for dealing with parametric theories. Locales may be instantiated by assigning concrete data to parameters, and the resulting instantiated declarations are added to the current context. This is called *locale interpretation*. In its simplest form, a **locale** declaration consists of a sequence of context elements declaring parameters (the keyword **fixes**) and assumptions (the keyword **assumes**).

To enhance readability, we will use standard mathematical notation where possible. Examples of formal specification will be given in the Isabelle/HOL syntax.

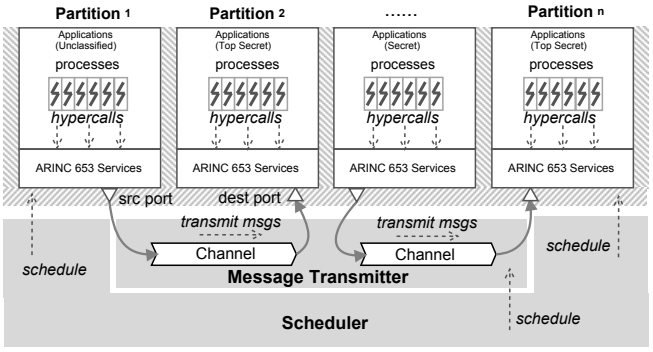

Fig. 1: Architecture of ARINC 653 Separation Kernels

3 Method Overview

The architecture of ARINC SKs we consider in this paper is shown in Fig. 1. Since ARINC 653 Part 1 in Version 3 [5] is targeted at single-core processing environments, we consider single-core separation kernels. For simplicity, separation kernels usually disable interrupts in kernel mode [16] and thus there is no in-kernel concurrency, which means that the hypercalls and system events (e.g. scheduling) are executed in an atomic manner.

We adopt intransitive noninterference in state-event manner due to its practicality in industry and intransitive channel-control policies used in ARINC 653. We formalize various definitions of information-flow security, e.g. noninterference, nonleakage, and noninfluence. An inference framework of these properties is provided, in which the implication relationship among the properties is proven.

Since automatic analysis of information-flow security is still restricted by the size of state space ([43], [44]), it is difficult to deal with operating system kernels so far. Automatic analysis techniques (e.g. model checking) usually verify one configuration of the system at a time [45]. However, the deployment of partitions on separation kernels is unknown in advance. Therefore, it is well suited to use theorem proving based approaches. Interactive theorem proving requires

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2672983, IEEE Transactions on Dependable and Secure Computing

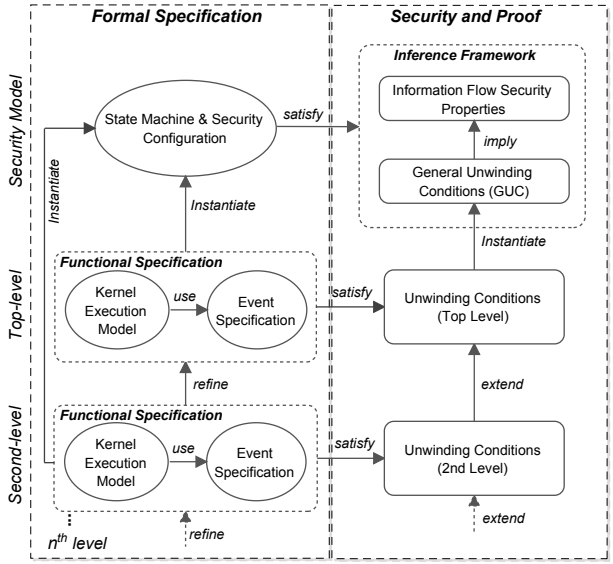

Fig. 2: Method Overview

human intervention and creativity, but produces machine-checkable proofs required by CC certification. Moreover, it is not constrained to specific properties or finite state spaces. This approach is mostly adopted to formal verification of operating systems [46]. We choose Isabelle/HOL in this work because (1) it can deal with large-scale formal verification of real-world systems [47] due to the expressiveness of HOL, a high degree of proof automation, and powerful libraries, etc.; (2) most of the OS verification are using Isabelle/HOL, e.g. seL4 [16], [28] and PikeOS [10], [11]; and (3) Isabelle/HOL is recommended as a formal methods tools as required by CC certification on high assurance levels [48]. Although other approaches, e.g. Event-B, provide the refinement framework, properties of information-flow security could not be formulated straightforwardly by their inherent specification languages. We could see some efforts on this direction [49] but no supported tools.

The method overview is shown in Fig. 2. The security model is a generic model of ARINC SKs for information-flow security, which includes a state machine based execution model and the security configuration. The inference framework presents the implication among these properties and an unwinding theorem. It follows the classical unwinding relation based proof [20], i.e. satisfaction of the general unwinding conditions (GUCs) implies the security. The security model is a parameterized abstraction, which means that the kernel components are defined as abstract types. It could be instantiated to a concrete model, i.e. a functional specification of ARINC SKs. By this instantiation, all elements (e.g. properties, implications, and proofs) of the security model are reused and preserved in the specification. In order to improve the reusablity of the specification and proofs, a functional specification is decomposed into two parts: a kernel execution model and an event specification for ARINC 653. The execution model is an instance of the security model, while the event specification defines Isabelle/HOL functions to implement the state changes when an event occurs. The concrete functions in the event specification are invoked by the execution model. By the instantiation, the properties of the inference framework are preserved on the specification. The rest of security proofs of the specification is to show satisfaction of unwinding conditions (UCs) which are instances of GUCs.

In order to support additional design elements in the refinement, we use superposition refinement [26] in this work. Since formal specification of ARINC SKs combines the system dynamics and security components, we extend the superposition refinement with security constraints for ARINC SKs. The state represented by a set of state variables at the abstract level is transformed into concrete state variables and extended by new state variables. An event is refined to a set of concrete events and new events may be introduced in the refinement. Therefore, in addition to a state simulation relation, we use an event relation to connect the abstract and concrete events. To show information-flow security at the concrete level, UCs of the abstract level are extended due to the new state variables and new events. The proofs of UCs at the concrete level only consider the new state variables and security proofs at the abstract level are reused. In the case of data refinement, i.e. without introducing new events and new state variables, security is automatically preserved in the refinement.

A major part of requirements in ARINC 653 are modeled in this work. We develop two levels of specification by refinement. ARINC SKs use IPC to implement controlled information flows among partitions. Moreover, in the IPFP policy of ARINC 653, communication ports and channels are associated with partitions, and all processes in a partition can access the ports configured for this partition. Therefore, we model functionalities related to information flows, i.e. partition management and IPC services, in the top-level functional specification. For functional completeness, kernel initialization and partition scheduling are also modeled. At the second level, we use the refinement to add process scheduling and process management services, which are functionalities in partitions. Other functionalities and services of ARINC 653 are possible to be added by refinement in future.

4 Security Model of Separation Kernels

We design a security model for ARINC SKs in this section. The model consists of a nondeterministic state machine, the security configuration of ARINC SKs, information-flow security properties, and an inference framework of these properties.

4.1 State Machine and Security Configuration

The state-event based information-flow security uses a state machine to represent the system model. For universality, we adopt a nondeterministic model. The state machine of ARINC SKs is defined as follows.

Definition 1 (State Machine of ARINC SKs). $\mathcal{M} = \langle \mathcal{S}, \mathcal{E}, \varphi, s_0 \rangle$ is a tuple, where $\mathcal{S}$ is the state space, $\mathcal{E}$ is the set of event labels, $s_0 \in \mathcal{S}$ is the initial state, and $\varphi : \mathcal{E} \to \mathbb{P}(\mathcal{S} \times \mathcal{S})$ is the state-transition function.

The φ function characterises the single-step behaviour of separation kernels by executing an event, such as a hypercall or scheduling. The auxiliary functions used in

$$\begin{cases} run([\,]) = Id & s \,\$\!\!\rightarrowtail es \triangleq \{s\} \lhd run(es) \\ run(e\#es) = \varphi(e) \circ run(es) & \mathcal{R}(s) \triangleq \exists es.\ (s_0, s) \in run(es) \end{cases}$$

$$s \overset{D}{\approx} t \triangleq \forall d \in D.\ s \overset{d}{\sim} t \qquad ss \overset{d}{\asymp} ts \triangleq \forall s\ t.\ s \in ss \wedge t \in ts \longrightarrow s \overset{d}{\sim} t$$

$$\begin{cases} sources([\,], s, d) = \{d\} \\ sources(e\#es, s, d) = (\bigcup\{sources(es, s', d) \mid (s, s') \in \varphi(e)\}) \cup \\ \qquad\qquad \{w \mid w = kdom(s, e) \wedge (\exists v\ s'.\ (w \rightsquigarrow v) \\ \qquad\qquad \wedge (s, s') \in \varphi(e) \wedge v \in sources(es, s', d))\} \end{cases}$$

$$\begin{cases} ipurge([\,], d, ss) = [\,] \\ ipurge(e\#es, d, ss) = \textbf{if} (\exists s \in ss.\ kdom(s, e) \in sources(e\#es, s, d)) \\ \qquad\qquad \textbf{then} \\ \qquad\qquad\quad e\#ipurge(es, d, (\bigcup_{s \in ss} \{s' \mid (s, s') \in \varphi(e)\})) \\ \qquad\qquad \textbf{else}\ \ ipurge(es, d, ss) \end{cases}$$

Fig. 3: Auxiliary Functions in Security Model

the security model are defined in detail in Fig. 3. Based on the φ function, we define the run function to represent the execution of a sequence of events. The $execution(s, es)$ function (denoted as $s \,\$\!\!\rightarrowtail es$) returns the set of final states by executing a sequence of events es from a state s, where $\lhd$ is the domain restriction of a relation. By the $execution$ function, the reachability of a state s is defined as $reachable(s)$ (denoted as $\mathcal{R}(s)$).

The security configuration of separation kernels is usually comprised of security domains, security policies, domain of events, and state equivalence as follows.

Security domains: Applications in partitions have various security levels. We consider each partition as a security domain. Since partition scheduling is strictly deterministic over time, we define a security domain *scheduler* for partition scheduling, which cannot be interfered by any other domains to ensure that the *scheduler* does not leak information via its scheduling decisions. Note that process scheduling is conducted in partitions. Since ARINC 653 defines the channel-based communication services using ports and leaves the implementation of message transmission over channels to underlying separation kernels, a security domain *message transmitter* is defined for message transmission. The transmitter also decouples message transmission from the scheduler to ensure that the scheduler is not interfered by partitions. Therefore, the security domains ($\mathcal{D}$) of ARINC SKs are the scheduler ($\mathbb{S}$), the transmitter ($\mathbb{T}$), and the configured partitions ($\mathcal{P}$), i.e. $\mathcal{D} = \mathcal{P} \cup \{\mathbb{S}, \mathbb{T}\}$.

