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.


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 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 paper presents a specification development and security analysis method based on stepwise refinement for ARINC SKs. The challenges mentioned above are not well addressed in the literature. In formal verification of the sel4 microkernel and its separation extension, 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 SKs are not the emphasis of these works and the formal specification is not compliant with ARINC 653. Information-flow security verification has been enforced on PikeOS, INTEGRITY-178B, ARM-based separation kernel, and an ARM-based separation kernel. However, refinement is not considered in these works. Correctness-by-construction methods have been used to create formal specification of separation kernels. 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, modeling ARINC 653 for model-driven development of IMA applications, and formal verification of application software on top of ARINC 653. In the process of refining the ARINC 653 specification, 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 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://security.sce.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. 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, and POK, in accordance with the formal specification. During the verification and code review, six covert channels to leak information 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 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 order to provide a formal foundation for the specification and analysis of information-flow security policies. The idea is that a
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 [40], [41]. 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 considers 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-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], [42] 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) [43], which is intransitive. It is expressed abstractly in a partition flow matrix partition flow : partition × partition → mode, whose entries indicate the mode of the flow. For instance, 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 [34], an implementation of high-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 ("[]") | Cons 'a 'a list "(infixr "#")", where [] is the empty list and # is the concatenation. The polymorphic option type is defined as datatype 'a option = None | Some (the : 'a). A function of type 'b → 'a models a partial function, which is equal to 'b ⇒ ('a option). Record types may be defined, for example record point = xcoord :: int 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.

### 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 ([44], [45]), 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 [46]. 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

![Fig. 1: Architecture of ARINC 653 Separation Kernels](image-url)
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 IFPP 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). \( M = (S, E, \varphi, s_0) \) is a tuple, where \( S \) is the state space, \( E \) is the set of event labels, \( s_0 \in S \) is the initial state, and \( \varphi : E \to \mathcal{P}(S \times S) \) is the state-transition function.

The \( \varphi \) function characterises the single-step behaviour of separation kernels by executing an event, such as a hypercall or scheduling. The auxiliary functions used in
the security model are defined in detail in Fig. 3. Based
on the \(\varphi\) function, we define the \(\text{run}\) function to represent
the execution of a sequence of events. The \(\text{execution}(s, es)\)
function (denoted as \(s \rightarrow es\)) returns the set of final states
by executing a sequence of events \(es\) from a state \(s\), where \(\rightarrow\)
is the domain restriction of a relation. By the \(\text{execution}\) function,
the reachability of a state \(s\) is defined as \(\text{reachable}(s)\)
(denoted as \(R(s)\)).

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 determinis-
tic over time, we define a security domain \(\text{sche}-\)
dule for partition scheduling, which cannot be interfered by any
other domains to ensure that the \(\text{sche}-\)
dule does not leak information via its scheduling decisions. Note that process scheduling is conducted in partitions. Since ARINC 653 de-
defines the channel-based communication services using ports
and leaves the implementation of message transmission
over channels to underlying separation kernels, a security
domain \(\text{message transimission}\) is defined for message transmis-
sion. 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 \(S\), the transmitter \(T\), and
the configured partitions \(P\), i.e. \(\mathcal{D} = P \cup \{S, T\}\).

**Security policies:** In order to discuss information-flow
security policies, we assume an \(\text{interference relation} \sim \subseteq \mathcal{D} \times \mathcal{D}\) according to the \(\text{partition flow}\) matrix and \(\not\sim\) is the
complement relation of \(\sim\). If there is a channel from
a partition \(p_1\) to a partition \(p_2\), then \(p_1 \sim p_2\) and \(T \sim p_2\)
since the transmitter is the message intermediate. 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 deter-
mined solely from the event itself [20]. However, in separa-
tion kernels that mapping is dynamic [23]. When a \(\text{hypercall}\)
occcurs, 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 \(kdom : S \times E \rightarrow D\).

