Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2016-017 December 15, 2016

Sound and Complete Runtime Security Monitor for Application Software

M. Taimoor Khan , Dimitrios Serpanos, and Howard Shrobe

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Sound and Complete Runtime Security Monitor for Application Software[*]

M. Taimoor Khan[†], Dimitrios Serpanos[‡] and Howard Shrobe[⋆]

[†]muhammad.khan@aau.at
[‡]serpanos@ece.upatras.gr
[*]hes@csail.mit.edu
[†]Software Engineering Research Group
Alpen-Adria University, Klagenfurt, Austria
[‡]ECE, University of Patras and
Industrial Systems Institute/RC ATHENA, Greece
[⋆]MIT CSAIL, USA

December 15, 2016

Abstract

Conventional approaches for ensuring the security of application software at run-time, through monitoring, either produce (high rates of) false alarms (e.g. intrusion detection systems) or limit application performance (e.g. run-time verification). We present a run-time security monitor that detects both known and unknown cyber attacks by checking that the run-time behavior of the application is consistent with the expected behavior modeled in application specification. This is crucial because, even if the implementation is consistent with its specification, the application may still be vulnerable due to flaws in the supporting infrastructure (e.g. the language run-time system, supporting libraries and the operating system). This run-time security monitor is sound and complete, eliminating false alarms, as well as efficient, so that it does not limit run-time application performance and so that it supports real-time systems. Importantly, this monitor is readily applicable to both legacy and new system platforms.

The security monitor takes as input the application specification and the application implementation, which may be expressed in different languages. The specification language of the application software is formalized based on monadic second order logic (i.e. first order logic and set theory) and event calculus interpreted over algebraic data

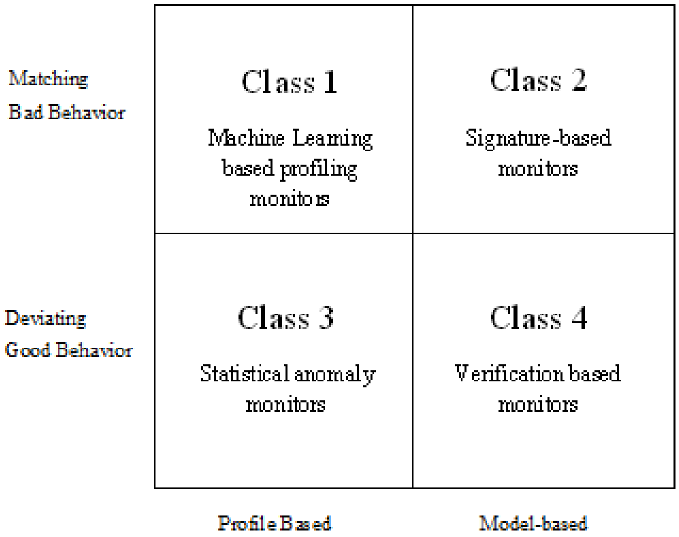
[*]The part of this work was developed at QCRI.

structures. This language allows us to express behavior of an application at any desired (and practical) level of abstraction as well as with high degree of modularity. The security monitor detects every attack by systematically comparing the application execution and specification behaviors at runtime, even though they operate at two different levels of abstraction. We define the denotational semantics of the specification language and prove that the monitor is sound and complete, i.e. if the application is consistent with its specification, the security monitor will produce no false alarms (soundness) and that it will detect any deviation of the application from the behavior sanctioned by the specification language (completeness). Furthermore, the monitor is efficient because of the modular application specification at appropriate level(s) of abstraction. Importantly, the application specification language enables the description of known or potential attack plans, enabling not only attack detection but attack characterization as well and, thus, facilitating effective and efficient defenses to sophisticated attacks.

Contents

1 Introduction

Runtime security monitors are components of defending systems against cyber attacks and must provide fast and accurate detection of attacks. Conventional run-time monitoring systems suffer from high false alarm rates, for both positive and negative alarms, and are inefficient because their typical amount of observed parameters is large and possibly irrelevant to a number of attacks. There are two key reasons for these limitations: first, the systems do not “understand” the complete behavior of the system they are protecting, and second, the systems do not “understand” what an attacker is trying to achieve. Actually, most such systems are retrospective, taking into account and analyzing historical data, resulting to attack surface signatures of previous attacks and attempting to identify the same signature(s) in new traffic. Thus, conventional run-time monitors are passive, waiting for (and expecting that) something similar to what has already happened to recur. Attackers, of course, respond by varying their attacks so as to avoid detection.