Security policies: In order to discuss information-flow security policies, we assume an *interference relation* $\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$ according to the **partition_flow** matrix and $\not\rightsquigarrow$ is the complement relation of $\rightsquigarrow$. If there is a channel from a partition p_1 to a partition p_2, then $p_1 \rightsquigarrow \mathbb{T}$ and $\mathbb{T} \rightsquigarrow p_2$ since the transmitter is the message intermediator. Since the scheduler can possibly schedule other domains, it can interfere with them. In order to prevent the scheduler from leaking information via its scheduling decisions, we require that no other domains can interfere with the scheduler.

Domain of events: Traditional formulations in the state-event based approach assume a static mapping from events to domains, such that the domain of an event can be determined solely from the event itself [20]. However, in separation kernels that mapping is dynamic [23]. When a *hypercall* occurs, the kernel must consult the partition scheduler to determine which partition is currently executing, and the current executing partition is the domain of the hypercall. On the other hand, separation kernels can provide specific hypercalls which are only available to privileged partitions. Therefore, the domain of an event is represented as a partial function $kdom : \mathcal{S} \times \mathcal{E} \rightharpoonup \mathcal{D}$.

State equivalence: It means states are *identical* for a security domain. We define an equivalence relation $\sim$ on states for each domain such that $s \overset{d}{\sim} t$ if and only if states s and t are identical for domain d, that is to say states s and t are indistinguishable for d. Two states are equivalent for the scheduler when domains of each event in the two states are the same. For a set of domains D, we define $s \overset{D}{\approx} t$ as shown in Fig. 3 to represent that states s and t are equivalent for all domains in D. For two sets of states ss and ts, we define $ss \overset{d}{\asymp} ts$ as shown in Fig. 3 to represent that any two states in ss and ts are equivalent for domain d.

Based on the discussion above, we define the security model of ARINC SKs as follows. We assume that each event is always enabled in a state. Whenever an event e should not be executed in a state s, it does not change the state.

Definition 2 (Security Model of ARINC SKs).

$$\mathcal{SM} = \langle \mathcal{M}, \mathcal{D}, kdom, \rightsquigarrow, \sim \rangle$$

with assumptions as follows.

1) $\forall p_1, p_2 \in \mathcal{P}.\ p_1 \rightsquigarrow p_2 \longrightarrow p_1 \rightsquigarrow \mathbb{T} \wedge \mathbb{T} \rightsquigarrow p_2$
2) $\forall d \in \mathcal{D}.\ \mathbb{S} \rightsquigarrow d$
3) $\forall d \in \mathcal{D}.\ d \rightsquigarrow \mathbb{S} \longrightarrow d = \mathbb{S}$
4) $\sim$ is an equivalence relation.
5) $\forall s\ t\ e.\ s \overset{\mathbb{S}}{\sim} t \longrightarrow kdom(s, e) = kdom(t, e)$
6) events are enabled in any state, i.e. $\forall s\ e.\ \mathcal{R}(s) \longrightarrow (\exists s'.\ (s, s') \in \varphi(e))$.

The security model is represented in Isabelle/HOL as a locale $\mathcal{SM}$, which could be instantiated to create functional specification.

4.2 Information-flow Security Properties

In order to express the allowed information flows for the intransitive policies, we use a function $sources(es, s, d)$ as shown in Fig. 3, which yields the set of domains that are allowed to pass information to domain d when event sequence es occurs from state s. It is inductively defined on event sequences. We include in $sources(e\#es, s, d)$ all domains that can pass information to d when es occurs from all successor states s' of s, as well as the domain $kdom(s, e)$ performing the event e, whenever there exists some intermediate domain v by which the domain $kdom(s, e)$ can pass information to d indirectly. In the intransitive purged sequence ($ipurge(es, d, ss)$ in Fig. 3), the events of partitions that are not allowed to pass information to d directly or indirectly are removed. Given an event sequence $e\#es$ executing from a set of state ss, $ipurge$ keeps the first event e if this event is allowed to affect the target domain d, i.e. $\exists s \in ss.\ kdom(s, e) \in sources(e\#es, s, d)$. It then continues on the remaining events es from the successor states of ss after executing e. On the other hand, if e is not allowed to affect the target domain d, then it is removed from the sequence and purging continues on the remaining events es

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2672983, IEEE Transactions on Dependable and Secure Computing

TABLE 1: Information-flow Security Properties

No.	Properties
(1)	**noninterference**: $\forall d\ es.\ (s_0 \upharpoonright\!\!\rightarrow es) \overset{d}{\asymp} (s_0 \upharpoonright\!\!\rightarrow ipurge(es, d, \{s_0\}))$
(2)	**weak_noninterference**: $\forall d\ es_1\ es_2.\ ipurge(es_1, d, \{s_0\}) = ipurge(es_2, d, \{s_0\}) \longrightarrow ((s_0 \upharpoonright\!\!\rightarrow es_1) \overset{d}{\asymp} (s_0 \upharpoonright\!\!\rightarrow es_2))$
(3)	**noninterference_r**: $\forall d\ es\ s.\ \mathcal{R}(s) \longrightarrow ((s \upharpoonright\!\!\rightarrow es) \overset{d}{\asymp} (s \upharpoonright\!\!\rightarrow ipurge(es, d, \{s\}))$
(4)	**weak_noninterference_r**: $\forall d\ es_1\ es_2\ s.\ \mathcal{R}(s) \wedge ipurge(es_1, d, \{s\}) = ipurge(es_2, d, \{s\}) \longrightarrow ((s \upharpoonright\!\!\rightarrow es_1) \overset{d}{\asymp} (s \upharpoonright\!\!\rightarrow es_2))$
(5)	**nonleakage**: $\forall d\ es\ s\ t.\ \mathcal{R}(s) \wedge \mathcal{R}(t) \wedge (s \overset{\mathbb{S}}{\sim} t) \wedge (s \overset{sources(es,s,d)}{\approx} t) \longrightarrow ((s \upharpoonright\!\!\rightarrow es) \overset{d}{\asymp} (t \upharpoonright\!\!\rightarrow es))$
(6)	**weak_noninfluence**: $\forall d\ es_1\ es_2\ s\ t.\ \mathcal{R}(s) \wedge \mathcal{R}(t) \wedge (s \overset{sources(es_1,s,d)}{\approx} t) \wedge (s \overset{\mathbb{S}}{\sim} t) \wedge ipurge(es_1, d, \{s\}) = ipurge(es_2, d, \{t\}) \longrightarrow ((s \upharpoonright\!\!\rightarrow es_1) \overset{d}{\asymp} (t \upharpoonright\!\!\rightarrow es_2))$
(7)	**noninfluence**: $\forall d\ es\ s\ t.\ \mathcal{R}(s) \wedge \mathcal{R}(t) \wedge (s \overset{sources(es,s,d)}{\approx} t) \wedge (s \overset{\mathbb{S}}{\sim} t) \longrightarrow ((s \upharpoonright\!\!\rightarrow es) \overset{d}{\asymp} (t \upharpoonright\!\!\rightarrow ipurge(es, d, \{t\})))$

from the current set of states ss. The observational equivalence of an execution is denoted as $(s \upharpoonright\!\!\rightarrow es_1) \overset{d}{\asymp} (t \upharpoonright\!\!\rightarrow es_2)$, which means that a domain d is identical to any two final states after executing es_1 from s ($s \upharpoonright\!\!\rightarrow es_1$) and executing es_2 from t.

The intransitive noninterference [20] on the execution model is defined as *noninterference* in Table 1. Its intuitive meaning is that events of domains that cannot interfere with a domain d should not affect d. Formally, given an event sequence es and a domain d, final states after executing es from the initial state s_0 and after executing its purged sequence from s_0 are identical to d. In *noninterference*, the $ipurge$ function only deletes all unsuitable events. Another version is introduced in [22] to handle arbitrary insertion and deletion of secret events, which is defined as *weak_noninterference* in Table 1. In *weak_noninterference*, we only ask for two event sequences which have the same effects on the target domain d, i.e. $ipurge(es_1, d, \{s_0\}) = ipurge(es_2, d, \{s_0\})$. The above definitions of noninterference are based on the initial state s_0, 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 *noninterferece_r* based on the reachable function $\mathcal{R}$. This general noninterference requires that systems starting from any reachable state are secure. It is obvious that this noninterference implies the classical noninterference due to $\mathcal{R}(s_0) = True$. *Weak_noninterference* is also generalized as a new property *weak_noninterference_r*.

Nonleakage and noninfluence are defined for ARINC SKs based on the scheduler as shown in Table 1. The intuitive meaning of nonleakage is that if data are not leaked initially, data should not be leaked during executing a sequence of events. Separation kernels are said to preserve nonleakage

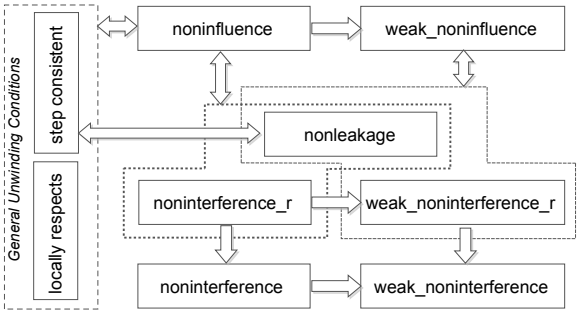

Fig. 4: Inference Framework of Security Properties

when for any pair of reachable states s and t and an observing domain d, if (1) s and t are equivalent for all domains that may (directly or indirectly) interfere with d during the execution of es, i.e. $s \overset{sources(es,s,d)}{\approx} t$, and (2) the same domain is currently executing in both states, i.e. $s \overset{\mathbb{S}}{\sim} t$, then s and t are observationally equivalent for d and es. Noninfluence is the combination of nonleakage and noninterference to ensure that there is no secrete data leakage and secrete events are not visible according to security policies. Murray et al. [23] have defined noninfluence for seL4, which is a weaker one and defined as *weak_ noninfluence* in Table 1. Similar to *weak_noninterference*, it can handle arbitrary insertion and deletion of secret events. It is the combination of *weak_noninterference_r* and *nonleakage*. We propose a stronger one, i.e. *noninfluence*, according to the original definition [22] by extending the scheduler and the state reachability. It is the combination of *noninterference_r* and *nonleakage*. In the next subsection, we will show that noninfluence is actually the conjunction of noninterference and nonleakage.

