IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 2, 2018, accepted March 10, 2018, date of publication March 14, 2018, date of current version April 18, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2815766

A Verified Capability-Based Model for Information Flow Security With Dynamic Policies

JIANWEN SUN[1], XIANG LONG[1], AND YONGWANG ZHAO[2,3]

[1]State Key Laboratory of Virtual Reality Technology and System, School of Computer Science and Engineering, Beihang University, Beijing 100083, China
[2]Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing 100083, China
[3]State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing 100083, China

Corresponding author: Yongwang Zhao (zhaoyw@buaa.edu.cn)

This work was supported in part by the Project of State Key Laboratory of Software Development Environment under Grant SKLSDE-2017ZX-13 and in part by the Key Research Fund of the MIIT of China under Project MJ-Y-2012-07.

ABSTRACT Formal verification of information flow security with dynamic policies of security-critical systems is a grand challenge. This paper presents the first effort to formally specify and verify a capability-based system model with dynamic information flow policies. We build a generic security model with dynamic security policies. In the security model, we define a set of information flow security properties and provide an inference framework for them. Based on the security model, we propose a system model for capability-based secure systems. The system model specifies critical events including system initialization, inter-domain communication, and capability management. We prove information flow security of the capability-based model by an unwinding theorem. Formal specification and security proof are carried out in the Isabelle/HOL theorem prover and could be applied to formally develop and verify the security of capability-based secure systems, such as separation kernels and secure hypervisors. To our knowledge, this is the first machine-checked proof of capability-based information security with dynamic policies.

INDEX TERMS Information flow security, dynamic noninterference, capability, formal specification, Isabelle/HOL, theorem proving.

I. INTRODUCTION

In recent years, an exponential growth has been achieved in the development of Cyber-Physical Systems (CPS). A large number of such systems have been deployed in our daily life, including many safety-critical and security-critical applications. Protecting the confidentiality of information manipulated by such systems is an increasingly important problem [1], [2]. Current systems provide little assurance to protect data confidentiality and integrity. This is not just because those existing theoretical frameworks are still unpractical to express these security properties, but also that common techniques for enforcing these security properties are unsatisfactory [3].

The standard mechanism to enforce confidentiality is information flow control [3], [4], which must indicate that information cannot flow to a component where the information policy is violated. Noninterference [5], as a security property, formalizes the absence of unwanted information flows within a system. Here, a system is divided into protection domains that represent distinct components of the system, and the information flow permitted between domains is specified by an information flow policy. To guarantee that a system is secure, we must prove that the system enforces a particular policy. Noninterference provides a framework to reason whether the information flow policy is enforced or not. A large number of works [6]–[10] have been done to verify that noninterference policies hold in several separation kernels formally.

Capability-based systems argue that a strong underlying protection model can improve software robustness and security [9], [11], [12]. In these systems, capabilities configuration could change as a result of system reconfiguration, transfer of authorities, personal account promotion, revocation of privileges and for many other reasons [13]. To verify intransitive noninterference of such systems, it is essential to establish a formal model focusing on dynamic policies with capability-based designs.

Most of the studies on proving intransitive noninterference for operating system kernels have only concerned static policies, such as PikeOS [14], the seL4 project [7], [9] and CertiKOS [15], where the information flow policy is a static configuration and does not change during system running.

A static security policy is incapable of expressing dynamic interactions between domains and is impractical for modern system designs.

In decentralized information flow control(DIFC) models [16]–[18], tags are used as the representation of policy and capabilities are adopted as the mechanism to enforce the secure change of these tags. Krohn *et al.* [16] and Krohn and Tromer [19] provided a noninterference proof of the Flume operating system with a very abstract CSP model [20]. Several operating systems used endpoint capabilities to enforce the security of inter-process communication mechanism [11], [21], [22], while the secure change of capabilities configuration, such as capability transfer and revocation, has not been verified formally.

A machine-checked noninterference theory focusing on dynamic policies are needed to build a dynamic execution model. To date, only a few works have studied this issue [13], [23]. Leslie [23] gave a pen-and-paper proof for dynamic intransitive noninterference based on Rushby's [24] intransitive purge method. Eggert and Meyden [13] developed an alternate semantics for dynamic intransitive noninterference policies with TA-security, and provided sound and complete proofs. A machine-checked dynamic intransitive noninterference theorem has not been provided yet.

In this paper, we propose a capability-based information flow control model with dynamically changeable security policies and provide machine-checked proof on its security properties. The security policies in this model are represented as capabilities. Thus, the operations of changing policies can be modeled as capability transferring or revocation. Since the infrastructure of capabilities has been provided by the capability-based systems, such as Barrelfish [22], our model can be used to reason about the information security of such systems.

Because information-flow security is a sort of hyperproperties [25], automatic verification of information-flow security of OS kernels is difficult and needs exhausting manual efforts. Theorem proving based approaches offer a comprehensive analysis of the system and are independent of the specific configuration. It not only analyses the safe execution model of the kernel, but also provides a full specification and proof for the kernel's precise behaviors. Therefore, we use Isabelle/HOL to conduct all the specifications and security proofs. Isabelle/HOL is a prover for higher-order logic with strong soundness properties: all derivations must pass through a proof kernel. The specifications and proofs of our model are sound and have passed through the Isabelle proof kernel. Our work is available at ''https://github.com/capflow/capflow''.

Figure 1 shows our methodology. We start from a security model with dynamic security policies. This model is an abstract configuration, which consists of a state machine and its security configuration. We propose a set of dynamic information flow security properties and prove the dynamic noninterference theorem of the security model. Based on this theorem, we prove the implication relations between the dynamic security properties which are called the dynamic inference framework.

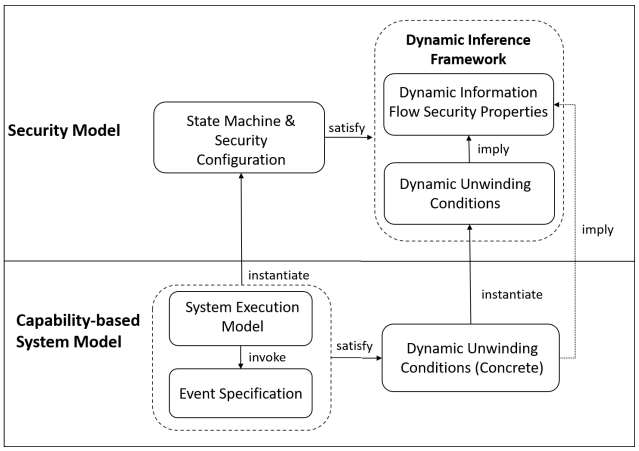

FIGURE 1. The proposed methodology.

Based on this theorem foundation, we specify a capability-based system model. In the system model, we use capabilities as the information flow policies. The events in the system model include inter-domain communications and capability transferring. We establish the security of the system model using the noninterference theorem of the security model.

To summarize, the technical contributions of this work are as follows.

1) We define a parameterized security model with dynamic information flow policies.
   A set of information flow security properties is defined in this model. We provide dynamic unwinding conditions and prove a theory of dynamic intransitive noninterference policies based on this model.
2) An inference framework of abstract model is presented to sketch out the implications of security properties. We provide the security proofs to indicate the relationship between different information flow security properties of the specification.
3) We develop a concrete execution model and event specification for a capability-based system, where capabilities are used to represent information-flow policies. The event specification of execution model covers system initialization, inter-domain communication, and capability management. Proofs for security properties of each event are provided. Security is proven by noninterference theorem instantiated from the security model.
4) We use more than 5000 lines of specification and proof in Isabelle/HOL [26] to formalize the capability-based model and prove information-flow security.

The rest of this paper is organized as follows. Section II discusses the related work. Section III introduces some basic notations of this paper. Section IV presents the security model for dynamic noninterference and shows the unwinding conditions, the security properties and inference framework focusing on dynamic policies. Section V explains how we

build the capability-based system model and describes its formal specification. Section VI shows formal verification for the dynamic unwinding conditions of every event in the system model. Section VII discusses the proof efforts and the reusability of our formal specifications and proofs. Section VIII gives the conclusion and future work.