	Profile Based	Model-based
Matching Bad Behavior	Class 1 Machine Learning based profiling monitors	Class 2 Signature-based monitors
Deviating Good Behavior	Class 3 Statistical anomaly monitors	Class 4 Verification based monitors

Figure 1: Classification of Runtime Security Monitoring Systems

There are two dimensions along which run-time monitoring systems for security can be classified. The first one is the behavior description method, i.e. profile-based or model-based. The second one is the behavior comparison method, i.e. matching to bad behavior or deviation from good behavior. This classification approach leads to four classes, as shown in Figure 1, which include existing techniques and systems, each with its own strengths and weaknesses. Profile-based systems that detect attacks by matching with bad behavior (Class 1 in the figure) typically employ statistical and machine learning methods to build a profile of bad behavior of the systems and more specifically, build statistical profiles of attacks (e.g., [27, 28]). These systems are more robust than model based systems, since the machine learning techniques tend to generalize from the data presented. However, they do not provide rich diagnostic information and suffer from false alarms. Alternatively, profile-based systems that detect deviation from good behavior (Class 3 in the figure) typically build a statistical profile of normal (good) behavior and detect deviations from this profile (e.g. [25, 26]). Such anomaly detectors are even more robust than Class 1 systems, because they do not depend on historical knowledge of the form of an attack. However, they have a significant false alarm rate, because they have limited diagnostic information: when a deviation is detected, the known information about it is that something out of the ordinary has happened, but there is no sufficient information whether this is malicious, accidental or just a variation of the normal behavior beyond the statistically accepted profile.

Model-based systems (Classes 2 and 4 in Figure 1) are popular in highly secure environments, where successful attacks cause significantly high costs. Signature-based systems are a typical example in this class (e.g. [23, 24]), and they look for matches to bad behavior, i.e. they are systems in Class 2. The advantage of such systems is that, when a match occurs, i.e. an attack is detected, the systems have enough diagnostic information available to "understand" what the failure has been. However, they lack robustness, since they will fail to detect an attack, if they have no model of it; thus, they are susceptible to zero-day attacks and, in general, attacks they have not been trained for. Finally, model-based systems that employ run-time software verification to detect deviation from good behavior fall in Class 4 of the figure. These systems model the good behavior of a system (e.g. [29, 30]) and detect deviations from that behavior using run-time software verification techniques. Their advantage is that, whenever the system execution deviates from good behavior, there is knowledge of the exact problem that led to the deviation (i.e. the offending instruction or routine). However, such verification methods (a) require adequate design/implementation information of the system to operate (which is usually not the case for legacy systems) and (b) limit run-time system performance, with high impact on real-time systems, such as industrial control systems (ICS).

Our run-time security monitor falls in Class 4, because it (a) models normal (good) behavior of the system through a formal specification description and (b) raises an alarm when the behavior of the application's execution deviates from the behavior described in the (executable) specification. Specifically, our security monitor has an active model of normal behavior, namely an executable specification of the application [3]. This executable specification consists of a decomposition into sub-modules and pre-

and post-conditions and invariant for each sub-module. In addition, data-flow and control-flow links connect the sub-modules, specifying the expected flow of values and of control. The pre- and post-conditions and invariant are arbitrary first-order statements about the set of data values (that flow into and out of the sub-modules) and about other arbitrary constraints respectively.

Our run-time security monitor is suitable not only for new systems, which derive application implementation from application specification, but also for "legacy" systems, where application implementations exist without adequate (formal or informal) application specifications. This can be achieved by describing application specification at any feasible level of abstraction through available specification information. Furthermore, modular application specification at any desired level of abstraction also allows us to monitor only attack(s) specific behavior of "real-time" systems without affecting their performance at run-time. As our run-time security monitor is using an executable application specification, it is efficient for use in real-time system as has been proven for real-time safety-critical systems [31].