4.3 Inference Framework of Security Properties

The inference framework clarifies the implication relations among the security properties on the security model. We have proven all implication relations as shown in Fig. 4, where an arrow means the implication between properties.

The standard proof of information-flow security is discharged by proving a set of unwinding conditions [20] that examine individual execution steps of the system. This paper also follows this approach. We first define the unwinding conditions on event (UCE) as follows.

Definition 3 (Step Consistent Condition of Event).

$$SC(e) \triangleq \forall d\ s\ t.\ \mathcal{R}(s) \wedge \mathcal{R}(t) \wedge (s \overset{d}{\sim} t) \wedge (s \overset{\mathbb{S}}{\sim} t) \wedge (kdom(s, e) \rightsquigarrow d) \wedge (s \overset{kdom(s,e)}{\sim} t) \longrightarrow (\forall s'\ t'.\ (s, s') \in \varphi(e) \wedge (t, t') \in \varphi(e) \longrightarrow s' \overset{d}{\sim} t')$$

Definition 4 (Locally Respects Condition of Event).

$$LR(e) \triangleq \forall d\ s\ s'.\ \mathcal{R}(s)\ \wedge (kdom(s, e) \not\rightsquigarrow d) \wedge (s, s') \in \varphi(e) \longrightarrow (s \overset{d}{\sim} s')$$

The *step consistent* condition requires that for any pair of reachable states s and t, and any observing domain d, the next states after executing an event e in s and t

are indistinguishable for d, i.e. $s' \overset{d}{\sim} t'$, if (1) s and t are indistinguishable for d, (2) the current executing domains in s and t are the same, (3) the domain of e in state s can interference with d, and (4) s and t are indistinguishable for the domain of e. The *locally respects* condition means that an event e that executes in a state s can affect only those domains to which the domain executing e is allowed to send information.

Based on the UCEs, the general unwinding conditions of the security model can be defined as $SC \triangleq \forall e.\ SC(e)$ and $LR \triangleq \forall e.\ LR(e)$. The soundness and completeness of unwinding conditions in the security model are shown as follows, where the soundness is the unwinding theorem in [20], [22].

Theorem 1 (Soundness and Completeness of Unwinding Conditions). The security model $\mathcal{SM}$ satisfies that

$$(SC \wedge LR) = noninfluence \textbf{ and } SC = nonleakage$$

The objective of noninfluence is to ensure data confidentiality and secrecy of events by combing noninterference and nonleakage. We show the soundness and completeness of noninfluence in the security model as follows.

Theorem 2 (Soundness and Completeness of Noninfluence). The security model $\mathcal{SM}$ satisfies that

$$noninfluence = (noninterference\_r \wedge nonleakage) \textbf{ and}$$
$$weak\_noninfluence = (weak\_noninterference\_r \wedge nonleakage)$$

5 Specification and Refinement Framework

After discussing the security model, we present the functional specification by instantiating the security model and a refinement framework for stepwise specification development in this section.

5.1 Specification by Security Model Instantiation

A functional specification consists of a kernel execution model and an event specification. The kernel execution model is an instance of the security model. The instantiation is an interpretation of the $\mathcal{SM}$ locale, i.e. $SM_I = \textbf{interpretation}\ \mathcal{SM}\{s_0/s_{0I}, \varphi/\varphi_I, kdom/kdom_I, \mathbb{S}/Sched, \mathbb{T}/Trans, \sim\ /\ \sim_I, \rightsquigarrow\ /\ \rightsquigarrow_I\}$, where the parameters of $\mathcal{SM}$ are substituted by concrete ones. In order to assure the correctness of the instantiation, we provide the *instantiation proof* to show that the instance parameters preserve the assumptions of $\mathcal{SM}$.

The event specification is a mapping from events to a set of functions to model the functionalities and services in ARINC 653. The φ_I function in the execution model executes events by invoking these functions. Thus, a functional specification of ARINC SKs is defined as follows.

Definition 5 (Functional Specification of ARINC SKs). A functional specification is a tuple $SK = \langle SM_I, F\rangle$, where

1) $SM_I = \langle \mathcal{M}_I, \mathcal{D}_I, kdom_I, \rightsquigarrow_I, \sim_I\rangle$ is the execution model, where $\mathcal{M}_I = \langle \mathcal{S}_I, \mathcal{E}_I, \varphi_I, s_{0I}\rangle$.
2) $F : \mathcal{E}_I \to \Gamma$ is the event specification, where Γ is the set of functions and each $f \in \Gamma$ has the type $\mathcal{S}_I \to \mathbb{P}(\mathcal{S}_I)$.
3) φ_I in $\mathcal{M}_I$ defines the execution of each event e, where $\varphi_I(e) = \{(s,t) \mid t \in F(e)(s)\}$.

5.2 Specification Refinement

In the superposition refinement, existing state variables and events can be refined. In addition, new state variables and events can be added to an abstract specification. Here, we use subscripts A and C to represent the abstract and concrete specifications, respectively. During the refinement the abstract state $\mathcal{S}_A$ can be transformed into $\mathcal{S}_T$ and new state variables denoted as $\mathcal{S}_\Delta$ can be incorporated. Thus, the concrete state $\mathcal{S}_C = \mathcal{S}_T + \mathcal{S}_\Delta$, where the symble "+" is the extension of data type (e.g. **record** extension in Isabelle/HOL). The security extended superposition refinement for the functional specification is defined as follows.

Definition 6 (Refinement). Given two functional specifications SK_A and SK_C, SK_C refines SK_A using a state simulation relation $\Psi : \mathcal{S}_C \to \mathcal{S}_A$ and an event relation $\Theta : \mathcal{E}_C \to (\mathcal{E}_A \cup \{\tau\})$, denoted as $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, if

1) the initial state in SK_C establishes R, i.e. $\Psi(s_{0C}) = s_{0A}$.
2) each event in $\mathcal{E}_A$ is refined by a set of events in $\mathcal{E}_C$, i.e. Θ is a surjection and $\forall e\ s\ t.\ \Theta(e) \neq \tau \wedge (s,t) \in \varphi_C(e) \longrightarrow (\Psi(s), \Psi(t)) \in \varphi_A(\Theta(e))$.
3) new events only change the new state variables introduced in the refinement and does not affect the variables in SK_A, i.e. $\forall e\ s\ t.\ \Theta(e) = \tau \wedge (s,t) \in \varphi_C(e) \longrightarrow \Psi(t) = \Psi(s)$.
4) security domains are preserved and refined events has the same execution domain as that at the abstract level, i.e. $\mathcal{D}_C = \mathcal{D}_A$ and $\forall e\ s.\ \Theta(e) \neq \tau \longrightarrow kdom_C(s,e) = kdom_A(\Psi(s), \Theta(e))$.
5) the refinement does not change the interference relation, i.e. $\rightsquigarrow_C = \rightsquigarrow_A$.
6) $s \overset{d}{\sim}_C t = \Psi(s) \overset{d}{\sim}_A \Psi(t) \wedge s \overset{d}{\sim}_\Delta t$, where $\sim_\Delta$ only considers the new state variables $\mathcal{S}_\Delta$ introduced in the refinement.

The relation Ψ maps concrete states to abstract ones. It is actually the transformation of S_A to S_T. Abstract states of next states by executing an event e in a state t at the concrete level is a subset of next states by executing the refined event $\Theta(e)$ in a state s ($s = \Psi(t)$) at the abstract level. If the event is a new one at the concrete level, its execution should not affect the abstract state. We could consider that a new event at the concrete level refines a τ action, which does not change the abstract states. The security configuration at the abstract level is preserved in the refinement, i.e. (1) $\mathbb{S}$, $\mathbb{T}$, partitions $\mathcal{D}$, and the relation $\rightsquigarrow$ are the same, (2) the domain of events at the abstract level is preserved, and (3) the state equivalence relation at the abstract level is preserved, while the relation at the concrete level also requires that the two states are equivalent for the new state variables (i.e. $\sim_\Delta$).

The refinement in this paper is reflexive and transitive. The correctness is ensured by Theorem 3. Since noninterference considers the state and event together, our refinement implies that the state-event trace set of a concrete specification is a subset of the abstract one. Thus, the refinement preserves noninterference as we present in the next subsection.

Theorem 3 (Soundness of Refinement). If SK_A and SK_C are instances of $\mathcal{SM}$ and $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, then

$$\forall es.\ \Psi^*(s_{0C} \nrightarrow es) \subseteq s_{0A} \nrightarrow \Theta^*(es)$$

In the theorem, Ψ^* maps a set of concrete states to a set of abstract states by the relation Ψ, and Θ^* maps a sequence of concrete events to a sequence of abstract events by the relation Θ. The τ event is ignored during executing a sequence of events at the abstract level.

From the definition of the refinement, we could see that for any reachable state s at the concrete level, its abstract state on the Ψ relation is also reachable at the abstract level.

Lemma 1 (State Reachability in Refinement). If SK_A and SK_C are instances of $\mathcal{SM}$ and $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, then

$$\forall s.\ \mathcal{R}_C(s) \longrightarrow \mathcal{R}_A(\Psi(s))$$

Superposition refinement [26] assumes the termination of new events introduced in the refinement. Since hypercalls and system events of ARINC SKs terminate, we assume that events terminate in our refinement framework. In the formal specification of ARINC SKs, we use the **primrec** and **definition** in Isabelle/HOL to define the event specification. Thus, the termination of events is automatically ensured in Isabelle/HOL.

5.3 Security Proofs of Refinement

The security proofs of a concrete specification also follow the unwinding theorem introduced in Subsection 4.3. Since a concrete specification is an instance of the security model, we have following theorem according to Theorem 1.

Theorem 4 (Unwinding Theorem for Concrete Specification). If SK_C is an instances of $\mathcal{SM}$ and SK_C satisfies $SC_C \wedge LR_C$, then SK_C satisfies $noninfluence$.

We define the step consistent condition of events on the new state variables as follows.

Definition 7 (Step Consistent Condition of Event on New State Variables).

$$\begin{aligned} SC_{C\Delta}(e) \triangleq \forall\ d\ s\ t.\ \mathcal{R}_C(s) \wedge \mathcal{R}_C(t) \wedge (s \sim_C^d t) \wedge (s \sim_C^{\mathbb{S}} t) \\ \wedge\ (kdom_C(s,e) \rightsquigarrow_C d) \wedge (s \sim_C^{kdom_C(s,e)} t) \\ \longrightarrow (\forall s'\ t'.\ (s,s') \in \varphi_C(e) \wedge (t,t') \in \varphi_C(e) \longrightarrow s' \sim_\Delta^d t') \end{aligned}$$

The locally respects condition of events on the new state variables is defined in an analogous manner. Based on definitions of the refinement and the unwinding conditions as well as Lemma 1, we have four lemmas shown as follows.