II. RELATED WORK

In recent years, noninterference has been studied widely in the field of Information flow control [3]. Rushby [24] developed automation based noninterference and showed that Bell and La Padula's [27] access control model could be specified using transitive noninterference. And he also proved that intransitive noninterference, which is more suitable for modern OS kernels, could express channel-control policies. Van der Meyden [28] proposed an alternative, stronger, definition of security, which is shown to provide a better fit for Rushby's proof theory. For the dynamic policy, Eggert and Meyden [13] developed an alternate semantics for dynamic intransitive noninterference policies that generalize an adequate notion of TA-security in the static case. And Leslie [23] proposed a dynamic noninterference model based on Rushby's work. Von Oheimb [29] extended the Rushby framework to nondeterministic systems and supplemented noninterference with the notion of nonleakage. Nonleakage complements noninterference by exposing information flow, and these two concepts are integrated into a security definition called noninfluence. Reference [10] and [30] formally defined all of these properties and their variants, and proposed an inference framework to clarify the implications of these properties. In our work, we inherit the definitions from Zhao's work [10], [30] in the static situation and prove this framework for our dynamic model.

As intransitive noninterference is a property that any secure OS kernel needs to enforce, it is natural to prove this property for the implementations of these kernels. Krohn *et al.* [16] and Krohn and Tromer [19] presented a formal model and proof for the Flume operating system. However, since Flume is built as a library in Linux system, its trusted computing base includes the entire Linux kernel. PikeOS [14] provided a formal API specification on its separation kernel with verified intransitive noninterference. Different from our model, this work does not consider dynamic policies. The seL4 project [7], [9] verified a realistic microkernel for a machine-checked intransitive noninterference. Sewell *et al.*'s proof [21] of integrity and authority confinement shows that any modification that the current subject can perform is permitted by the authority represented in an upper bound on authority within a system, which means access control policy can only change within this upper bound. The information flow policy in seL4, which is inherited from access control policy at system initialization, is a static configuration and will not be changed during system state transaction. In the verification of concurrent OS kernels, CertiKOS [15] provided a certified kernel with machine-checkable contextual correctness proofs without sacrificing assembly performance. The difference between our work and seL4 is that the security policies in our work are dynamically changeable and we give a formal specification and verification of those policy-changing operations.

The capability has long been used in systems for security concerns [9], [11]. In theory, capabilities alone suffice to implement mandatory access control. EROS [11] achieved military-grade security by isolating processes into compartments and successfully realized the capability-based security principles on modern hardware. Influenced by EROS, seL4 [9] used capability as a high-level access control model and formally prove this model can enforce system-global security and authority confinement. In Distributed Information Flow Control (DIFC) systems [16], [17], [31], capabilities were used as distributed privilege. The asbestos system [31] implemented capability-like policies within its label mechanism to express the dynamic changes of the labelling authority. CHERI [12] extended a conventional RISC instruction-set architecture with capabilities and used a hardware-software object-capability model to realize application compartmentalization. Our capability-based mechanism is inspired by seL4 [21] and take-grant model [32]. The original analysis of the take-grant model [32] used the approximation method to model the exposure and propagation of access rights. The difference here is that the propagation rules of rights in our model are restricted by the unwinding conditions of dynamic noninterference and we conduct all proofs in Isabelle/HOL.

III. BASIC NOTATIONS

Isabelle is a generic and tactic-based theorem prover. We use Isabelle/HOL [26], an implementation of high-order logic in Isabelle, for our development.

The keyword **datatype** is used to define an inductive data type. Lists are one of the essential datatypes in Isabelle. The list is defined as **datatype** $'a\ list = Nil$ ("[]") $\mid Cons\ 'a$ " $'a\ list$ ". Here, lists have two constructors: *Nil*, the empty list (noted as "[]"), and *Cons* (can be abbreviated into #), which puts an element (of type $'a$) in front of a list (of type $'a\ list$). The polymorphic option type is defined as **datatype** $'a\ option = None \mid Some\ 'a$. It can model the result of a computation that may either terminate with an error (represented by *None*) or return some value v (represented by *Some* v).

The **definition** command is used for non-recursive definition and recursive definition is made by the **primrec** command. The recursive definition is of the form: **primrec** *name* :: *type* **where** *equations*. The equations must be separated by $\mid$. Isabelle uses the $\Rightarrow$ arrow for the function type, e.g. $'a \Rightarrow' b \Rightarrow' c$ is the type of a function that takes an element of $'a$ and an element of $'b$ as inputs, and returns an element of $'c$ as output. The following shows an example of recursive definitions.

```
primrec app :: " 'a list ⇒ 'a list ⇒ 'a list"
 where "app [] ys = ys" |
       "app (x # xs) ys = x # (app xs ys)"
```

In this example, the equations for *app* mean that *app* appends two lists. Instead of *Nil* and *Cons x xs*, we can write [] and *x#xs*, which is an alternative syntax in Isabelle/HOL.

A **record** of Isabelle/HOL covers a collection of fields, by selecting and updating operations. Each field has a specified type, which may be polymorphic. For example, the *point* type can be defined as follows.

record point = Xcoord :: int
Ycoord :: int

Records of type point have two fields named Xcoord and Ycoord, and both of types are int. Record type expression updates the value of specific field, such as $pt(|Xcoord = 8, Ycoord = 8|)$.

Locales are Isabelles approaches to deal with parametric theories. They have been designed as a module system for a theorem prover that can adequately represent the complex inter-dependencies between structures found in abstract algebra. The command **interpretation** is for the interpretation of locale in theories. Locales generalise interpretations from theorems to conclusions, enabling the reuse of definitions and other constructs that are not part of the specifications of the locales. The locale interpretation assigns concrete data to parameters and is called instantiation in our paper. The pair (a_1, a_2) is of type $\tau_1 \times \tau_2$ provided each a_i is of type τ_i. The function *fst* and *snd* extract the components of a pairs: $fst(x, y) = x$ and $snd(x, y) = y$. The **consts** command is used to declare a constant. The *simp*, *auto*, and *blast* are proving methods used in Isabelle/HOL.

For the sake of readability, we will use standard mathematical notations as much as possible. We will show examples of formal specifications in the Isabelle/HOL syntax.

IV. SECURITY MODEL WITH DYNAMIC POLICY

A. DYNAMIC NONINTERFERENCE

We use noninterference to provide a formal theorem foundation for reasoning about information flow security of our proposed model. The idea behind noninterference is that, for two domains u and v, if no action performed by u can influence the subsequent observations seen by v, we say u is noninterfering with v. The classical theory of noninterference assumes a transitive security policy, which can be divided into partially ordered security levels. In language-based information-flow security [3], program variables and data structures are assigned with *High* or *Low* labels and the confidentiality of secret data is realised by preventing information leakage from High-level data to Low-level data. Transitive noninterference is unable to express channel control policies and thus is declassified as intransitive one [19]. In intransitive case, the system should allow specific flows of information from *High* domains to *Low* domains, if the secret information is carefully declassified. In Rushby's work [24], intransitive noninterference is defined based on state machines and concerns the visibility of events, i.e. some secrets caused by certain events.

In dynamic security model, as information flows between domains, the security policy may change over time. The actions in the system can change not only the observable state of information distribution, but also the information-flow policy. The policy can be reconfigured according to upper-level needs from users, which is flexible for secure configuration of modern systems.

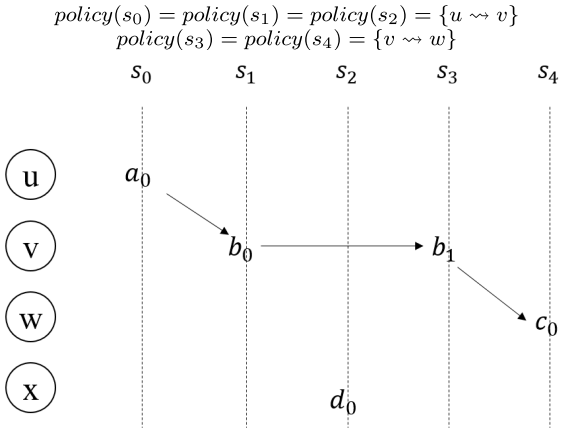

FIGURE 2. An example of dynamic information flow policy.