Our run-time security monitor ("RSM"), shown in Figure 2, is the core component of a larger system named ARMET. ARMET takes as input a specification ("AppSpec") and an implementation ("AppImpl") of the application of interest. Based on the specification, the "Wrapper Synthesizer" of ARMET generates probes to observe the run-time behavior of the application that corresponds to the specification elements. During execution of the "AppImpl", the RSM checks whether the actual behavior of the system (*observations* generated by "Wrapper Synthesizer") is consistent with the *predictions* generated from "AppSpec". If an inconsistency is detected, RSM raises an alarm and ARMET suspends the application execution and proceeds to diagnosis, in order to identify why the execution of "AppImpl" did not behave as predicted. In addition to run-time monitoring, ARMET employs diagnostic reasoning techniques to further isolate and characterize the failure [11]. ARMET is highly robust and has high diagnostic data resolution, which is a key requirement of real-time systems that require continuous operation even after a successful attack. ARMET achieves continuous operation through the construction of a far more complex models of applications.

RSM runs executable application specification in parallel with the actual application code, comparing their results at the granularity and abstraction level of the executable specification. The executable specification is hierarchical and modular, allowing flexibility in the granularity of the monitoring. Depending on the environment, the executable specification may run at a high level of abstraction, incurring less overhead, but requiring more diagnostic reasoning when the program diverges from the behavior of the executable specification. Alternatively, the executable specification can be elaborated in greater detail, incurring more overhead, but providing more

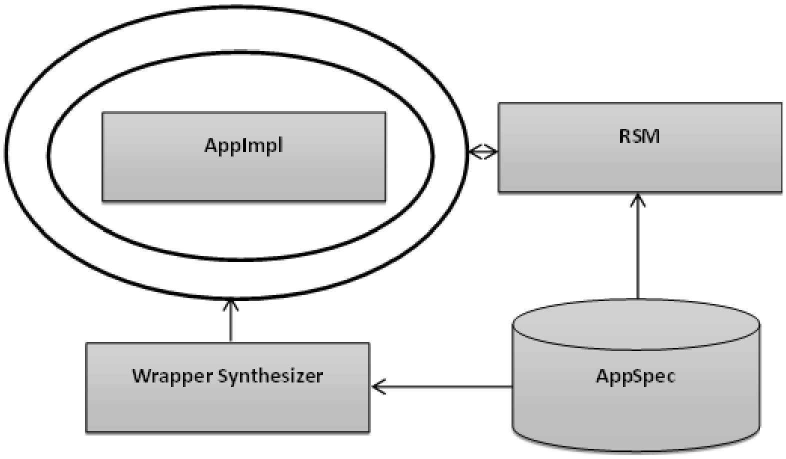


Figure 2: The Architecture of Core Defending-System

containment.

Optionally, the model can also specify suspected incorrect behaviors of a component and associated potential attack plans, allowing the diagnostic reasoning to characterize the way in which a component may have misbehaved. Then, diagnosis is a selection of behavioral modes for each component of the specification, such that the specification predicts the observed misbehavior of the system.

Through this work, we introduce a highly reliable run-time security monitor with proven absence of false alarms (i.e. soundness and completeness). Importantly, the proof establishes a contract between the monitor and its user such that, if the user establishes the *assumptions* of the proof, the monitor *guarantees* to detect any violation at run-time.

The remaining of the report is organized as follows. In Section 2, we describe related work and in Section 3 we present the calculus (syntax and semantics) of the application specification language. In Section 4, we present the calculus (syntax and semantics) of the security monitor. Section 5 and 6 formulate and proves soundness and completeness of the monitor respectively. The auxiliary functions and predicates are discussed in section 7, while section 8 presents auxiliary lemmas and sketches of their corresponding proofs. We conclude in section 9.

2 Related Work

The operation of RSM is to check the consistency between the specified and execution behaviors of an application at run-time. This may be viewed as a run-time verification problem. The goal of run-time verification is to specify the intended behavior of a system in some formalism and to generate an executable monitor from this formalism (i.e. specification) that reports inconsistent execution, if detected.

There has been extensive research on specification based run-time monitoring. Most such approaches employ formalism such as context grammars, regular expressions [13], event calculus [10], temporal logic [7, 6] and rule systems operating over atomic formulas [9]. Such formalism offer limited expressive power to formalize complex system properties, although they can be translated into efficient executable monitors. To addresses the challenges of run-time monitoring of "legacy" and "real-time" systems (namely the lack of design information and performance respectively), our formalism allows not only to specify dependencies, system level behavior and security properties (in case of partial design details), but also to specify internal system behavior and complex security properties (in case of desired design details) of such systems as well.