Lemma 2 (Step Consistent of Event Refinement). If SK_A and SK_C are instances of $\mathcal{SM}$ and $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, then

$$\forall\ e.\ \Theta(e) \neq \tau \wedge SC_A(\Theta(e)) \wedge SC_{C\Delta}(e) \longrightarrow SC_C(e)$$

Lemma 3 (Locally Respects of Event Refinement). If SK_A and SK_C are instances of $\mathcal{SM}$ and $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, then

$$\forall\ e.\ \Theta(e) \neq \tau \wedge LR_A(\Theta(e)) \wedge LR_{C\Delta}(e) \longrightarrow LR_C(e)$$

Lemma 4 (Step Consistent of New Event in Refinement). If SK_A and SK_C are instances of $\mathcal{SM}$ and $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, then

$$\forall\ e.\ \Theta(e) = \tau \wedge SC_\Delta(e) \longrightarrow SC_C(e)$$

Lemma 5 (Locally Respects of New Event in Refinement). If SK_A and SK_C are instances of $\mathcal{SM}$ and $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, then

$$\forall\ e.\ \Theta(e) = \tau \wedge LR_\Delta(e) \longrightarrow LR_C(e)$$

Finally, based on these lemmas and Theorem 4, we have Theorem 5. The theorem means that if SK_C is a refinement of SK_A and SK_A satisfies the unwinding conditions, we only need to prove the unwinding conditions on new state variables to show information-flow security of SK_C.

Theorem 5 (Security of Refinement). If SK_A and SK_C are instances of $\mathcal{SM}$, SK_A satisfies $SC_A \wedge LR_A$, SK_C satisfies $SC_\Delta \wedge LR_\Delta$, and $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, then SK_C satisfies $noninfluence$.

In the case of data refinement, i.e. without new events and new state variables introduced in the refinement, information-flow security is automatically preserved on the concrete specification.

Corollary 1. If SK_A and SK_C are instances of $\mathcal{SM}$, SK_A satisfies $SC_A \wedge LR_A$, $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$, $\nexists e.\ \Theta(e) = \tau$, and $\forall s\ t\ d.\ s \sim_\Delta^d t$, then SK_C satisfies $noninfluence$.

6 Top-level Specification and Security Proofs

In the top-level functional specification, we model kernel initialization, partition scheduling, partition management, and inter-partition communication defined in ARINC 653. We first instantiate the security model as the kernel execution model. Then, we present the event specification. Finally, security proofs are discussed.

6.1 Kernel Execution Model

Basic components at the top level are partitions and communication objects. A *partition* is basically the same as a program in a single application environment [5]. IPC is conducted via messages on channels configured among partitions. Partitions have access to channels via *ports* which are the endpoints of channels. The modes of transferring messages over channels are *queuing* and *sampling*. A significant characteristic of ARINC SKs is that the basic components are statically configured at built-time. Partitions, communication objects, and the system configuration are specified in Isabelle/HOL as "**record** *Sys_Config*" and defined as a constant "*conf::Sys_Config*" in the top-level specification. When creating ports by invoking IPC services in a partition, only configured ports in the partition are created. A set of configuration constraints are defined to ensure the correctness of the system configuration.

We first instantiate the security model by a set of concrete parameters as follows.

Events: We consider two types of events in the specification: *hypercalls* and *system events*. Hypercalls cover all partition management and IPC services in ARINC 653. System events are the actions of the kernel itself and include kernel initialization, scheduling, and transmitting messages over channels. Other types of events could be introduced during subsequent refinements. Events are illustrated in Fig. 1 as dotted line arrows and italics.

Kernel State and Transition: In the top-level specification, the kernel state concerns states of partitions, the scheduler, and ports. The state of a partition consists of its operating mode ($partitions$) and the created ports ($part\_$

$ports$). The state of the scheduler shows which is the current executing partition (cur). The state of a port is mainly about messages in its buffer ($comm$). The **datatype** $Port\_Type$ models sampling and queuing mode ports as well as their message buffers. We define the type of $port\_id$ as natural number and use the partial functions to store the created ports and their states. We prove a set of invariants about types in the specification, such as the domain of $part\_ports$ is the same as that of $comm$ in all reachable states.

record *State* = *partitions* :: *part_id* ⇀ *part_mode_type*
part_ports :: *port_id* ⇀ *part_id*
cur :: *domain_id* *comm* :: *port_id* ⇀ *Port_Type*

The state transition function φ is instantiated as the $exec\_event$ function in the top-level specification, in which the mapping function F in Definition 5 is also modeled.

Event Domain: The domain of the system events is static: the domain of the event *scheduling* is $\mathbb{S}$ and the domain of *message transmission* is $\mathbb{T}$. The domain of hypercalls is dynamic and depends on the current state of the kernel. It is defined as $kdom\ s\ (\mathbf{hyperc}\ h) = cur\ s$, where $cur\ s$ returns the current executing partition in the state s.

Interference Relation: The interference relation in the security model is instantiated as the function *interference1* as follows in the top-level specification.

definition *interference1* :: *domain-id* ⇒ *domain-id* ⇒ *bool* ((- ⇝ -))
where *interference1 d1 d2* ≡
if *d1* = *d2* ***then*** *True*
else if *is-sched conf d1* ***then*** *True*
else if ¬(*is-sched conf d1*) ∧ (*is-sched conf d2*) ***then*** *False*
else if *is-part conf d1* ∧ *is-trm conf d2* ***then*** *part-intf-trm conf d1*
else if *is-trm conf d1* ∧ *is-part conf d2* ***then*** *trm-intf-part conf d2*
else *False*

The $interference1$ is in conformance with the assumptions 1) - 3) in Definition 2.

State Equivalence: For a partition d, $s \overset{d}{\sim} t$ if and only if $vpeq\_part\ s\ d\ t$, which is defined as follows.

definition *vpeq_part* :: "*State* ⇒ *part_id* ⇒ (*State* × *bool*)" **where**
vpeq_part s d t ≡ *(partitions s) d* = *(partitions t) d* ∧ *vpeq_part_comm s d t*

It means that states s and t are equivalent for partition d, when the partition state and the 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, partition d has the same ability on p in s as in t. This is because d is able to receive a message from p in the 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 current executing partition in the two states are the same. The equivalence of states for the transmitter requires that all ports and states of the ports are the same.

6.2 Event Specification

The event specification on the top level models kernel initialization, partition scheduling, all services of IPC and partition management defined in ARINC 653.

Kernel Initialization and Scheduling: The kernel initialization considers initialization of the kernel state, which is

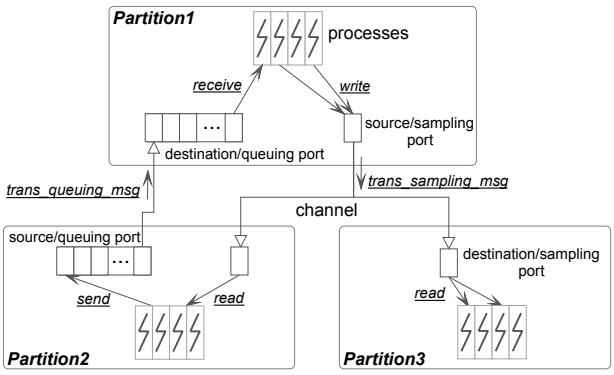

Fig. 5: Channel-based Communication in ARINC 653

defined as $s_0 = system\_init\ conf$. The $system\_init$ function assigns initial values to the kernel state according to the kernel configuration. Because the execution of message transmission is also under the control of scheduling, we define an abstract partition scheduling as follows that non-deterministically chooses one partition or the transmitter as the current executing domain, where *partconf* is a field of *Sys_Config* with the type $partition\_id \rightharpoonup Partition\_Conf$ to store the configured partitions.

definition *schedule* :: "*Sys_Config* ⇒ *State* ⇒ *State*" **where**
schedule sc s ≡ *s*(|*cur* := SOME *p*. *p*∈ {*x*. (*partconf sc*) *x* ≠ *None* ∨ *x* = T *sc*}|)

Partition Management: Partition management services in ARINC 653 are available to the application software for setting a partition's operating mode and to obtain a partition's status. The *Set_Partition_Mode* service request is used to set the operating mode of the current partition to *NORMAL* after the initialization of the partition is complete. The service is also used for setting the partition back to *IDLE* (partition shutdown), and to *COLD_START* or *WARM_START* (partition restart), when a fault is detected and processed. The *Set_Partition_Mode* service is specified as follows. The partition mode transition in ARINC 653 does not allow transitting from *COLD_START* to *WARM_START*. The function updates the operating mode of the current executing partition.

definition *set_part_mode* :: *Sys_Config* ⇒ *State* ⇒ *part_mode_type* ⇒ *State*
where
set_part_mode sc s m ≡
(***if*** (*partitions s*) (*cur s*) ≠ *None* ∧
¬ (*the* ((*partitions s*) (*cur s*)) = *COLD_START*
∧ *m* = *WARM_START*) ***then***
let *pts* = *partitions s* ***in*** *s*(|*partitions* := *pts*(*cur s* := *Some m*)|)
else *s*)

Inter-partition Communication: The communication architecture is illustrated in Fig. 5. 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 8 in detail). We change the service specification in ARINC 653 by allowing message loss to avoid these covert channels according to the results in [50].

We use a set of functions to implement one service. For instance, the *Send_Queuing_Message* service is implemented by the $send\_que\_msg\_lost$ function as follows. The

manipulated port should be a source (*is_source_port s p*) and queuing (*is_a_queuingport s p*) port. It should belong to current partition too (*is_a_port_of_partition s p*). If the message buffer of the port is full (*is_full_portqueuing*), the service just omits the message. Otherwise, the message is inserted into the buffer (*insert_msg2queuing_port*).