The dynamic policy could provide a fine-grained channel control mechanism. Figure 2 shows an example. In the diagram, columns represent system states (e.g. s_0 to s_5) and rows represent the domains in the system (e.g. domains u, v, w and x). The elements in this diagram represent the actions, e.g. a_0 executes in s_0 producing s_1 and a_0 is invoked by domain u. Note that $u \rightsquigarrow v$ means the system permits information flowing from u to v. That is, u can interfere with v. An arrow between two actions indicates interference. For example, the arrow from a_0 to b_0 means u can interfere with v through action a_0. In this example, the security policy is configured as $u \rightsquigarrow v$ initially. In state s_2 (after a_0 and b_0 execute), domain x invokes an action d_0 to change the security policy from $u \rightsquigarrow v$ to $v \rightsquigarrow w$ such that v can interfere with w. Finally, information flowing from u to w is possible through action b_1.

For a system configured with dynamic policy, the communication capability between domains can be configured with least interference relationship. This idea is based on the concept of least privilege principle. It aims to provide domains with enough but irredundant ability to reduce the risks of any unintended actions.

Although the dynamic security policy is more flexible and can express security requirements more accurate, it makes an information-flow control model more complicated and increases the difficulty of security analysis. The system model must be carefully adjusted to build a formal noninterference proof.

B. STATE MACHINE AND SECURITY CONFIGURATION

The information-flow security we discussed is based on a state machine, which represents the system model. The state machine is defined as follows.

Definition 1 (State Machine for Dynamic Information-flow Policies): A state machine, $M = \langle S, A, step, s_0 \rangle$, is a tuple, where S is a set of states, A is a set of actions, *step* is the state-transition function and $s_0 \in S$ is the initial state.

The function $step :: 's \Rightarrow 'a \Rightarrow 's$ defines the single-step execution of the system is defined as a function. Here $'s$ and $'a$ are the abstract datatypes for states and actions. $s' = step(s, a)$ means that the system state transits from s to s' after a executes.

```
primrec run :: "'s ⇒ 'a list ⇒ 's"
  where run_Nil: "run s [] = s" |
        run_Cons: "run s (a#as) = run (step s a) as"
definition _reachable :: " 's ⇒ 's ⇒ bool"
        "_reachable s1 s2 ≡ (∃as. run s1 as = s2 )"
definition reachable :: " 's ⇒ bool"
        "reachable s ≡ _reachable s0 s"
```

Auxiliary function $run :: 's \Rightarrow 'a\ list \Rightarrow 's$ represents the execution of a sequence of actions (of type $'a\ list$), e.g. $run(s, as)$ represents the final state produced by executing a sequence of actions *as* from state s. With the *run* function, $reachable :: 's \Rightarrow 's \Rightarrow bool$ defines the reachability of a state executing from the initial state s_0. Here *bool* is the type of truth values in Isabelle/HOL.

Actions in the system are invoked by a set of domains, D. Domains represent the protection components in a system, i.e. processes in computer systems. The relationship between actions and domains are mapped by a function $domain :: 'd \Rightarrow 'a$, where $'d$ is the abstract datatype for domains.

As the information-flow policy in our model is dynamic, the interference relationship is related to system state. Each state in the system has a policy configuration, which represents current security policies for that state. Interference relationships between domains are defined as $interferes :: 'd \Rightarrow 's \Rightarrow 'd \Rightarrow bool$. A domain u can interfere with v in state s (denoted as $u@s \rightsquigarrow v$) if information is allowed to flow from u to v, which means that v can observe the effects of u's action.

State equivalence, which is also called the view-partitioning equivalence, is used to check whether observations of the system are identical in the different states. The equivalence relation $\sim$ is defined as $vpeq ::' s \Rightarrow 'd \Rightarrow 's \Rightarrow bool.\ vpeq(s, d, t)$ (denoted as $s \sim d \sim t$) means that states s and t are identical for domain d, which is to say states s and t are indistinguishable for domain d. State equivalence relation for a set of domains, $\approx$, is defined naturally from $\sim$, where $s \approx D \approx t \equiv \forall d \in D.s \sim d \sim t$.

Based on the above definitions, we define security model as follows.

Definition 2 (Security Model for Dynamic Information-flow Policies):

$$SM = \langle M, D, domain, interferes, vpeq \rangle,$$

with assumptions as follows.

1) $\forall s, t, d, r.(s \sim d \sim t) \wedge (t \sim d \sim r) \longrightarrow (s \sim d \sim r)$
2) $\forall s, t, d.(s \sim d \sim t) \longrightarrow (t \sim d \sim s)$
3) $\forall s, d.(s \sim d \sim s)$
4) $\forall s, d.(d@s \rightsquigarrow d)$
5) $\forall s, t, v, u.(s \sim u \sim t) \longrightarrow (v@s \rightsquigarrow u) \leftrightarrow (v@t \rightsquigarrow u)$

In Isabelle/HOL, the security model is represented as a locale *SM*, which can be instantiated to concrete functional specification. In Definition 2, assumptions 1), 2), 3) are transitivity, symmetry and reflexive of *vpeq* relation. Assumption 4) is reflexive of *interferes* relation. Assumption 5) is a particular property of our dynamic situation. Unlike static policy, $v@s \rightsquigarrow u$ may not equal with $v@t \rightsquigarrow u$ because dynamic policies are not the same as the state changes. This assumption captures the relationship between *vpeq* and *interferes* in the dynamic case. If states s and t are identical for a domain u, all the observations are the same from the perspective of the domain u. Interference relationship means u can observe the effects of the actions invoked by any domain which can interfere with u, which means that those domains interfering with u should be equal in states s and t. Assume there is a domain v such that $v@s \rightsquigarrow u$ and $v@t \not\rightsquigarrow u$. It means u is able to receive a message from v in s while u cannot receive any information from v in t. And then u can tell the difference between s and t, which is contrary to the hypothesis. Thus $v@s \rightsquigarrow u$ and $v@t \rightsquigarrow u$ must be equal if $s \sim u \sim t$ is true. That is, for states s and t, the information-flow policy interfering with u is equivalent. Assumption 5) is an unwinding condition in Leslie's proof [23]. However, this paper argues that this assumption is a property of *interferes* and *vpeq* and does not correlate with any state transition.

C. INFERENCE FRAMEWORK OF INFORMATION-FLOW SECURITY PROPERTIES

The definitions of noninterference commonly use the idea of purging. Applying the purging function to the history of executed actions of the system removes some interfering actions. Usually, the interference defined by purging, is to compare the system states produced by executing the original sequence of actions and the purged ones. In this paper, function *purge* is used to remove actions which are non-interfering with the target domain from the original action sequence. Function *purge* needs a helper function, *sources*, to decide who have leaked information to the target domain after the action sequence executes.

```
primrec sources :: " 'a list ⇒ 'd ⇒ 's ⇒ 'd set" where
  sources_Nil: "sources [] d s = d" |
  sources_Cons: "sources (a # as) d s =(⋃{sources as d (step s a)}
                ∪ {w . w = the (domain a) ∧ (∃v . interferes w s v
                ∧ v∈sources as d (step s a))}"
```

Function *sources* is inductively defined on action sequences. In our dynamic case, *sources* needs to take state s into its argument list as the policy may change during executions. Given a non-empty action sequence, $(a\#as)$, *sources* checks the interference relation in s between $domain(a)$ and any domain in $sources(as, u, step(s, a))$ to see whether a can directly interfere with $sources(as, u, step(s, a))$. If an intervening action in *as* allows information to flow from $domain(a)$ to u, $domain(a)$ should be included in

TABLE 1. Information-flow security properties for dynamic security model.