Run-time monitoring of legacy systems has not received significant attention. However, there have been attempts to apply similar monitoring techniques. For example, Kaiser et al. instrument the systems by probing and passing data to another component that forms a basis of the system's model which is later used to monitor run-time modifications automatically [15]. More recently, Wofgang et al. have automatically generated run-time monitor for network traffic from a high-level specification language which is based on first order predicate logic and set theory [14]. Furthermore, based on a variant of denotational semantics of the specification language and operational semantics of the monitor [17], they verified soundness of the resource analysis of the monitor [16]. The resource analysis identifies the number of instances of the monitor and the number of messages required to detect a violation.

Model-based executable specifications have been rarely used for run-time monitoring of real-time systems [18]. However, Barnett et al. have used ASML as an executable specification language for run-time monitoring [19]. ASML is an extension of ASM, which is based on the formalism of a transition system whose states are first order algebras [4]. There is no formal semantics of ASML, however, the operational semantics of some constructs of ASM has been defined by Hannan et al. [5]. More recently, Choilko et al. have developed a framework for executable specification based run-time monitoring of timed systems [21]. In this work, the formalism of the specification is based on an *extended time interval* which is a pair of a time event and a time interval. The formalism for implementation is based on *timed word* which is a sequence of time events and the goal of the monitor is to check the conformance of an implementation word and the specification trace.

In contrast to the approaches discussed above, the focus of our run-time security monitor is to check consistency of automatically generated *predictions* (conditions) from an executable specification language and run-time *observations* of application execution. The formalism of our specification language is based on monadic second order logic [20] and event calculus interpreted over algebraic data structures. This formalism allows specification of faulty behaviors of a system. Furthermore, the formalism enables description of attack plans, which are exploited by the monitor at run-time for early threat detection against more sophisticated and complex attacks, e.g. advanced persistent threats. Our formalism is similar to Crash Hoare-logic that is used to capture the faulty behavior of a file system [22]. Our formalism allows sound construction (resp. specification) of high-level abstract behavior of a system from low-level abstract behavior(s) using a method analogous to classical set builder. Our security monitor is the first approach in run-time monitoring that formally assures the absence of false alarms and thus is sound and complete. For our proof we use the denotational semantics of the application specification language as described in [2].

3 Application Specification Language

Our executable (application) specification language [3] consists of a *decomposition* of an application behavior into sub-modules and pre- and post-conditions and invariant (*behavioral description*) for each sub-module: in rest of the paper, we use the term system for application behavior. The *decomposition* is further equipped with data-flow and control-flow links that connect the sub-modules, specifying the expected flow of values and of control. The specification also allows to specify potential *attack plan*s for the components based on attack models and associated rules that imply a certain attack model.

In the following subsection, we discuss selected high level syntactic domains and their semantics.

3.1 Syntax

Based on the aforementioned description, syntactically, the specification language (represented by syntactic domain ω) has following three main top level constructs:

1. hierarchical decomposition (ζ) of sub-modules,
2. behavioral description (η) of each sub-module and
3. attack plans (ϵ) of modules/sub-modules.

The simplified grammar of these top level domains is shown in Figure 3.

In the following we briefly discuss the decomposition and attack plans, and will focus more on behavioral description, being core and the only one that is also used in the following sections for semantics and proof.

Application Specification	$\omega ::= ... \ \zeta \ \eta \ \epsilon ...$
Decomposition	$\zeta ::= \alpha \mid (\alpha) \ \zeta$
Behavioral Model	$\eta ::= \beta \mid (\beta) \ \eta$
Attack Plan	$\epsilon ::= \delta \ \rho \mid (\delta \ \rho) \ \epsilon$
	...

Figure 3: Top Level Syntactic Domains of the Language

Decomposition (α)

The hierarchical decomposition α of a component[1] consists of

1. its interface
   - sets of inputs and outputs respectively
   - a set of the resources used by the component (e.g. files, binary code, ports) and a set of sub-components
   - sets of events that allow entry and exit to and from the component respectively
   - a set of events that are allowed to occur during the execution of the component
   - a set of conditional probabilities between the possible modes of the resources and the possible modes of the component and a set of known vulnerabilities occurred to the component
2. and a structural model that is a set of sub-components some of that might be splits or joins of
   - data-flows between linking ports of the sub-components and
   - control-flow links between cases of a branch and a component that will be enabled if that branch is taken

The syntactical domain α is defined in Figure 4.