**State equivalence:** It means states are identical for a
security domain. We define an equivalence relation \(\sim\) on
states for each domain such that \(s \sim t\) 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 \(s \sim t\) 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
\(ss \sim ts\) 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).**

\[
\mathcal{SM} = (\mathcal{M}, \mathcal{D}, kdom, \sim, \sim)
\]

with assumptions as follows.

1) \(\forall p_1, p_2 \in P, p_1 \sim p_2 \rightarrow p_1 \sim T \land T \sim p_2\)
2) \(\forall d \in D, S \sim d\)
3) \(\forall d \in D, d \sim d = S\)
4) \(\sim\) is an equivalence relation.
5) \(\forall s t e, s \sim t \rightarrow kdom(s, e) = kdom(t, e)\)
6) Events are enabled in any state, i.e. \(\forall s e. \ R(s) \rightarrow
(\exists s'. (s, s') \in \varphi(e))\).

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 \(\text{sources}(es, s, d)\) 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 \(\text{sources}(e\#es, s, d)\) 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 inter-
mediate domain \(v\) by which the domain \(kdom(s, e)\) can
pass information to \(d\) indirectly. In the intransitive purged
sequence \(\text{ipurge}(es, d, ss)\) 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 \(ss\), \(\text{ipurge}\) keeps the first event
\(e\) if this event is allowed to affect the target domain \(d\), i.e.
\(\exists s \in ss. \ kdom(s, e) \in \text{sources}(e\#es, s, d)\). 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 \(es\).
from the current set of states ss. The observational equivalence of an execution is denoted as \((s \Rightarrow es) \approx d (t \Rightarrow es)\), which means that a domain \(d\) is identical to any two final states after executing \(es_1\) from \(s (s \Rightarrow es_1)\) 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. \(\text{ipurge}(es_1, d, \{s_0\}) = \text{ipurge}(es_2, d, \{s_0\})\). 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 noninterference\(_R\) based on the reachable function \(R\). 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 \(R(s_0) = \text{True}\). 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 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. \(s \approx es\) \(\approx t\), and (2) the same domain is currently executing in both states, i.e. \(s \approx t\), 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 secret data leakage and secret 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.

![Inference Framework of Security Properties](image)

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).**

\[
SC(e) \triangleq \forall d s t. R(s) \land R(t) \land (s \not\approx t) \land (s \not\approx t) \\
\land (kdom(s, e) \not\approx d) \land (s \not\approx t) \\
\rightarrow (\forall s'. (s', s') \in \varphi(e) \land (t, t') \in \varphi(e) \rightarrow s' \not\approx t')
\]

**Definition 4 (Locally Respects Condition of Event).**

\[
LR(e) \triangleq \forall d s t. R(s) \land (kdom(s, e) \not\approx d) \land (s, s') \in \varphi(e) \land (s, s') \in \varphi(e) \\
\rightarrow (s \not\approx s')
\]

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. $s' \not\sim \triangleleft t'$, 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 interfere 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 $SC \equiv \forall e. SC(e)$ and $LR \equiv \forall e. LR(e)$. 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 $SM$ satisfies that

$$ \text{(SC \land LR) = noninfluence and SC = nonleakage} $$

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 $SM$ satisfies that

\[
\begin{align*}
\text{noninfluence} &= (\text{noninterference}_r \land \text{nonleakage}) \quad \text{and} \\
\text{weak_noninfluence} &= (\text{weak_noninterference}_r \\
&\quad \land \text{nonleakage})
\end{align*}
\]

### 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 $SM$ locale, i.e. $SM_I = \text{interpretation} SMI(s_0/s_0', \varphi/\varphi_1, kdom/kdom_1, S/Sched, T/Trans, \sim_1/\sim_1, \sim_1', \sim_1)$, where the parameters of $SM$ 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 $SM$.