No.	Security Properties
1	**noninterference** $\forall d, as.\ s_0 \upharpoonright\!\!\rightarrow as \cong d \cong s_0 \upharpoonright\!\!\rightarrow (ipurge(as, d, s_0))$
2	**weak_noninterference**: $\forall d, as, bs.\ ipurge(as, d, s_0) = (ipurge, bs, d, s_0)$ $\longrightarrow (s_0 \upharpoonright\!\!\rightarrow as \cong d \cong s_0 \upharpoonright\!\!\rightarrow bs)$
3	**noninterference_r**: $\forall s, d, as.\ reachable(s)$ $\longrightarrow s \upharpoonright\!\!\rightarrow as \cong d \cong s \upharpoonright\!\!\rightarrow (ipurge(as, d, s))$
4	**weak_noninterference_r**: $\forall s, d, as, bs.\ reachable(s)$ $\wedge ipurge(as, d, s) = ipurge(bs, d, s)$ $\longrightarrow (s \upharpoonright\!\!\rightarrow as \cong d \cong s \upharpoonright\!\!\rightarrow bs)$
5	**nonleakage**: $\forall d, as, s, t.\ reachable(s) \wedge reachable(t)$ $\wedge (s \approx (sources(as, d, s)) \approx t)$ $\longrightarrow (s \upharpoonright\!\!\rightarrow as \cong d \cong t \upharpoonright\!\!\rightarrow as)$
6	**weak_noninfluence**: $\forall d, as, bs, s, t.\ reachable(s) \wedge reachable(t)$ $\wedge (s \approx (sources(as, d, s)) \approx t)$ $\wedge ipurge(as, d, s) = ipurge(bs, d, t)$ $\longrightarrow (s \upharpoonright\!\!\rightarrow as \cong d \cong t \upharpoonright\!\!\rightarrow bs)$
7	**noninfluence**: $\forall d, as, s, t.\ reachable(s) \wedge reachable(t)$ $\wedge (s \approx (sources(as, d, s)) \approx t)$ $\longrightarrow (s \upharpoonright\!\!\rightarrow as \cong d \cong t \upharpoonright\!\!\rightarrow (ipurge(as, d, t)))$

sources(*a*#*as*, *u*, *s*). That is, *sources*(*as*, *u*, *s*) calculates the set of domains which may leak information to domain *u* when action sequence *as* executes from state *s*.

```
primrec ipurge :: " 'a list ⇒ 'd ⇒ 's ⇒ 'a list" where
 ipurge_Nil: "ipurge [] u s = []" |
 ipurge_Cons: "ipurge (a#as) u s =
                (if (the (domain a) ∈ (sources (a#as) u s))
                 then a # ipurge as u (step s a)
                 else ipurge as u (step s a))"
```

Purge function *ipurge* removes all the actions which cannot interfere with *u* directly or indirectly. Given a non-empty action sequence, *a*#*as*, *ipurge* checks first action *a* to see whether *a* can leak information to *u*, i.e. $domain(a) \in sources(a\#as, u, s)$. If this condition is false, which means *a* is non-interfering with target domain *u*, *a* is removed from *a*#*as*. And then, the purging process continues on remaining action sequence *as* from the next state *step*(*s*, *a*) when *a* executes. An observation equivalence function for an action sequence is defined as $s \upharpoonright\!\!\rightarrow as \cong d \cong t \upharpoonright\!\!\rightarrow bs \equiv run(s, as) \sim d \sim run(t, bs)$. It indicates that two final states, produced by executing *as* from *s* and executing *bs* from *t*, are identical for domain d.

The intransitive noninterference [24] in the dynamic situation is defined as *noninterference* in Table 1. Given a domain *d*, and a sequence of actions *as*, final states produced by executing *as* from s_0 and executing *as*'s purged action sequence from s_0, are identical to *d*. Its intuitive meaning is that actions of domains which cannot interfere with *d* should not affect the state of *d*, regardless of whether these actions are in the execution sequence.

Base-on the security properties for the static policy defined in [10] and [30], this paper redefines the security properties suitable for our proposed dynamic security model. Table 1 lists these properties.

A standard technique to prove that a system satisfies the security properties is called unwinding [24]. The unwinding approach defines well-behaved state transitions. A set of unwinding conditions specifies expecting properties for individual execution steps. An unwinding theorem provides formal proofs that security properties can be implied from the unwinding conditions. By proving that state transitions satisfy the unwinding conditions, we can establish the security properties of the security model. In the proof of our model, we define dynamic unwinding conditions as follows.

Definition 3 (Dynamic Step Consistent):

$$\begin{aligned} DSC \equiv \forall a, d, s, t.\ & reachable(s) \wedge reachable(t) \wedge (s \sim d \sim t) \\ & \wedge domain(a)@s \rightsquigarrow d \\ & \longrightarrow (s \sim domain(a) \sim t)) \\ & \longrightarrow (step(s, a) \sim d \sim step(t, a)) \end{aligned}$$

Definition 4 (Dynamic Local Respect):

$$\begin{aligned} DLR \equiv \forall a, d, s.\ & reachable(s) \wedge (domain(a)@s \not\rightsquigarrow d) \\ & \longrightarrow (s \sim d \sim (step(s, a)) \end{aligned}$$

The dynamic step consistent condition (DSC in Definition 3) specifies under what conditions if an actions *a* can interfere with a domain *d*, two identical states *s* and *t* for *d* are still indistinguishable after *a* executes. It requires that *s* and *t* are identical for both *d* and *domain*(*a*) (the domain who invokes *a*). The intended meaning of dynamic local respect (DLR in Definition 4) is that if an action does not interfere with a domain, the effects of the action cannot be observed by that domain, e.g. if actions *a* does not interfere with *d* in state *s*, the state produced by executing *a* (i.e. *step*(*s*, *a*)) is identical with *s* for domain *d*.

As the interference relationship is dynamic in our model, proving the unwinding theorem is entirely different from the static case. The functions *sources* and *ipurge* are state-dependent in our security model, e.g. *sources*(*as*, *u*, *s*) and *sources*(*as*, *u*, *t*) may not be equivalent in the dynamic case. This makes it complicated to prove the dynamic unwinding theorem. We provide equivalence lemmas for these two functions first.

Note that in Leslie's proof [23], sources equivalence is implied by $s \sim u \sim t$, i.e. for two equivalent states *s* and *t*, *sources* is equivalent after the same sequence of actions *as* executes, which is a property following directly from the definition of sources [23]. This lemma cannot be proved in our theorem prover because $s \sim u \sim t$ can only guarantee that policies associated with *u* are equivalent, but can not ensure that all the policies associated with *sources*(*as*, *u*, *s*) are equivalent. Figure 3 shows such an example. In state *s*, as shown in Figure 3 (a), domain *v* can interfere with domains *u* and *w* can interfere with *v*. Actions *a* and *b* are invoked by domains *w* and *v*, respectively. After actions [*a*, *b*] execute, *sources*([*a*, *b*], *u*, *s*) = {*u*, *v*, *w*}. While in state *t*, as shown in Figure 3 (b), the security configuration is the same as *s*

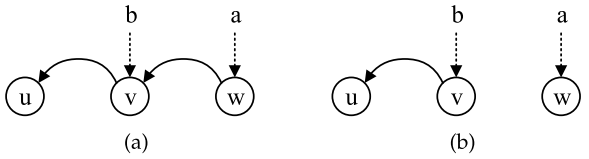

FIGURE 3. Indirect information flow policies affect the equivalence of *sources*.

except that w can not interfere with v. From u's perspective, s and t are identical. However, after actions $[a, b]$ execute, $sources([a, b], u, t) = \{u, v\} \neq sources([a, b], u, s)$. Thus, we argue that the necessary condition of sources equivalence in this example is $s \approx sources(as, u, s) \approx t$, other than $s \sim u \sim t$. Moreover, the equivalence of sources also needs to take into account that the state transitions satisfy the unwinding conditions.

Lemma 1 (Sources Equivalence):

$$\forall s, t, as, u.reachable(s) \wedge reachable(t) \wedge DSC \wedge DLR \wedge (s \approx sources(as, u, s) \approx t) \longrightarrow sources(as, u, s) = sources(as, u, t)$$

In our proof, Lemma 1 ensures *sources* function calculates a correct set of domains for two equivalent states. The premise of this lemma requires the equivalence of states on $sources(as, u, s)$ and the satisfaction of unwinding conditions. Based on the *sources* equivalence, we prove *ipurge* equivalence as follows.

Lemma 2 (Ipurge Equivalence):

$$\forall s, t, as, u.reachable(s) \wedge reachable(t) \wedge DSC \wedge DLR \wedge (s \approx sources(as, u, s) \approx t) \longrightarrow ipurge(as, u, s) = ipurge(as, u, t)$$

Lemma 2 ensures *ipurge* function behaves consistently for two equivalent states. This lemma is a fundamental precondition to prove unwinding theorem.