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

The message transmission on channels is shown in Fig. 5. For instance, the message transmission in queuing mode is specified as follows. If the source and destination port have been created (*get_portid_by_name s sn* ≠ *None*) and there are messages in the buffer of the source port (*has_msg_inportqueuing*), a message in the buffer is removed (*remove_msg_from_queuingport*) and inserted into the buffer of the destination port. When the buffer of the destination port is full (*is_full_portqueuing*), the message is discarded.

primrec *transf_que_msg_lost* :: "*Sys_Config* ⇒ *State* ⇒ *Channel_Type* ⇒ *State*" **where**
transf_que_msg_lost sc s (*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_que_msg_lost sc s (*Sampling* _ _ _) = *s*

6.3 Security Proofs

Since the top-level specification is an instance of the security model, the first part of the security proofs is the instantiation proof. The assumptions 1) - 3) of the security model (Definition 2) on the interference relation are preserved by the $interference1$ function. The assumption 4) is preserved by $\sim$ for the scheduler. The definition of $\sim$ is an equivalence relation at the top level, which means the preservation of the assumption 5). By the following lemma, the assumption 6) of the security model is preserved by the top-level specification.

Lemma 6 (Event Enabled in Top-level Specification).

$$\forall s\ e.\ \mathcal{R}(s) \longrightarrow (\exists s'.\ (s, s') \in exec\_event(e))$$

The second step of the security proofs is to show the UCEs, i.e. satisfaction of Definitions 3 and 4. We define a set of *concrete conditions* for events. Satisfaction of the concrete conditions of one event implies that the event satisfies the UCEs. For instance, Definition 8 shows the concrete condition of step consistent for the *Create_Queuing_Port* event, which is an instance of UCEs of the event.

Definition 8 (Concrete $SC(e)$ of Creating_Queuing_Port).

$$\begin{aligned}\forall\ & d\ s\ t\ s'\ t'\ p.\ \mathcal{R}(s) \wedge \mathcal{R}(t) \wedge s \overset{d}{\sim} t \wedge s \overset{\mathbb{S}}{\sim} t \\ & \wedge is\_part\ conf\ (cur\ s) \wedge (cur\ s) \rightsquigarrow d \wedge s \overset{cur\ s}{\sim} t \\ & \wedge s' = fst\ (create\_queuing\_port\ conf\ s\ p) \\ & \wedge t' = fst\ (create\_queuing\_port\ conf\ t\ p) \\ & \longrightarrow s' \overset{d}{\sim} t'\end{aligned}$$

Finally, we conclude the satisfaction of the *noninfluence* property on the top-level specification and all other information-flow security properties according to the inference framework of the security model.

7 Second-level Specification and Security Proofs

In the second-level specification, we refine the top-level one by adding processes, process scheduling in partitions, and process management services in ARINC 653. We first instantiate the security model as the kernel execution model. Then, we present the refinement. Finally, the security proofs of the specification are discussed.

7.1 Kernel Execution Model

In ARINC 653, a partition comprises one or more processes that combine dynamically to provide the functions associated with that partition [5]. Processes are separated among partitions, while processes within a partition share the resources of the partition. Process management provides the services to control the life-cycle of processes. We refine the state at the top-level as follows. The state variables at the top level are not changed. The state is extended by new state variables of processes by means of **record** extension. Thus, the state relation R is simple and defined as follows. A partition has a set of created processes ($procs$) and may have a current executing process (cur_proc_part). A process has a state ($proc\_state$). We found a covert channel if we use global process identifiers (Section 8 in detail). Here, we use separated process identifiers for each partition.

record *StateR* = *State* +
procs :: *partition-id* ⇀ (*process-id set*)
cur-proc-part :: *partition-id* ⇀ *process-id*
proc-state :: *partition-id* × *process-id* ⇀ *Proc-State*

definition *R* :: *StateR* ⇒ *State* (⇑- [50]) **where**
R r = (|*cur* = *cur r*, *partitions* = *partitions r*,
comm = *comm r*, *part-ports* = *part-ports r* |)

Each event at the top level is refined by one at the second level. We add events for process scheduling and process management services. The $exec\_event$ function at the top level is extended by adding maps of new events to new functions in the event specification. The event domain and the interference relation at the second level are the same as those at the top level. The state equivalence relation is extended on new state variables as $s\ .\overset{d}{\sim}.\ t$ as follows. The state equivalence on new state variables, i.e. $s\ \overset{d}{\sim}_{\Delta}\ t$ in Definition 6, is defined as $s\ .\overset{d}{\sim}_{\Delta}.\ t$ at the second level. It requires that the processes of a partition d, the states of processes, and the current executing process in the partition are the same in states s and t.

definition *vpeqR* :: *StateR* ⇒ *domain-id* ⇒ *StateR* ⇒ *bool* ((- ∼. - .∼ -))
where *vpeqR s d t* ≡ (*R s*) ∼ *d* ∼ (*R t*) ∧ (*s*∼*d*.∼$_{\Delta}$*t*)

7.2 Event Specification

Due to the new events introduced in the refinement, we use new functions to implement the process management

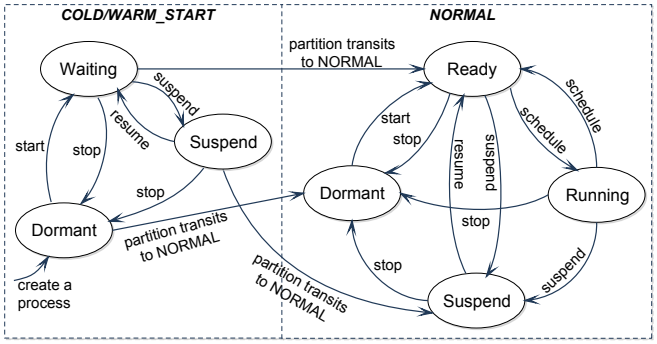

Fig. 6: Process State Transitions in Second-level Specification

and scheduling. The existing functions in the top-level specification are refined according to the new state type. The process state transitions in the second-level specification are shown in Fig. 6. We capture all state transitions defined in ARINC 653 except those triggered by intra-partition communication, which will be modeled in future refinement.

Process Management: Seven basic services of process management in ARINC 653, i.e. *Create/Start/Stop/Suspend/Resume_Process*, *Set_Priority*, and *Get_Process_Status*, are modeled at the second level. We use a set of functions to implement these services. For instance, *Start_Process* is implemented by the $start\_process$ function as follows.

definition *start-process* :: *StateR* ⇒ *process-id* ⇒ *StateR* **where**
start-process s p ≡
  if *p* ∈ *the* ((*procs s*) (*cur s*)) ∧
    (*state* (*the* ((*proc-state s*) (*cur s*, *p*)))) = *DORMANT* ***then***
      let *st* = (*if the* ((*partitions s*) (*cur s*)) = *NORMAL*
            then READY else WAITING);
        pst = *the* ((*proc-state s*) (*cur s*, *p*));
        proc-state′ = (*proc-state s*) ((*cur s*, *p*) :=
                Some (*pst* (|*state* := *st*|))) ***in***
    s(|*proc-state*:=*proc-state′*|) ***else*** *s*

The process to be started should belong to the current executing partition and be in *DORMANT* state. As shown in Fig. 6, if the current partition is in *NORMAL* mode, the new state of the process is *READY*. Otherwise, the new state is *WAITING*.

Process Scheduling: ARINC 653 defines a priority based process scheduling in partitions, which is implemented as follows. The process scheduling occurs in a partition only when the partition is in *NORMAL* mode. It first sets the state of the current executing process to *READY* (*setRun2Ready*). Then it chooses one of processes in *READY* state with highest priority in the current partition. Finally, it sets the chosen process as the current executing process of the current partition.

definition *schedule-process* :: *StateR* ⇒ *StateR set* **where**
schedule-process s ≡
  if *is-part conf* (*cur s*) ∧
     (*the* ((*partitions s*) (*cur s*))) = *NORMAL* ***then***
    (***let*** *s′* = *setRun2Ready s*;
      readyprs = {*p*. *p*∈*the* (*procs s′* (*cur s′*)) ∧
            state (*the* (*proc-state s′* (*cur s′*,*p*))) = *READY*};
      selp = *SOME p*.
        p∈{*x*. *state* (*the* (*proc-state s′* (*cur s′*,*x*))) = *READY* ∧
          (∀ *y*∈*readyprs*. *priority* (*the* (*proc-state s′* (*cur s′*,*x*))) ≥
              priority (*the* (*proc-state s′* (*cur s′*,*y*))))};
      st = *the* ((*proc-state s′*) (*cur s′*, *selp*));
      proc-st = *proc-state s′*; *cur-pr* = *cur-proc-part s′* ***in***
    {*s′*(|*proc-state* :=
        proc-st ((*cur s′*, *selp*) := Some (*st*(|*state* := *RUNNING*|))),
      cur-proc-part := *cur-pr*(*cur s′* := Some *selp*)|)})
  else {*s*}

Refined Events: All events at the top level are refined according to the new state type. For instance, the $set\_partition\_mode$ function (Subsection 6.2) at the top level is refined as follows. When setting a partition to *NORMAL* mode, the states of processes in the partition are correctly set (*set_procs_to_normal*) according to process state transitions in Fig. 6. When setting a partition from *NORMAL* mode to others, all processes in the partition are deleted (*remove_partition_resources*).

definition *set-partition-modeR* :: *Sys-Config* ⇒ *StateR* ⇒ *partition-mode-type* ⇒ *StateR* **where**
set-partition-modeR sc s m ≡
  (***if*** (*partitions s*) (*cur s*) ≠ *None* ∧
    ¬ (*the* ((*partitions s*) (*cur s*)) = *COLD-START*
      ∧ *m* = *WARM-START*) ***then***
   let *pts* = *partitions s*;
     s′ = (***if*** *m* = *NORMAL* ***then***
          set-procs-to-normal s (*cur s*)
        else if *the* ((*partitions s*) (*cur s*)) = *NORMAL* ***then***
          remove-partition-resources s (*cur s*)
        else *s*) ***in***
    s′(|*partitions* := *pts*(*cur s′* := Some *m*)|)
  else *s*)

7.3 Security Proofs

Since the second-level specification is a refinement and an instance of the security model, the first part of the security proofs are the instantiation and refinement proofs. The assumptions (1) - (6) of the security model (Definition 2) have been proven on the second-level specification. In order to show the refinement relation, conditions (1) - (6) in Definition 6 are proven on the second-level specification.

The second step of security proofs is to show the UCEs, i.e. satisfaction of Definitions 3 and 4. By following Theorem 5, we only need to prove that SK_C satisfies $SC_\Delta \wedge LR_\Delta$ to show the information-flow security, since the satisfaction of $SC_A \wedge LR_A$ in SK_A has been proven at the top level and we have $SK_A \sqsubseteq_{\Psi,\Theta} SK_C$. Therefore, we define a set of *concrete conditions* for all events on the new state variables. Satisfaction of the conditions of one event implies that the event satisfies the unwinding conditions. For instance, Definition 9 shows the concrete condition of step consistent for the *Schedule_Process* event, which is an instance of UCEs on the new state variables ($SC_{C\Delta}(e)$) of the event.