The event specification is a mapping from events to a set of functions to model the functionalities and services in ARINC 653. The $\varphi_1$ 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 $SK = (SM_I, F)$, where

1) $SM_I = (M_I, D_I, kdom_1, \sim_1, \sim_1')$ is the execution model, where $M_I = (S_I, E_I, \varphi_1, s_0_I)$.
2) $F : E_I \rightarrow \Gamma$ is the event specification, where $\Gamma$ is the set of functions and each $f \in \Gamma$ has the type $S_I \rightarrow P(S_I)$.
3) $\varphi_1$ in $M_I$ defines the execution of each event e, where $\varphi_1(e) = \{(s, t) \mid t \in F(e)(s)\}$.

#### 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 $S_A$ can be transformed into $S_T$ and new state variables denoted as $S_\Delta$ can be incorporated. Thus, the concrete state $S_C = S_T + S_\Delta$, where the symbol “+” 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 $\Psi : SC \rightarrow SA$ and an event relation $\Theta : EC \rightarrow (EA \cup \{\tau\})$, denoted as $SK_A \subseteq \Psi, \Theta SK_C$, if

1) the initial state in $SK_C$ establishes $R_i$, i.e. $\Psi(s_0C) = s_0A$,
2) each event in $EA$ is refined by a set of events in $EC$, i.e. $\Theta$ is a surjection and $\forall s t. \Theta(e) \neq \tau \land (s, t) \in \varphi_C(e) \rightarrow (\Psi(s), \Psi(t)) \in \varphi_A(\Theta(e))$,
3) new state variables only change the new state variables introduced in the refinement and does not affect the variables in $SK_A$, i.e. $\forall e s t. \Theta(e) = \tau \land (s, t) \in \varphi_C(e) \rightarrow \Psi(t) = \Psi(s)$,
4) security domains are preserved and refined events have the same execution domain as that at the abstract level, i.e. $D_C = D_A$ and $\forall s t. \Theta(e) = \tau \rightarrow kdom_C(s, e) = kdom_A(\Psi(s), \Theta(e))$.
5) the refinement does not change the interference relation, i.e. $\sim_C = \sim_A$,
6) $s \sim^d_C t = \Psi(s) \sim^d_A \Psi(t)$ and $s \sim^d \sim^\Delta t$, where $\sim^\Delta$ only considers the new state variables $S_\Delta$ introduced in the refinement.

The relation $\Psi$ 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 s at the concrete level is a subset of next states by executing the refined event $\Theta(e)$ in a state $s = \Psi(t)$ 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 $\tau$ action, which does not change the abstract states. The security configuration at the abstract level is preserved in the refinement, i.e. (1) $S_i, T_i$, partitions $D_i$ and the relation $\sim$ 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 $SM$ and $SK_A \subseteq \Psi, \Theta SK_C$, then

$$ \forall e s. \Psi^*(s_{0C} \Rightarrow es) \subseteq s_0A \Rightarrow \Theta^*(es) $$
In the theorem, $\Psi^*$ maps a set of concrete states to a set of abstract states by the relation $\Psi$, and $\Theta^*$ maps a sequence of concrete events to a sequence of abstract events by the relation $\Theta$. The $\tau$ 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 $\Psi$ relation is also reachable at the abstract level.

**Lemma 1 (State Reachability in Refinement).** If $SK_A$ and $SK_C$ are instances of $SM$ and $SK_A \subseteq_{\psi, \theta} SK_C$, then

$$\forall s. R_C(s) \rightarrow R_A(\Psi(s))$$

Superposition refinement [26] assumes the termination of new events introduced in the refinement. Since hypercalls and system events of ARINC SKs terminate, we 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 $SM$ and $SK_C$ satisfies $SC_C \land LR_C$, 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).**

$$SC_{C\Delta}(e) \triangleq \forall d s t. R_C(s) \land R_C(t) \land (s \sim^d_C t) \land (s \sim^S_C t)$$

$$\land (\text{kdmc}(s,e), \sim^C_C) \land (s, \text{kdmc}(s,e)) \land$$

$$\rightarrow (\forall t'. (s', e') \in \varphi_C(e) \land (t,t') \in \varphi_C(e) \rightarrow s' \sim^d_C t')$$

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 $SM$ and $SK_A \subseteq_{\psi, \theta} SK_C$, then