Lemma 3 (Dynamic Noninfluence Lemma):

$$\forall s, t, as, u.reachable(s) \wedge reachable(t) \wedge DSC \wedge DLR \wedge (s \approx sources(as, u, s) \approx t) \longrightarrow (s \leadsto as \cong d \cong t \leadsto (ipurge(as, d, t)))$$

Based on Lemma 1 and Lemma 2, we prove dynamic noninfluence lemma 3. Next, we prove dynamic unwinding theorem showed as follows.

Theorem 1 (Dynamic Unwinding Theorem): The security model SM satisfies that

$$(DSC \wedge DLR) \Longrightarrow noninfluence$$

Theorem 1 is proven by Lemma 3, which establishes security of *SM*. It builds connections between the unwinding conditions and *noninfluence* property. Because *noninfluence* is the most stringent among all security properties, it can serve as proof of other properties. For instance, if we assign t as s in *noninfluence*, we have $s \leadsto as \cong d \cong s \leadsto (ipurge(as, d, s))$, which is the definition of *noninterference_r*.

The dynamic inference framework clarifies these implication relationships. As shown in Figure 4, an arrow means

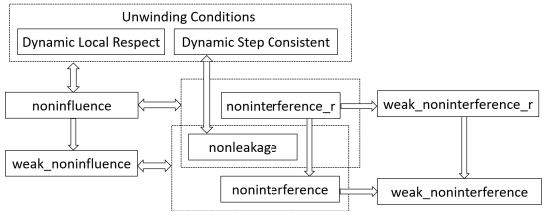

FIGURE 4. Dynamic Inference Framework of Security Properties.

the implication between security properties. We have proven all these implication relations and provide formal proofs in Isabelle/HOL.

In summary, the soundness and completeness of unwinding conditions and noninfluence in the security model are shown as follows.

Theorem 2 (Soundness and Completeness of Unwinding Conditions): The security model SM satisfies that

$$(DSC \wedge DLR) = noninfluence \text{ and } DSC = nonleakage$$

Theorem 3 (Soundness and Completeness of Noninfluence): The security model SM satisfies that

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

In Rushby's work [24], the proof method is sound but not complete. Our formal proof is both sound and complete, i.e. we are not only able to prove the system's security by the satisfaction of unwinding conditions but also prove the unsatisfied unwinding condition causes that system's insecurity.

V. CAPABILITY-BASED SYSTEM MODEL

We design a capability-based system model in this section. We first discuss the preconditions and limitations of instantiation. Then, we present its detailed specification. Finally, we instantiate the security model as the system execution model.

A. BASIC DEFINITIONS

We propose a capability-based information flow control model. In this model, there are two types of essential components: domain and endpoint. Domain here is an abstraction of a computer program which may interfere with each other. A domain owns a single execution environment which is independent of others. The meaning is the same as the process in general-purpose OS or partition in separation kernel. Influenced by EROS, seL4 and Barrelfish [9], [11], [22], we use endpoints to denote inter-domain communication (IDC) destinations. The endpoint is an abstraction of a communication node. It is an interface exposed by a communication channel. Each endpoint has a message buffer to store received messages. The endpoints are essential ports. The message buffer of endpoints, instead of being a part of domain's data structures, is a separate part of kernel objects.

The reasons to adopt capability in our concrete model are shown as follows. First, capabilities can be transferred to other holders, which makes capabilities capable of expressing dynamic changing operations. Second, influenced by take-grant protection model [32], capabilities transferring operations can be conducted by obeying specific rules. The capability-changing operation can be defined under preconditions for instantiation, which would make system model more reasonable for formal verification. Third, many capability-based operating systems, such as Barrelfish [22], can use our capability model to justify information flow security.

We uses capabilities to instantiate interference relation. In Isabelle/HOL, a capability is specified as a record with two fields: an identifier which assigns a name to the target domain, and a set of rights which defines the operation the owner is authorized to perform on the target.

$$\begin{aligned} &\textbf{datatype}\ \text{right} = \text{SEND} \mid \text{GRANT} \mid \text{REMOVE} \\ &\textbf{record}\ \text{cap} = \ \text{target} :: \text{domain\_id} \\ &\qquad\qquad\qquad\ \ \text{rights} :: \text{"right set"} \end{aligned}$$

Capabilities can be represented as a directed graph, where vertices are domains and edges are labelled with a set of rights. Each capability has associated with one or many rights. There are three fundamental rights in our capability model. Rights are defined as *datatype rights* $= Send|Grant|Remove$. *Send* right means that the owner of that capability can send messages to the target. It is the basic right to conduct an IDC. *Grant* right allows a domain to grant its capabilities to another domain. *Grant* right is inspired by take-grant protection model [32], who gives specific rules to capability systems to follow. *Remove* right allows a domain to remove rights it has over another domain.

B. SYSTEM EXECUTION MODEL AND INSTANTIATION PROOF

The execution model is statically configured at build time, except for its capability configuration and messages propagation because the information-flow policies (represented as capability configuration) are dynamic. Domains, endpoints, and the system configuration are defined in Isabelle/HOL as a const.

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

1) SYSTEM STATE AND TRANSITION

In our concrete dynamic model, the state of system concerns information-flow policies and information propagation. The information-flow policy is the capability configuration, which is represented by the capability set of each domain. Information propagates using sending or receiving messages. Information propagation is represented by message buffer on each endpoint. Message buffer has two fields we need to concern: message set and buffer size on that endpoint. We defined datatype *domain_id* and *endpoint_id* as the identifiers for domains and endpoints. For every state, we use a function to map *domain_id* to the set of capabilities the domain owns. Similarly, *endpoint_id* is defined to represent specified endpoint and functions are used to store the state of message buffer of each endpoint. System state transition is instantiated by the *exec_event* function, which is a case sentence to map events to related event execution specification.

$$\begin{aligned} &\textbf{record}\ \text{State} = \\ &\qquad \text{caps} :: \text{"domain\_id} \Rightarrow \text{cap set"} \\ &\qquad \text{e\_msgs} :: \text{"endpoint\_id} \Rightarrow \text{Message set"} \\ &\qquad \text{e\_buf\_size} :: \text{"endpoint\_id} \rightharpoonup \text{buffer\_size"} \\ &\qquad \text{domain\_endpoint} :: \text{"endpoint\_id} \rightharpoonup \text{domain\_id"} \end{aligned}$$

2) EVENTS

The events in our model are all system calls. These system calls are performed using capability invocation as tradition capability-based operating system does. The events can be divided into three types: requesting system states, inter-domain communication and capability management. Only capability management events can change the capability configuration of the system, i.e. the security policies. The other two types of events are data-operating events, which cannot change the system's security policies. We define each event as a state transition function, i.e. $exec\_event(s, a)$, which returns the next state when a executes.

$$\begin{aligned} &\textbf{datatype}\ \text{Event} = \\ &\qquad \text{Client\_Lookup\_Endpoint\_Name domain\_id endpoint\_name} \\ &\qquad \mid \text{Send\_Queuing\_Message domain\_id endpoint\_id Message} \\ &\qquad \mid \text{Receive\_Queuing\_Message domain\_id endpoint\_id} \\ &\qquad \mid \text{Get\_My\_Endpoints\_Set domain\_id} \\ &\qquad \mid \text{Get\_Caps domain\_id} \\ &\qquad \mid \text{Grant\_Endpoint\_Cap domain\_id cap cap} \\ &\qquad \mid \text{Remove\_Cap\_Right domain\_id cap right} \end{aligned}$$

3) EVENT DOMAIN

To make our model more general to be applied in distributed systems, we didn't adopt a scheduler to infer the domain of event at run-time. Instead, we assign *domain_id* as a parameter to each event. When a user application invokes a system call, the system decides which domain performs this action. We use *domain_of _event* to map an event to its domain, which is a concrete substitution of *domain* in *SM*.