Definition 9 (Concrete $SC_{C\Delta}(e)$ ***of Schedule_Process).***

$$\begin{aligned}
&\forall\, d\ s\ t\ s'\ t'\ p.\ \mathcal{R}_C(s) \wedge \mathcal{R}_C(t) \wedge s \overset{d}{\sim} t \wedge s \overset{\mathbb{S}}{\sim} t \\
&\quad \wedge\ is_part\ conf\ (cur\ s) \wedge (cur\ s) \rightsquigarrow d \wedge s \overset{cur\ s}{\sim} t \\
&\quad \wedge\ s' \in schedule_process\ s \wedge t' \in schedule_process\ t \\
&\quad \longrightarrow s' \overset{d}{\sim}_\Delta t'
\end{aligned}$$

Finally, we conclude the satisfaction of the *noninfluence* property on the second-level specification and all other information-flow security properties according to Theorem 5 and the inference framework of the security model.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2672983, IEEE Transactions on Dependable and Secure Computing

TABLE 2: Specification and Proof Statistics

Item	Specification		Proof		PM
	# of type/ definition	LOC	# of lemma /theorem	LOP	
Security Model	25	130	63	900	1
Top-level Specification	116	680	193	5,200	6
Refinement	6	100	42	600	1
2nd-level Specification	45	330	111	1,100	4
Total	192	1,240	409	7,800	12

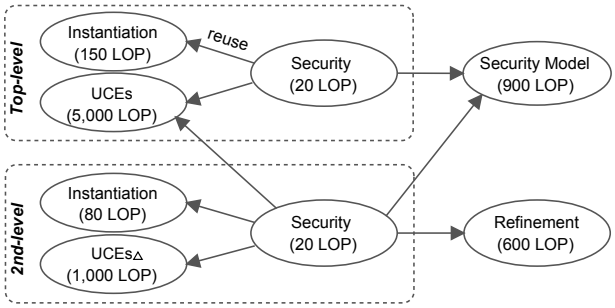

Fig. 7: Proof Reusability

8 Results and Discussion

8.1 Evaluation

We use Isabelle/HOL as the specification and verification system for ARINC SKs. The proofs 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 190 datatypes/definitions and $\sim$ 1,240 lines of code (LOC) of Isabelle/HOL to develop the security model, the refinement framework, and two levels of functional specifications. 409 lemmas/theorems in Isabelle/HOL are proven using $\sim$ 7,800 lines of proof (LOP) of Isar to ensure the information-flow security of the specification. The work is carried out by a total effort of roughly 12 person-months (PM). The proof reusability is shown in Fig. 7. The proof of the security model is reusable at each level. The proof of our refinement is reusable at each refinement step. The major part of proof at each level is UCEs which is reusable at a lower level, for example $\sim$ 5,000 LOP of the top-level specification is reused in the second-level specification.

8.2 Covert Channels and Attack Analysis

When proving the UCEs in our original specification which is completely compliant with ARINC 653, a few covert channels are found in the standard. Then, we conduct a code-to-spec review on VxWorks 653, XtratuM, and POK in accordance with our formal specification. The covert channels are also found in them. Table 3 shows the covert channels and their existence.

Covert channels 1, 2 and 6 actually exist in the ARINC 653 standard. Covert channels 3 and 5 are potential ones and may be introduced to the specification by careless design.

TABLE 3: The Found Covert Channels

Covert Channel	ARINC standard	VxWorks 653	XtratuM	POK
(1)	✓	✓	✓	✓
(2)	✓	✓		
(3)	∗	✓		
(4)		∗	∗	
(5)	∗			
(6)	✓	✓		

✓: existing; ∗: potential; *blank*: not existing

Covert channel 4 does not exist in ARINC 653. However, it is a potential flaw that should be taken into account. They are described as follows. In Appendix A, we present how we find and fix them in our specification in detail.

1) **Queuing mode channel**: If there is a queuing mode channel from a partition a to a partition b, we find a covert channel from b to a in ARINC 653.
2) **Leakage of port identifiers**: In the *Send/Receive_Queuing_Message* and *Write/Read_Sampling_Message* services, there is no judgement on whether the accessing port belongs to the current partition. The port identifier in ARINC 653 is a covert channel.
3) **Shared space of port identifiers**: In ARINC 653, the *Create_Sampling_Port* and *Create_Queuing_Port* services create a port and return a new unique identifier assigned by the kernel to the new port. A potential covert channel is to use a global variable for unused port identifiers.
4) **Partition scheduling**: State related scheduling policies, such as policies only selecting non-*IDLE* partitions, is a covert channel.
5) **Shared space of process identifiers**: In ARINC 653, the *Create_Process* service creates a process and assigns an unique identifier to the created process. A potential covert channel is to use a global variable for unused process identifiers.
6) **Leakage of process identifiers**: In the services of process management, there is no judgement on whether the process belongs to the current partition. Thus, the *locally respects* condition is not preserved on the events. The process identifier in ARINC 653 becomes a covert channel.

Then we review the source code of implementations to validate the covert channels. As we consider single-core separation kernels in this paper, we manually review the single-core version of the implementations. Since the reviewed implementations are non-preemptive during the execution of hypercalls and have the same execution model as in our specification, it makes sense that we review the implementations according to our specification. In EAL 7 evaluation of ARINC SKs, to ensure that the proved security really means that the implementation has the appropriate behavior, a formal model of the low-level design is created. Then, the correspondence between the model and the implementation is shown. Since our specification is at high level, we use the unwinding conditions to manually check the source code of hypercalls in the implementations, rather than to show their correspondence.

The result of code review is shown in Table 3. The version of VxWorks 653 Platform we review is v2.2. The

covert channel 5 is not found during code review. However, the covert channel 4 potentially exists and the other four covert channels are found in VxWorks 653. The version of XtratuM we review is v3.7.3. The covert channel 1 exists and the covert channel 4 is a potential one in XtratuM. The version of POK we review is the latest one released in 2014. We find the covert channel 1 in POK. In Appendix B, we present where the covert channels exist in the source code and how we find them in detail.

The found covert channels pose threats to real-world ARINC SKs. In order to analyze the potential attacks, we assume a threat model of ARINC SKs in which all user-level code in partitions is malicious and acting to break the security policy. The attacker's goal is to read or infer the information in partitions that should remain secret to it according to the policy. Malicious programs in partitions could utilise the covert channels to break the security policy. The security risk of covert channels 1, 2, and 6 is high. Security information in a secret partition can be easily leaked by attackers. The covert channel 4 is a timing channel. Covert channels 3 and 5 have low bandwidth in real-world systems. We illustrate potential attacks to ARINC SKs in Appendix C in detail.

8.3 Discussion

The refinement-based specification and analysis method in the paper is compliant to the EAL 7 of CC certification. With regard to the EAL 7, the main requirements addressed by formal methods are (1) a formal security policy model (SPM) of Target of Evaluation (TOE), (2) a complete semi-formal functional specification (FSP) with an additional formal specification, and (3) a complete semi-formal and modular design with high-level TOE design specification (TDS). The security model in this paper represents the security policies of ARINC SKs and is a formal model of the SPM. The two levels of functional specifications in this paper correspond to the FSP. The properties demonstrated on the SPM are formally preserved down to the FSP by the model instantiation and instantiation proofs. The functional specification can be refined to a TDS model using the step-wise refinement. The refinement provides the formal link between the FSP and the TDS. Finally, code-to-spec review can be considered between the last formal model of the TDS and the implementation to show the correspondence. In this paper, we conduct a code review of the implementations according to our specification.

The method in this paper can alleviate the efforts of CC certification. The instantiation of the security model and the refinement framework ease the development and make the proof easier. As the same in the high assurance levels of CC certification of INTEGRITY-178B and AAMP7G [51], formal model and proof are part of the evaluation and directly submitted to the evaluation team for certification. Certainly, formal model and proof should be created in accordance with a set of "safe" rules, such as rules of Isabelle/HOL in [48], which have been complied with by our specification. On the other hand, for a specific TOE in CC certification, our specification may be revised and TDS model has to be developed.

9 Conclusions and Future Work

In this paper, we have presented a refinement-based specification development and security analysis method for ARINC SKs. By the proposed superposition refinement, we have developed two levels of formal specification. We provided the mechanically checked proofs of information-flow security to overcome covert channels in separation kernels. We revealed some security flaws in ARINC 653 and implementations. In the next step, we will refine the second-level specification to lower levels by adding services of complicated process management and intra-partition communication. Supporting multi-core is also under consideration in future. Due to the kernel concurrency between cores, we are developing rely-guarantee based approach to specify and verify multi-core separation kernels.

Acknowledgement

We would like to thank David Basin of Department of Computer Science, ETH Zurich, Gerwin Klein and Ralf Huuck of NICTA, Australia for their suggestions.

References

[1] G. Parr and R. Edwards, "Integrated Modular Avionics," *Air & Space Europe*, vol. 1, no. 2, pp. 72 – 75, March - April 1999.
[2] *AUTOSAR Specifications (Release 4.2)*, AUTOSAR GbR, July 2015.
[3] J. Rushby, "Design and Verification of Secure Systems," *ACM SIGOPS Operating Systems Review*, vol. 15, no. 5, pp. 12–21, December 1981.
[4] J. Rushby, "Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance," U.S. Department of Transportation, Federal Aviation Administration, DOT/FAA/AR-99/58, 2000.
[5] *ARINC Specification 653: Avionics Application Software Standard Interface, Part 1 - Required Services*, Aeronautical Radio, Inc., November 2010.
[6] *Common Criteria for Information Technology Security Evaluation*, 3rd ed., National Security Agency, 2012.
[7] *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, National Security Agency, 2007.
[8] I. Craig, *Formal Refinement for Operating System Kernels*. New York, USA: Springer-Verlag, 2007, ch. 5.
[9] A. Velykis and L. Freitas, "Formal Modelling of Separation Kernel Components," in *Theoretical Aspects of Computing*, LNCS, Springer, 2010, vol. 6255, pp. 230–244.
[10] F. Verbeek, S. Tverdyshev, et al., "Formal Specification of a Generic Separation Kernel," *Archive of Formal Proofs*, 2014.
[11] F. Verbeek, O. Havle, J. Schmaltz, et al., "Formal API Specification of the PikeOS Separation Kernel," in *NASA Formal Methods*, LNCS, Springer, 2015, vol. 9058, pp. 375–389.
[12] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean, "Applying Formal Methods to a Certifiably Secure Software System," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 82–98, January - February 2008.
[13] R. Richards, *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer US, 2010, ch. Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel, pp. 301–322.
[14] M. Wilding, D. Greve, R. Richards, and D. Hardin, *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer US, 2010, ch. Formal Verification of Partition Management for the AAMP7g Microprocessor, pp. 175–191.
[15] L. Freitas and J. McDermott, "Formal Methods for Security in the Xenon Hypervisor," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 5, pp. 463–489, Octorber 2011.
[16] T. Murray, D. Matichuk, M. Brassil, et al., "seL4: From General Purpose to a Proof of Information Flow Enforcement," in *Proc. of S&P 13*, IEEE Press, CA, USA, May 2013, pp. 415 – 429.
[17] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, "Formal Verification of Information Flow Security for a Simple Arm-based Separation Kernel," in *Proc. of CCS 13*, ACM Press, Berlin, Germany, November 2013, pp. 223–234.