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

**Lemma 3 (Locally Respects of Event Refinement).** If $SK_A$ and $SK_C$ are instances of $SM$ and $SK_A \subseteq_{\psi, \theta} SK_C$, then

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

**Lemma 4 (Step Consistent of New Event in Refinement).** If $SK_A$ and $SK_C$ are instances of $SM$ and $SK_A \subseteq_{\psi, \theta} SK_C$, then

$$\forall e. \Theta(e) = \tau \land SC_{C\Delta}(e) \rightarrow SC_C(e)$$

**Lemma 5 (Locally Respects of New Event in Refinement).** If $SK_A$ and $SK_C$ are instances of $SM$ and $SK_A \subseteq_{\psi, \theta} SK_C$, then

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 $SM$, $SK_A$ satisfies $SC_A \land LR_A$, $SK_C$ satisfies $SC_{\Delta} \land LR_{\Delta}$, and $SK_A \subseteq_{\psi, \theta} SK_C$, 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 $SM$, $SK_A$ satisfies $SC_A \land LR_A$, $SK_A \subseteq_{\psi, \theta} SK_C$, and

$$\forall e. \Theta(e) = \tau,$$

and $\forall s t d. s \sim^d t$, 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 $\text{Sys\_Config}$” and defined as a constant “$\text{conf\_Sys\_Config}$” 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 ($\text{partitions}$) and the created ports ($\text{part_
The state transition function $\varphi$ 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 $\mathcal{S}$ and the domain of message transmission is $\mathcal{T}$. The domain of hypercalls is dynamic and depends on the current state of the kernel. It is defined as $kdom s (\text{hyperc} h) = cur s$, 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 $\text{interference1}$ as follows in the top-level specification.

**State Equivalence**: For a partition $d$, $s \sim d t$ if and only if $vpeq_part s d t$, which is defined as follows.

**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 transitioning from COLD\_START to WARM\_START. The function updates the operating mode of the current executing partition.

**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 [51].

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 defined as $s_0 = \text{system\_init\_conf}$. The $\text{system\_init\_func}$ 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 $\text{partconf}$ is a field of $\text{Sys\_Config}$ with the type $\text{partition\_id} \rightarrow \text{Partition\_Conf}$ to store the configured partitions.

Fig. 5: Channel-based Communication in ARINC 653
manipulated port should be a source (is_source_port s p) and queueing (is_a_queueing_port 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_queueing), the service just omits the message. Otherwise, the message is inserted into the buffer (insert_msg2queueing_port).

definition send_queue_msg_lost :: "Sys_Config ⇒ State ⇒ Message ⇒ (State × bool)" where
send_queue_msg_lost sc s p m =
(if (¬ is_a_queueing_port s p ∨ ¬ is_source_port s p)
then (s, False)
else if is_full_queueing sc s p then (s, True)
else (insert_msg2queueing_port s p m, True))

The message transmission on channels is shown in Fig. 5. For instance, the message transmission in queueing 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_queueingport) and inserted into the buffer of the destination port. When the buffer of the destination port is full (is_full_queueing), the message is discarded.

primrec transf_queue_msg_lost :: "Sys_Config ⇒ State ⇒ Channel_Type ⇒ State" where
transf_queue_msg_lost sc s (Queueing 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 (sp))
then let sm = remove_msg_from_queueingport s (sp) in
if is_full_queueing sc (fst sm) (dp) then s
else insert_msg2queueing_port (fst sm) (dp) (s (snd sm))
else s)
transf_queue_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 ~ for the scheduler. The definition of ~ 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).

∀ s ∈ E. R(s) → (∃ s' . (s, s') ∈ 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_QueueingPort event, which is an instance of UCEs of the event.

Definition 8 (Concrete SC(e) of Creating_QueueingPort).