4) INTERFERENCE RELATIONSHIP

Interference relationships are the security policies of the system. A domain can interfere with another domain using sending messages or transferring capabilities. All these events should be permitted by security policies, i.e. these events should be restricted by checking the capability configuration of that domain. For instance, if domain a tries to send a message to domain b, a must have a capability whose target is b and *Send* in its right set. If domain a tries to grant capability c to domain b, a must have a capability whose target is b and *Grant* in its right set. Briefly, domain a's action can interfere with domain b if a holds a capability whose target is b. The interference relation is instantiated as a function *interferes*. $interferes(w, s, v)$ checks

w's capability-set in *s* (*get_domain_cap_set_from_domain_id s w*) if there is a capability whose target is *v*. If the condition is true, *w* can interfere with *v* in state *s*.

```
definition interferes :: "domain_id ⇒ State ⇒ domain_id ⇒ bool"
  where "interferes w s v ≡
      if( w = v
        ∨ (∃ c. c ∈ (get_domain_cap_set_from_domain_id s w)
        ∧ target c = v))
      then True
      else False"
```

5) STATE EQUIVALENCE

The state equivalence ∼ is defined as a function *vpeq*1. For a domain *d*, if two states *s* and *t* are equivalent, it means that the domain state, the communication state, as well as the domain set which can interfere with *d* in *s* and *t* are the same. Domain state is the capability set of that domain in two states. The communication state includes the endpoints related to domain *d* and the message set for these endpoints. Endpoints for *d* are configured at build time. *vpeq*1 checks the message set of every endpoint of the target domain to make sure the communication state is the same. An example of the equality of domain set which can interfere with *d* in both *s* and *t* is that if a domain *v* can interfere with *d* in *s*, *v* must be able to interfere with *d* in *t*. This is because *d* is able to receive a message from *v* in the two states.

```
definition vpeq1 :: "State ⇒ domain_id ⇒ State ⇒ bool"
  where "vpeq1 s d t ≡
      let cs1 = get_domain_cap_set_from_domain_id s d;
        cs2 = get_domain_cap_set_from_domain_id t d;
        dom_eps1 = get_endpoints_of_domain s d;
        dom_eps2 = get_endpoints_of_domain t d
      in if(cs1 = cs2
          ∧ (∀ v. interferes v s d ⟷ interferes v t d)
          ∧ dom_eps1 = dom_eps2
          ∧ (∀ ep. ep ∈ dom_eps1
            ⟶ get_msg_set_from_endpoint_id s ep
              = get_msg_set_from_endpoint_id t ep ))
        then True
        else False"
```

6) INSTANTIATION OF SECURITY MODEL

The system execution model of system model is an instance of the security model. In order to assure the correctness of the instantiation, we provide following lemmas to ensure that the assumptions in Definition 2 are true respectively.

```
lemma vpeq1_transitive_lemma :
    "∀ s t r d. (vpeq1 s d t) ∧ (vpeq1 t d r) ⟶ (vpeq1 s d r)"
  using vpeq1_def by auto
lemma vpeq1_symmetric_lemma :
    "∀ s t d. (vpeq1 s d t) ⟶ (vpeq1 t d s)"
  using vpeq1_def by auto
lemma vpeq1_reflexive_lemma : "∀ s d. (vpeq1 s d s)"
  using vpeq1_def by auto
lemma interf_reflexive_lemma : "∀ d s. (interferes d s d)"
  using interferes_def by auto
```

TABLE 2. The description of new added events.

No.	Name	Description
1	client_lookup_endpoint_name	The client looks for the id of endpoint by its name
2	send_queuing_message	Send message to the message queue of remote endpoint
3	receive_queuing_message	Receive a message from local message queue
4	get_my_endpoints_set	Get local endpoint set
5	get_caps	Get local capability set
6	grant_endpoint_cap	Grant a local capability to destination domain
7	remove_cap_right	Remove a specified right of a capability locally

```
lemma policy_respect_lemma :
    "∀ v u s t. (s ∼ u ∼ t) ⟶ (interferes v s u = interferes v t u)"
  using vpeq1_def by auto
```

The instantiation is an interpretation of the *SM* locale, shown as follows. The parameters of *SM* are substituted for concrete ones, i.e. *s*0*t*, *exec_event*, *domain_of _event*, *vpeq*1 and *interferes*. Since the instantiated functions hold the same assumptions as parameterized security model, we successfully instantiate system model from the abstract security model.

```
interpretation SM
  s0t exec_event domain_of_event vpeq1 interferes
  using vpeq1_transitive_lemma vpeq1_symmetric_lemma
      vpeq1_reflexive_lemma interf_reflexive_lemma
      policy_respect_lemma reachable_top
      SM.intro[of vpeq1 interferes]
      SM_axioms.intro[of s0t exec_event]
      SM.intro[of vpeq1 interferes s0t exec_event] by blast
```

VI. EVENT SPECIFICATION AND SECURITY PROOFS FOR SYSTEM MODEL

A. EVENT SPECIFICATION

The event specification defines the functions to implement the concrete execution of events. The functionalities include system initialization, inter-domain communication and capability management. The names and descriptions of all events are listed in Table 2.

1) SYSTEM INITIALIZATION

The system initialization considers system's initial state. The system's initial state is defined as $s_0 = system\_init\ sysconf$. *system_init* is the system initialization function, which assigns initial values to initial state according to the system configuration.

```
consts s0t :: State
specification (s0t)
    s0t_init: "s0t = system_init sysconf"
    by simp
```

2) INTER-DOMAIN COMMUNICATION

The events related to inter-domain communication are sending a message to an endpoint and receiving a message from an endpoint. We use some helper functions to implement

one event. For instance, *send_queuing_message s did eid m* means a domain *did* sends message *m* to remote endpoint *eid* in state *s*. The helper function, *get_domain_id_from_endpoint s eid*, returns the target domain which owns that remote endpoint *eid* in *s*. This event changes the communication state of the target domain, i.e. it interferes with the target domain. Thus, the system checks the *interferes* relation to ensure security policies permit this message-sending event. If the destination domain exists and the current domain can interfere with it, the message is inserted into the buffer. Otherwise, the state of the system remains unchanged after this event executed.

```
definition send_queuing_message :: "State ⇒ domain_id
    ⇒ endpoint_id ⇒ Message ⇒ (State × bool)" where
  "send_queuing_message s did eid m ≡
    let
      dst_dom = get_domain_id_from_endpoint s eid;
      emsgs = e_msgs s;
      msg_set = get_msg_set_from_endpoint_id s eid
    in
      if (dst_dom ≠ None ∧ interferes did s dst_dom)
      then
        (s(|
          e_msgs := emsgs(eid := insert m msg_set)
        |), True)
      else
        (s, False)"
```

Event *receive_queuing_message* receives a message from the message queue of a local endpoint. The helper function *get_msg_set_from_endpoint_idseid* returns endpoint *eid*'s message queue in state *s*. To remove a message from message queue, the system model should check if the *domain_id* of *eid* is valid and the message queue is not empty. If these conditions are true, the message is returned to the event's invoker.

```
definition receive_queuing_message :: "State ⇒ Message option
    ⇒ endpoint_id ⇒ (State × bool)" where
  "receive_queuing_message s did eid ≡
    if(get_domain_id_from_endpoint s eid = Some did
      ∧ get_msg_set_from_endpoint_id s eid ≠ {})
    then let
      emsgs = e_msgs s;
      msg_set = get_msg_set_from_endpoint_id s eid;
      m = SOME x. x∈msg_set;
      new_msg_set = msg_set - {m}
    in (s(|
          e_msgs := emsgs(eid := new_msg_set)
        |), Some m)
      else
        (s, None)"
```

3) CAPABILITY MANAGEMENT

We define policy-changing events to reconfigure current security policies, i.e. current capability configuration of the system. These events make the information-flow policies dynamic. To change a specific edge of information-flow policies, the specification of that event must satisfy SM's assumptions and unwinding conditions.

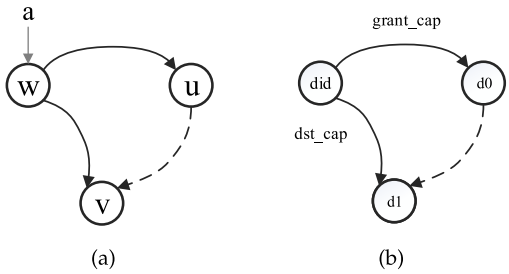

FIGURE 5. The grant operation in theorem model and event definition.