[18] D. Sanán, A. Butterfield, and M. Hinchey, "Separation Kernel Verification: The Xtratum Case Study," in *Verified Software: Theories, Tools and Experiments*, LNCS, Springer, 2014, vol. 8471, pp. 133–149.
[19] M. Clarkson and F. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.
[20] J. Rushby, "Noninterference, Transitivity, and Channel-control Security Policies," SRI International, Computer Science Laboratory, 1992.
[21] A. Sabelfeld and A. Myers, "Language-based Information-flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, January 2003.
[22] D. Oheimb, "Information Flow Control Revisited: Noninfluence = Noninterference+ Nonleakage," in *Computer Security*, LNCS, Springer, 2004, vol. 3193, pp. 225–243.
[23] T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein, "Noninterference for Operating System Kernels," in *Certified Programs and Proofs*, LNCS, Springer, 2012, vol. 7679 , pp. 126–142.
[24] R. Back and K. Sere, "Stepwise Refinement of Action Systems," in *Mathematics of Program Construction*, LNCS, Springer, 1989, vol. 375, pp. 115–138.
[25] Niklaus Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, April 1971.
[26] R. Back,, and K. Sere, "Superposition Refinement of Reactive Systems," *Formal Aspects of Computing*, vol. 8, no. 3, pp. 324–346, May 1996.
[27] J. Abrial and S. Hallerstede, "Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp.1–28, January 2007.
[28] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive Formal Verification of an OS Microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp.2:1 – 2:70, February 2014.
[29] Y. Zhao, Z. Yang, D. Sanan, and Y. Liu, "Event-based Formalization of Safety-Critical Operating System Standards: An Experience Report on ARINC 653 Using Event-B," in *Proc. of ISSRE 15*, IEEE Press, MD, USA, November 2015, pp. 281 – 292.
[30] A. Gomes, "Formal Specification of the ARINC 653 Architecture Using Circus," Master's thesis, University of York, 2012.
[31] J. Delange, L. Pautet, and F. Kordon, "Modeling and Validation of ARINC653 architectures," in *Proc. of ERTS 10*, Toulouse, France, May 2010.
[32] P. Cámara, J. Castro, M. Gallardo, and P. Merino, "Verification Support for ARINC-653-based Avionics Software," *Software Testing, Verification and Reliability*, vol. 21, no. 4, pp. 267–298, December 2011.
[33] T. Nipkow, M. Wenzel, and L. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.
[34] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach," in *Proc. of EDCC 10*, IEEE Press, Valencia, Italy, April 2010, pp.67–72.
[35] J. Delange, and L. Lec, "POK, an ARINC 653-compliant Operating System Released under the BSD License," in *Proc. of 13th Real-Time Linux Workshop*, Prague, Poland, October 2011, pp.1–13.
[36] J. Millen, "20 years of Covert Channel Modeling and Analysis," in *Proc. of S&P 99*, IEEE Press, Oakland, USA, May 1999, pp. 113–114.
[37] Y. Zhao, D. Sanan, F. Zhang and Y. Liu, "Reasoning About Information Flow Security of Separation Kernels with Channel-based Communication," in *Proc. of TACAS 16*, Springer, Eindhoven, The Netherlands, April 2016, pp. 791 – 810.
[38] J. Goguen and J. Meseguer, "Security Policies and Security Models," in *Proc. of S&P 82*, IEEE Press, Oakland, CA, USA, April 1982, pp. 11 – 20.
[39] H. Mantel, D. Sands, "Controlled Declassification Based on Intransitive Noninterference," in *Proc. of APLAS 04*, Spinger, Taipei, Taiwan, November 2004, pp. 129–145.
[40] M. Krohn, E. Tromer, "Noninterference for a Practical DIFC-Based Operating System," in *Proc. of S&P 09*, IEEE Press, Berkeley, CA, May 2009, pp. 61–76.
[41] A.G. Ramirez, J. Schmaltz, F. Verbeek, et al., "On Two Models of Noninterference: Rushby and Greve, Wilding, and Vanfleet," in *Proc. of SAFECOMP 14*, Springer, Florence, Italy, September 2014, pp. 246–261.
[42] T. Levin, C. Irvine, C. Weissman, and T. Nguyen, "Analysis of Three Multilevel Security Architectures," in *Proc. of CSAW 07*, ACM Press, VA, USA, October 2007, pp. 37–46.
[43] M. Clarkson, B. Finkbeiner, M. Koleini, K. Micinski, M. Rabe, and C. Sánchez, "Temporal Logics for Hyperproperties," in *Principles of Security and Trust*, LNCS, Springer, 2014, vol. 8414, pp. 265–284.
[44] S. Eggert, R. Van der Meyden, H. Schnoor, and T. Wilke, "The Complexity of Intransitive Noninterference," in *Proc. of S&P 11*, IEEE Press, California, USA, May 2011, pp. 196–211.
[45] V. Ha, M. Rangarajan, D. Cofer, H. Rues, and B. Dutertre, "Feature-Based Decomposition of Inductive Proofs Applied to Real-Time Avionics Software: An Experience Report," in *Proc. of ICSE 04*, IEEE Press, Scotland, UK, May 2004, pp. 304–313.
[46] G. Klein, "Operating System Verification - an Overview," *Sadhana*, vol. 34, no. 1, pp. 27–69, February 2009.
[47] J. Andronick, R. Jeffery, G. Klein, et al., "Large-scale Formal Verification in Practice: A Process Perspective," In *Proc. of ICSE 12*, IEEE Press, Zurich, June 2012, pp. 1002–1011.
[48] H. Blasum, O. Havle, S. Tverdyshev, B. Langenstein, W. Stephan, et al., "Used Formal Methods," White Paper, EURO-MILS, 2015.
[49] T.S. Hoang, "Security Invariants in Discrete Transition Systems," *Formal Aspects of Computing*, vol. 25, no. 1, pp. 59–87, January 2013.
[50] D. McCullough, "Noninterference and the Composability of Security Properties," in *Proc. of S&P 88*, IEEE Press, Oakland, USA, April 1988, pp. 177–186.
[51] R. Richards, D. Greve, M. Wilding, "The Common Criteria, Formal Methods and ACL2," in *Proc. of ACL2 Workshop 04*, Austin, Texas, USA, November 2004.

Yongwang Zhao received the Ph.D. degree in computer science from Beihang University (BUAA) in Beijing, China, in 2009. He is an associate professor at the School of Computer Science and Engineering, Beihang Univerisity. He has also been a Research Fellow in the School of Computer Science and Engineering, Nanyang Technological University, Singapore, from 2015. His research interests include formal methods, OS kernels, information-flow security, and AADL.

David Sanán received the Ph.D. degree in Software Engineering and Artificial Inteligent from the University of Málaga, Málaga, Spain, in 2009. He has been working as a Research Fellow in the Singapore University of Technology and Design (SUTD), Trinity College Dublin (TCD), and National University of Singapore (NUS). In 2015 he joined Nanyang Technological University in Singapore, where he is currently working as a research fellow. His interest research includes formal methods, and in particular the formalization and verification of separation micro-kernels aiming multi-core architectures.

Fuyuan Zhang received the Ph.D. degree in computer science from Technical University of Denmark in 2012. Currently, he is a research fellow in School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore. His research interests include formal verification of IT systems, computer security and quantum computation.

Yang Liu received the Ph.D. degree in computer science from National University of Singapore (NUS) in 2010 and continued with his post-doctoral work in NUS. Since 2012, he joined Nanyang Technological University as an Assistant Professor. His research focuses on software engineering, formal methods and security. Particularly, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the art model checker, Process Analysis Toolkit.

Refinement-based Specification and Security Analysis of Separation Kernels

Yongwang Zhao, David Sanán, Fuyuan Zhang, Yang Liu

APPENDIX A
COVERT CHANNELS IN ARINC 653

The covert channels in ARINC 653 are discussed and fixed in our specification as follows.

Covert Channel 1 - Queuing mode channel: If there is a queuing mode channel from a partition a to a partition b and no other channels exist, then it is secure that $a \rightsquigarrow \mathbb{T}$, $\mathbb{T} \rightsquigarrow b$, $\mathbb{T} \not\rightsquigarrow a$ and $b \not\rightsquigarrow \mathbb{T}$ according to the security policies in this paper. Actually, these security policies are violated in ARINC 653. First, when a sends a message by invoking the *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 *locally respects* condition is not preserved on the event of message transmission, and $\mathbb{T} \not\rightsquigarrow a$ is violated. Second, due to no message loss allowed in 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 the *Receive_Queuing_Message* service executed by the partition b. Thus, the *locally respects* condition is not preserved on *Receive_Queuing_Message*, and $b \not\rightsquigarrow \mathbb{T}$ is violated. To avoid this covert channel, we allow message loss on queuing mode channels in our specification.

Covert Channel 2 - Leakage of port identifiers: It is assumed in ARINC 653 that a port identifier is only stored in a partition after creation. In the *Send/Receive_Queuing_Message* and *Write/Read_Sampling_Message* services, there is no judgement on whether the accessing port belongs to the current partition. Thus, the *locally respects* condition is not preserved on the events. In this case, programs in a partition can guess the port identifiers of other partitions and then manipulate the ports. Therefore, the port identifier in ARINC 653 is a covert channel. This covert channel is avoided in our specification by checking that the port belongs to the current partition.