∀ d t s t' p. R(s) ∧ R(t) ∧ s ▷ d t ∧ s ≈ t ∧ is_part_con_fs (cur s) ∧ (cur s) \sim d ∧ s \sim s t ∧ s' = fst (create_queueing_port_confs s p)
∧ t' = fst (create_queueing_port_confs t p)
→ s' ▷ d t'

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). 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 :: 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 . d . t as follows. The state equivalence on new state variables, i.e. s ≈ d t in Definition 6, is defined as s . d . 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 ((¬ \sim . \sim . \sim))
where vpeqR s d t ≡ (R s) ≈ d (R t) ∧ (s~d~t)
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:**

```
definition start-process :: StateR ⇒ process-id ⇒ StateR where
  start-process s p ≡
    if p ∈ (the (proc-state (cur s))) ∧ (state (the (proc-state s) (cur s, p))) = DORMANT
    then NORMAL otherwise ⇒
    start = (if the (proc-state 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 selp (the state s))
    s (proc-state := proc-state')) in
      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:**

```
definition schedule-process :: StateR ⇒ StateR where
  schedule-process s ≡
    if is-part conf (cur s) ∧
      (the (proc-state s)) = NORMAL then
      let s' = (setRun2Ready s);
        readyprs = p p ∈ (the (proc-state s)) ∧
        state (the (proc-state s) (cur s')) = READY;
        selp = Some p;
        pc (x. state (the (proc-state s)) (cur s')) = READY ∧
        (∀ y ∈ readyprs. priority (the (proc-state s) (cur s')) ≥
          priority (the (proc-state s) (cur s')))
        st = the (proc-state s) (cur s', selp);
        proc-st = proc-state s, cur-pr = cur-pr s' in
      s (proc-state :=
      proc-proc-part := cur-prs (cur s', 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_mode:**

```
definition set_partition_modeR :: Sys-Config ⇒ StateR ⇒ partition-mode-type ⇒ StateR where
  set_partition_modeR sc s m ≡
    if (partition s) (cur s) sels m = None ∧
      (the (partition s) (cur s)) = COLD-START
    then NORMAL otherwise ⇒
      let pts = partitions s;
        s' = (if m = NORMAL
             then set-procs-to-normal s (cur s) pts
             else if (the (partition s) (cur s)) = NORMAL
             then remove_partition_resources s (cur s)
             else s) in
      s (partition := 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 \( SC_A ∩ LR_A \) to show the information-flow security, since the satisfaction of \( SC_A ∩ LR_A \) in \( SK_A \) has been proven at the top level and we have \( SK_A ⊆ SK_c \). 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 (\( SC_A(c) \)) of the event.

**Definition 9 (Concrete SC_A(c) of Schedule_Process):**

```
∀ d s t s' t' p. \( R_c(s) ∩ R_c(t) ∩ s . \sim_A . t \) \( ∃ . s . s' \sim_A . t \)
∧ is_part conf (cur s) \( ∩ \ (cur s) . s ∼_A . t \) \( ∩ s' \in schedule_process s \( ∩ \) t' \in schedule_process t \)
⇒ s' . \sim_A . t'.
```

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.

### 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
TABLE 2: Specification and Proof Statistics

<table>
<thead>
<tr>
<th>Item</th>
<th>Specification</th>
<th>Proof</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of type/def</td>
<td>LOC</td>
<td># of lemma/theorem</td>
</tr>
<tr>
<td>Security Model</td>
<td>25</td>
<td>130</td>
<td>63</td>
</tr>
<tr>
<td>Top-level Specif</td>
<td>116</td>
<td>680</td>
<td>193</td>
</tr>
<tr>
<td>Refinement</td>
<td>6</td>
<td>100</td>
<td>42</td>
</tr>
<tr>
<td>2nd-level Specif</td>
<td>45</td>
<td>330</td>
<td>111</td>
</tr>
<tr>
<td>Total</td>
<td>192</td>
<td>1,280</td>
<td>409</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Covert Channel</th>
<th>ARINC standard</th>
<th>VxWorks 653</th>
<th>XtratuM</th>
<th>POK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

*: existing; *: potential; blank: not existing

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. 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 [52], 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 [49], 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 using CSimpl [53] and its rely-guarantee framework 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