Consider the example in Figure 5 (a). Given an action *a*, a domain *w* who invokes *a* and another two domains *u* and *v*, we say that *a* changes the security policies by adding a new edge from *u* to *v* in state *s*. *w* must interfere with *u* first, because from *u*'s perspective, *u* can tell the difference between *s* and *step*(*s*, *a*), which makes DLR false. If *w* cannot interfere with *v* in state *s*, from the definition of DLR, $s \sim v \sim step(s, a)$ is true. Since *interferes* satisfies the assumption 5) of *SM*, we can deduce that $\forall x.x@s \rightsquigarrow v \leftrightarrow x@step(s, a) \rightsquigarrow v$, which means the interference relation from any domain to *v* is equivalent for *s* and *step*(*s*, *a*). This conclusion contradicts the fact that $u@s \rightsquigarrow v$ has been changed. Therefore, to add a new edge between *u* and *v*, *w* must interfere with both *u* and *v* in state *s* in this example.

In the specification of capability-granting operation, *grant_endpoint_cap s did grant_cap dst_cap* means that the current domain *did* grants a capability, *dst_cap*, to the target domain which *grant_cap* specifies. Here, we use the target field of *grant_cap* to represent the target domain instead of the domain's identifier. The reason is that it is more intuitive to use capabilities to represent the interference relation. Since this event adds a new edge to the security policies, as Figure 5 (b) shows, the current domain *did* should interfere with the target of *grant_cap* and the target of *dst_cap*, i.e. *did* owns both *grant_cap* and *dst_cap*. This event also needs to check whether *grant_cap* has the *GRANT* right and to ensure that it does not grant capability to itself. If all the prerequisites for the grant event are valid, the system inserts *dst_cap* to the capability set of *did_dst* (the target of *grant_cap*). Otherwise, the state of the system remains.

```
definition grant_endpoint_cap :: "State ⇒ domain_id
    ⇒ cap ⇒ cap ⇒ (State × bool)"
  where "grant_endpoint_cap s did grant_cap dst_cap ≡
    if(grant_cap ∈ get_domain_cap_set_from_domain_id s did
      ∧ GRANT ∈ rights grant_cap
      ∧ target grant_cap ≠ target dst_cap
      ∧ dst_cap ∈ get_domain_cap_set_from_domain_id s did)
    then let
      did_dst = target grant_cap;
      caps0 = caps s;
      cs_dst = get_domain_cap_set_from_domain_id s did_dst
      in (s (|
```

```
            caps := caps0(did_dst := (insert dst_cap cs_dst))
            |), True)
    else (s, False)"
```

A domain can remove its capability's rights by invoking *remove_cap_right*. The basic requirement to delete a capability is that the domain has the capability's REMOVE right. *remove_cap_right s did rm_cap right_to_rm* means domain *did* removes *rm_cap*'s right *right_to_remove*. If REMOVE is the only right of *rm_cap*, the system model will remove the entire capability because the unauthorized capability is meaningless. If *rm_cap* has more rights, its right *right_to_remove* will be removed. Otherwise, the model will not make any changes to its state.

```
definition remove_cap_right :: "State ⇒ domain_id
    ⇒ cap ⇒ right ⇒ (State × bool)"
  where "remove_cap_right s did rm_cap right_to_rm ≡
    let caps0 = caps s;
        cs_dst = get_domain_cap_set_from_domain_id s did;
        cs_rest = c. c∈cs_dst ∧ c≠rm_cap
    in if(rm_cap ∈ get_domain_cap_set_from_domain_id s did
          ∧ REMOVE ∈ rights rm_cap
          ∧ right_to_rm ∈ rights rm_cap
          ∧ REMOVE = right_to_rm
          ∧ REMOVE = rights rm_cap)
    then (s(|
              caps := caps0(did := (cs_rest))
            |), True)
    else if(rm_cap ∈ get_domain_cap_set_from_domain_id s did
          ∧ REMOVE ∈ rights rm_cap
          ∧ right_to_rm ∈ rights rm_cap)
    then let new_cap = âęĞ target = target rm_cap,
         rights = (rights rm_cap) - right_to_rmâęĹ
      in (s(|
              caps := caps0(did := (insert new_cap cs_rest))
            |), True)
    else (s, False)"
```

B. SECURITY PROOFS FOR SYSTEM MODEL

Since the system model's specification is an instance of the security model, its security properties can be implied by the satisfaction of the unwinding conditions in Definition 3 and 4. To prove the satisfaction of *DSC* and *DLR* on the system model's event specification, we can induct on every type of events and prove each concrete event satisfies two unwinding conditions. We provide a set of concrete unwinding conditions for all the events in the system model. Lemma 4 and 5 show the concrete conditions for event *Grant_Endpoint_Cap*. Note that *fst* in Isabelle/HOL means the first element in a tuple, a state of the system in this example.

Lemma 4: Concrete DLR of Grant_Endpoint_Cap::

$$\begin{aligned}&\forall s, d, did.\\ &\quad reachable(s) \wedge \neg interferes(did, s, d)\\ &\quad \wedge s' = fst(grant\_endpoint\_cap(s, did, grt, dst)))\\ &\quad \longrightarrow s \sim d \sim s'\end{aligned}$$

Lemma 5: Concrete DSC of Grant_Endpoint_Cap:

$$\begin{aligned}&\forall s, t, d, did.\\ &\quad reachable(s) \wedge reachable(t)\\ &\quad \wedge s \sim d \sim t \wedge interferes(did, s, d) \wedge s \sim did \sim t\\ &\quad \wedge s' = fst(grant\_endpoint\_cap(s, did, grt, dst)))\\ &\quad \wedge t' = fst(grant\_endpoint\_cap(t, did, grt, dst)))\\ &\quad \longrightarrow s' \sim d \sim t'\end{aligned}$$

The goals of Lemma 4 and 5 are to show the state equivalence, e.g. $s \sim d \sim s'$. In the definition of *vpeq*1, we can divide the single goal into four subgoals, i.e. the equivalences of domain d's capability set, d's endpoint set, the policies that interfere with d and all the message sets of d's endpoints. By proving these subgoals, we can prove $s \sim d \sim s'$.

Lemma 6: Concrete DLR of System Model:

$$\begin{aligned}&\forall s, d, a.\\ &\quad reachable(s)\\ &\quad \wedge \neg interferes(domain\_of\_event(a), s, d)\\ &\quad \longrightarrow (s \sim d \sim exec\_event(s, a))\end{aligned}$$

Lemma 7: Concrete DSC of System Model:

$$\begin{aligned}&\forall s, t, d, a.\\ &\quad reachable(s) \wedge reachable(t)\\ &\quad \wedge s \sim d \sim t \wedge s \sim domain\_of\_event(a) \sim t\\ &\quad \wedge interferes(domain\_of\_event(a), s, d)\\ &\quad \longrightarrow exec\_event(s, a) \sim d \sim exec\_event(t, a)\end{aligned}$$

After proving concrete unwinding conditions of every event, we inductively prove the unwinding conditions of system model in Lemma 6 and 7, which ensures all events in system model satisfy *DLR* and *DSC*. The state transition statement here is replaced by $exec\_event(s, a)$, which is more general.

Theorem 4 (Noninfluence of the System Model):

The system model satisfies *noninfluence*.

Finally, Thoerem 4 is proved. We can say that the system has the security property of *noninfluence*. Other properties in Table 1 of *SM* can be applied on our proposed model. These properties can be implied following the dynamic inference framework in Figure 4.

VII. DISCUSSION

We use Isabelle/HOL to specify and verify the security of our proposed capability-based information flow model. The proof of our model is understandable for both humans and computers. All these proofs have passed through the Isabelle proof kernel. We use 60 functions and 1000 lines of code of Isabelle/HOL to specify the dynamic security model and 130 functions and 4000 lines of proof to prove the security properties for the system model.