Covert Channel 3 - Shared space of port identifiers: In ARINC 653, the *Create_Sampling_Port* and *Create_Queuing_Port* services create a port and return a new unique identifier assigned by the kernel to the new port. Careless design of the port identifier can cause covert channels. In the initial specification, we use a natural number to maintain this new identifier. This number is initially assigned to "1" and increased by one after each port creation. In such as design, the two events do not preserve the *step consistent* condition. Thus, the number becomes a covert channel that can flow information from any partition to others. This covert channel is then avoided in our specification by assigning the port identifier to each port during system initialization or in the system configuration.

Covert Channel 4 - Partition scheduling: ARINC 653 defines a cyclic partition scheduling. The time windows of a partition in the *IDLE* mode are not preempted by other partitions. The result of executing the *Schedule* event is that the current executing partition is set to a partition or the message transmitter. Although some partition is in *IDLE* mode, it is also possible to be selected. That means the execution of *Schedule* is independent with the state of partitions. However, state related scheduling policies, such as policies only selecting non-*IDLE* partitions, do not have the information-flow security. In such a case, the *step consistent* condition is not preserved on the *Schedule* event, and the *Set_Partition_Mode* service in partitions can interfere with the scheduler. This covert channel can be avoided by disabling the insecure partition scheduling.

Covert Channel 5 - Shared space of process identifiers: In ARINC 653, the *Create_Process* service creates a process and assigns an unique identifier to the created process. Careless design of the process identifier can cause covert channels. In the initial specification, we use global identifiers. When creating a process, we assign a new identifier which is not used by created processes. In such as design, the *Create_Process* service does not preserve the *step consistent* condition. Thus, the global identifiers become covert channels that can flow information from any partition to others. This covert channel is then avoided in our specification by assigning a new identifier which is not used by created processes in current executing partition. Unlike communication ports, processes in ARINC 653 are not configured at built-

- *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.*
- *Y. Zhao is with the School of Computer Science and Engineering, Beihang Univerisity, Beijing, China (Email: zhaoyw@buaa.edu.cn) and the School of Computer Science and Engineering, Nanyang Technological University, Singapore (Email: ywzhao@ntu.edu.sg).*
- *D. Sanán, F. Zhang and Y. Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore (Email: {sanan, fuzh, yangliu}@ntu.edu.sg).*

time.

Covert Channel 6 - Leakage of process identifiers: It is assumed in ARINC 653 that a process identifier is only stored in a partition after creation. In process management services, e.g. *Start/Stop/Suspend/Resume_Process*, there is no judgement on whether the process belongs to the current partition. Thus, the *locally respects* condition is not preserved on the events. In such a case, programs in a partition can guess the process identifiers of other partitions and then manipulate them. Therefore, created processes in ARINC 653 become covert channels. This covert channel is avoided in our specification by checking that the process belongs to the current partition.

APPENDIX B
COVERT CHANNELS IN IMPLEMENTATIONS

We find five covert channels in VxWorks shown as follows. The covert channel 5 is not found during code review.

Covert channel 1: In VxWorks 653, the message transmission is not implemented as a system event, but invoked by the *Send_Queuing_Message* service. If there is a queuing mode channel c from a partition a to a partition b and no other channels exist, then it is secure that $a \rightsquigarrow b$ and $b \not\rightsquigarrow a$. VxWorks uses the *portQMsgPut* and *portQMsgGet* functions to implement the *Send_Queuing_Message* and *Receive_Queuing_Message* services, respectively. According to the source code of *portQMsgPut*, we find that when partition a sends messages and the source port of c is full, it invokes the *portQMsgDistribute* function to transmit messages in the source port to the destination port. But when the destination port is also full, the *portQMsgPut* function returns an error. However, the full status of the destination port can be changed by the *portQMsgGet* function executed in partition b. Thus, the *locally respects* condition is not preserved on *portQMsgGet*, and covert channel 1 exists in VxWorks 653.

Covert channel 2: VxWorks 653 uses the *portSMsgPut* and *portSMsgGet* functions to implement the *Write_Sampling_Message* and *Read_Sampling_Message* services, respectively. According to the source code of the *portQMsgPut*, *portQMsgGet*, *portSMsgPut*, and *portSMsgGet* functions, we find that when accessing a port, VxWorks 653 does not check that the port belongs to the current partition. Thus, these functions executed in a partition can manipulate ports in another noninterfering partition. The *locally respects* condition is not preserved on these functions and covert channel 2 exists in VxWorks 653.

Covert channel 3: VxWorks 653 uses the *portQCreate* and *portSCreate* functions to implement the *Create_Queuing_Port* and *Create_Sampling_Port* services, respectively. According to the source code of *portQCreate* and *portSCreate*, VxWorks 653 uses an object classifier to allocate port identifiers by invoking the *objAlloc* function. When invoking *objAlloc*, it uses the same object class (*portQClassId*) for all queuing ports. Thus, the *step consistent* condition is not preserved on the two functions and the *portQClassId* becomes a covert channel, i.e. covert channel 3.

Covert channel 4: Beside the ARINC 653 partition scheduling, in which the partition scheduler is not interfered by partitions, VxWorks also supports Priority Preemptive Scheduling (PPS) for partitions. The PPS runs ARINC partitions in a priority pre-emptive manner during idle time in an ARINC schedule. The PPS is implemented in the *ppsSchedulePartition* function, which choose the partition with the highest priority and ready tasks. In such a scheduling, the *step consistent* condition is not preserved on the *ppsSchedulePartition* function, and the *Set_Partition_Mode* service and the services of process management in partitions can interfere with the scheduler. Thus, covert channel 4 potentially exists.

Covert channel 6: VxWorks 653 uses the *taskActivate*, *taskStop*, *taskSuspend*, and *taskResume* functions to implement the *Start_Process*, *Stop_Process*, *Suspend_Process*, and *Resume_Process* services, respectively. In these functions of VxWorks 653, it does not check that the accessing task belongs to the current partition. Thus, the *locally respects* condition is not preserved on the functions. In such a case, programs in a partition may manipulate tasks in other partitions that the partition can not interfere with. Therefore, created tasks in VxWorks 653 become covert channels, i.e. covert channel 6.

We find two covert channels in XtratuM as follows. During code review of unwinding conditions, other covert channels are not found.

Covert channel 1: XtratuM uses one shared buffer between the source port and the destination port of a queuing mode channel as a transmitter. If there is a queuing mode channel c from a partition a to a partition b and no other channels exist, then it is secure that $a \rightsquigarrow b$ and $b \not\rightsquigarrow a$. XtratuM uses the *SendQueuingPort* and *ReceiveQueuingPort* hypercalls to implement the *Send_Queuing_Message* and *Receive_Queuing_Message* services, respectively. According to the source code of *SendQueuingPort*, we find 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*. However, the full status of the buffer can be changed by the *ReceiveQueuingPort* hypercall executed in partition b. Thus, the *locally respects* condition is not preserved on *ReceiveQueuingPort*, and covert channel 1 exists in XtratuM.

Covert channel 4: Beside the ARINC 653 partition scheduling, in which the partition scheduler is not interfered by partitions, XtratuM also supports fixed priority partition scheduling (FPS). The FPS chooses the *READY* partition with the highest priority to be executed. The *Schedule* function in XtratuM could be configured at built-time as FPS or ARINC 653 scheduling. In the FPS, the *step consistent* condition is not preserved on the *Schedule* function, and the *Set_Partition_Mode* service in partitions can interfere with the scheduler. Thus, covert channel 4 exists in XtratuM.

We find one covert channel in POK as follows. During code review of unwinding conditions, other covert channels are not found.

Covert channel 1: 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 there is a queuing mode channel from a partition a to a partition b and no other channels exist, then it is secure that $a \rightsquigarrow \mathbb{T}$, $\mathbb{T} \rightsquigarrow b$, $\mathbb{T} \not\rightsquigarrow a$ and $b \not\rightsquigarrow \mathbb{T}$. POK uses the *pok_port_queueing_send* and *pok_port_queueing_receive* syscalls to implement the *Send_Queuing_Message* and *Receive_Queuing_*

Message services, respectively. First, according to the source code of *pok_port_queueing_send*, we find if the buffer of the source port is not full, *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*. However, the full status of the source port can be changed by the *pok_port_transfer* function which is in charge of transmitting messages from a source port to a destination one. Thus, the *locally respects* condition is not preserved on *pok_port_transfer*, and $\mathbb{T} \not\rightarrowtail a$ is violated. Second, *pok_port_transfer* returns *POK_ERRNO_SIZE* when the destination port has no available space to store messages. However, the full status of the destination port can be changed by the *pok_port_queueing_receive* function executed by partition b. Thus, the *locally respects* condition is not preserved on *pok_port_queueing_receive*, and $b \not\rightarrowtail \mathbb{T}$ is violated.

APPENDIX C
ATTACK ANALYSIS OF COVERT CHANNELS

We illustrate potential attacks to ARINC SKs as follows.

Covert channel 1: It is a typical storage channel between a sender and a receiver [1]. If there is a queuing mode channel from a partition a to a partition b, malicious programs in a and b can collaborate to create a covert channel from b to a. Partition a sends messages to the channel until the buffer is full. Then, the *receive* event in b can be inferred by a, and thus a gets one bit of information from b at each time of *receive*. By the covert channel, any secret information in b can be sent to a.

Covert channel 2: It is a storage channel about resource isolation and leakage in ARINC SKs. Let's assume a system with three partitions (a, b, and c) and a communication channel ch from b to c. According to security policies, information in a cannot be leaked to other partitions. However, by the covert channel, malicious programs in partition a can manipulate the channel ch and send secret information of a to c.

Covert channels 3 and 5: They are storage channels on the shared identifiers among partitions. Malicious programs in partitions have to create resources (ports and processes) to reach the maximum number of the identifiers, and then can send one bit of information. Thus, the covert channels have very low bandwidth in real-world systems.

Covert channel 4: It is a timing channel by the PPS in VxWorks 653 and the FPS in XtratuM. Higher prioritised partitions can directly influence when and for how long lower prioritised partitions run [2].

Covert channel 6: It is a storage channel about resource isolation and leakage in ARINC SKs. Assume a system with a set of partitions and there is no communication channel between partitions a and b. According to security policies, information in a and b has to be isolated. However, by the covert channel, malicious programs in a can manipulate the processes (e.g. change the process state) in b, and thus send some bits of information to b at each time.

REFERENCES

[1] *A Guide to Understanding Covert Channel Analysis of Trusted Systems*, National Computer Security Center, USA, NCSC-TG-030, November 1993.

[2] M. Völp, C.J. Hamann, H. Härtig, "Avoiding Timing Channels in Fixed-priority Schedulers," in *Proc. of ASIACCS 08*, ACM Press, Tokyo, Japan, March 2008, pp. 44–55.