Our specifications and proofs can be used to verify the information security in capability-based operating systems, such as Barrelfish [22]. The reason why our proposed model

is suitable for Barrelfish system is as follows. First, the Barrelfish kernel is based on capability. Capabilities reference the kernel objects, and the system uses capabilities to manage system resources. All the system calls are implemented as capability invocations including inter-process communication (IPC). Since Barrelfish provides necessary data structures for different types of capabilities, we can implement our capability model in its kernel. Second, the kernel of Barrelfish is designed as a single-threaded execution model, which cannot be interrupted when executing privileged events. Thus, it is appropriate to use the state machine in our model to describe the behavior of the Barrelfish kernel. Third, Barrelfish is a micro-kernel design, and most of the system services are deployed as the user applications. IPC is critical for the system because the dispatching of computation tasks is heavily dependent on the collaboration between processes. If a formal model can verify the security of IPC, then the system can be applied in safety-critical areas. We need to tackle some challenges when applying our model to the Barrelfish system. The first is to mind the gap between the formal model and the high-performance implementation of Barrelfish kernel. We also need to focus on the impact on the kernel's performance, especially the IPC performance.

VIII. CONCLUSION

We have presented a reusable and extensible approach to build a verified capability-based execution model that has a machine-checked dynamic noninterference theorem foundation. We show that it is feasible and practical to build a secure information flow model with a dynamic policy. Our capability-based policy representation is machine-checkable, and it makes the information flow analysis concise and convincing. We believe that our execution model will open up a new efficient approach toward building secure and extensible systems.

In the next step, we will extend our system model to lower level specifications by adding services of complicated process management and capability manipulation. We will use the lower specifications to formally specify a capability-based OS kernel in implementation code level and prove its information-flow security.

REFERENCES

[1] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security—A survey," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1802–1831, Dec. 2017.

[2] R. Akella, H. Tang, and B. M. McMillin, "Analysis of information flow security in cyber–physical systems," *Int. J. Critical Infrastruct. Protection*, vol. 3, nos. 3–4, pp. 157–173, 2010.

[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[4] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *ACM SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 129–142, Oct. 1997. [Online]. Available: http://doi.acm.org/10.1145/269005.266669

[5] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. Secur. Privacy*, Apr. 1982, p. 11.

[6] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, "Formal specification and verification of data separation in a separation kernel for an embedded system," in *Proc. 13th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2006, pp. 346–355. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180448

[7] T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein, "Noninterference for operating system kernels," in *Certified Programs and Proofs*, C. Hawblitzel and D. Miller, Eds. Berlin, Germany: Springer, 2012, pp. 126–142.

[8] D. Costanzo, Z. Shao, and R. Gu, "End-to-end verification of information-flow security for C and assembly programs," in *Proc. 37th ACM SIGPLAN Conf. Programm. Lang. Design Implement. (PLDI)*, New York, NY, USA, 2016, pp. 648–664. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908100

[9] T. Murray *et al.*, "seL4: From general purpose to a proof of information flow enforcement," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 415–429.

[10] Y. Zhao, D. Sanan, F. Zhang, and Y. Liu, "Refinement-based specification and security analysis of separation kernels," *IEEE Trans. Depend. Sec. Comput.*, to be published.

[11] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A fast capability system," in *Proc. 17th ACM Symp. Oper. Syst. Principles (SOSP)*, New York, NY, USA, 1999, pp. 170–185. [Online]. Available: http://doi.acm.org/10.1145/319151.319163

[12] J. Woodruff *et al.*, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proc. 41st Annu. Int. Symp. Comput. Archit. (ISCA)*, Piscataway, NJ, USA, 2014, pp. 457–468. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665740

[13] S. Eggert and R. van der Meyden, "Dynamic intransitive noninterference revisited," *CoRR*, vol. abs/1601.05187, pp. 1087–1120, Jan. 2016. [Online]. Available: http://arxiv.org/abs/1601.05187

[14] F. Verbeek *et al.*, "Formal API specification of the PikeOS separation kernel," in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds. Cham, Switzerland: Springer, 2015, pp. 375–389.

[15] R. Gu *et al.*, "CertiKOS: An extensible architecture for building certified concurrent OS kernels," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, Savannah, GA, USA, 2016, pp. 653–669. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[16] M. Krohn *et al.*, "Information flow control for standard OS abstractions," in *Proc. 21st ACM SIGOPS Symp. Oper. Syst. Principles (SOSP)*, New York, NY, USA, 2007, pp. 321–334. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294293

[17] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proc. 7th USENIX Symp. Oper. Syst. Design Implement.*, vol. 7. Berkeley, CA, USA, 2006, p. 19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267327

[18] D. E. Porter, M. D. Bond, I. Roy, K. S. Mckinley, and E. Witchel, "Practical fine-grained information flow control using laminar," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 1, pp. 4:1–4:51, Nov. 2014. [Online]. Available: http://doi.acm.org/10.1145/2638548

[19] M. Krohn and E. Tromer, "Noninterference for a practical DIFC-based operating system," in *Proc. 30th IEEE Symp. Secur. Privacy*, May 2009, pp. 61–76.

[20] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585

[21] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *Proc. 2nd Int. Conf. Interact. Theorem Proving (ITP)*, Berlin, Germany, 2011, pp. 325–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033939.2033965

[22] A. Baumann, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system," in *Proc. Workshop Managed Many-Core Syst.*, Jun. 2008, pp. 1–9. [Online]. Available: https://www.microsoft.com/en-us/research/publication/embracing-diversity-in-the-barrelfish-manycore-operating-system/

[23] R. Leslie, "Dynamic intransitive noninterference," in *Proc. IEEE Int. Symp. Secure Softw. Eng.*, Washington, DC, USA, Mar. 2006.

[24] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI Int., Comput. Sci. Lab., Menlo Park, CA, USA, Tech. Rep. CSL-92-02, 1992.

[25] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1891823.1891830

[26] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Germany: Springer-Verlag, 2002.
[27] D. E. Bell and L. J. L. Padula, "Secure computer system: Unified exposition and multics interpretation," MITRE Corp., Bedford, MA, USA, Tech. Rep. ESD-TR-75-306, Mar. 1976, p. 133.
[28] R. van der Meyden, "What, indeed, is intransitive noninterference?" in *Computer Security—ESORICS*, J. Biskup and J. López, Eds. Berlin, Germany: Springer, 2007, pp. 235–250.
[29] D. von Oheimb, "Information flow control revisited: Noninfluence = noninterference + nonleakage," in *Computer Security—ESORICS*, P. Samarati, P. Ryan, D. Gollmann, and R. Molva, Eds. Berlin, Germany: Springer, 2004, pp. 225–243.
[30] Y. Zhao, D. Sanán, F. Zhang, and Y. Liu, "Reasoning about information flow security of separation kernels with channel-based communication," *CoRR*, vol. abs/1510.05091, pp. 791–810, Oct. 2015. [Online]. Available: http://arxiv.org/abs/1510.05091
[31] P. Efstathopoulos *et al.*, "Labels and event processes in the asbestos operating system," in *Proc. 20th ACM Symp. Oper. Syst. Principles (SOSP)*, New York, NY, USA, 2005, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095813
[32] R. J. Lipton and L. Snyder, "A linear time algorithm for deciding subject security," *J. ACM*, vol. 24, no. 3, pp. 455–464, Jul. 1977. [Online]. Available: http://doi.acm.org/10.1145/322017.322025

JIANWEN SUN is currently pursuing the Ph.D. degree in computer science with the School of Computer Science and Engineering, Beihang University, Beijing, China. His research interests include formal methods, parallel and distributed systems, and information-flow security.

XIANG LONG received the B.S. degree in mathematics from Peking University, China, in 1985, and the M.S. and Ph.D. degrees in computer science from Beihang University, China, in 1988 and 1994, respectively. He has been a Professor with Beihang University since 1999. His research interests include parallel and distributed systems, computer architecture, real-time systems, embedded systems, and multi-/many-core oriented operating systems.

YONGWANG ZHAO received the Ph.D. degree in computer science from Beihang University, Beijing, China, in 2009. He is currently an Associate Professor with the School of Computer Science and Engineering, Beihang University. He has been a Research Fellow with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, since 2015. His research interests include formal methods, OS kernels, information-flow security, and AADL.