The elements of α are informally discussed above. Further details of α are out of the scope of this paper.

Behavioral Description (β)

The β describes normal (and optionally various compromised) behavior of a component that includes

- set of inputs and outputs respectively,
- allowable events during the execution in that mode and

[1] The ”component” and ”module/sub-module” are used interchangeably.

α ::= **define-ensemble** CompName

:entry-events	**:auto** | `set`(Evnt)
:exit-events	`set`(Evnt)
:allowable-events	`set`(Evnt)
:inputs	`set`(ObjName)
:outputs	`set`(ObjName)
:components	`set`(Comp)
:controlflows	`set`(CtrlFlow)
:splits	`set`(SpltCF)
:joins	`set`(JoinCF)
:dataflows	`set`(DataFlow)
:resources	`set`(Res)
:resource-mapping	`set`(ResMap)
:model-mappings	`set`(ModMap)
:vulnerabilities	`set`(Vulnrablty)

Figure 4: Syntactic Domain for Decomposition (α)

β ::= **defbehavior-model** (CompName **normal** | **compromised**)

:inputs	`set`(ObjName)
:outputs	`set`(ObjName)
:allowable-events	`set`(Evnt)
:prerequisites	`set`(BehCond)
:postconditions	`set`(BehCond)
:invariant	`set`(BehCond)

Figure 5: Syntactic Domain for Behavioral Description (β)

- preconditions on the inputs, post-conditions and invariant, all of that are first order logical expressions.

The complete syntax of β is defined in Figure 5.

Attack Plan (ϵ)

The attack plan ϵ consists of a description of potential attack models (δ) and the rules (ρ) that imply a certain attack. Syntactically, an attack plan includes

- a set of types of attacks that are being anticipated and the prior probability of each of them,
- a set of effects such that how each attack type can effect mode (normal/compromised) of a resource and

δ ::= **define-attack-model** AtkModName
:attack-types (`set`(AtkType))
:vulnerability-mapping (`set`(AtkVulnrabltyMap))

ρ ::= **defrule** AtkRulName (**:forward**)
if `set`(AtkCond) **then** `set`(AtkCons)

Figure 6: Syntactic Domains of Attack Model (δ) and Rule (ρ)

- a set of rules expressing the conditional probabilities between attack types and resource modes.

The syntactic domains of δ and ρ are defined in Figure 6 resp.

In principle, attack plans are hypothetical attacks based on rules that describe different ways of compromising a component. The monitor exploits such plans to match at run-time and detect any such attack, thus making the monitor more robust.

3.2 Example

To provide an intuitive grounding for these ideas we will consider an example of a simple ICS and of its model in the specification language. The system consists of a water tank, a level sensor and a pump that is capable of either filling or draining the tank. The tank has a natural leakage rate that is proportional to the height of the water column in the tank. The tank is controlled by a PID controller; this is a computational device running a standard (PID) control algorithm that has a simple structure:

The algorithm has two inputs: The *set-point*, i.e. the water level that the tank should maintain and the *sensor value* provide by the level sensor. It has a simple output, the *command*. The algorithm performs the following computations based on the three parameters notated as *Kp*, *Ki* and *Kd* that are used as scaling weights in the algorithm as shown in Figure 9 (a).

1. Calculate the *error*, the difference between the set-point and the sensor value

2. Calculate three terms:

   (a) The *Proportional* term; this is just the error weighted by Kp.

   (b) The *Integral* term; this is a running sum of the *errors* seen so far, weighted by Ki.

   (c) The *Derivative* term; this is a local estimate of rate of change of the sensor value, weighted by Kd.

3. Calculate the sum of the three terms.

4. The value of the sum is the *command* output of the algorithm.

The *command* output of the algorithm is sent to the pump, controlling the rate at which the pump either adds or removes water. The algorithm is “tuned” by the choice of the three parameters Kp, Ki and Kd; when well tuned the system responds quickly to deviations from the set-point with little over-shoot and very small oscillations around the set-point.

Finally, we note that the level sensor can be viewed as (and often is) a computational and communication device that estimates the actual height of the water tank and communicates the estimated height back to the controller.

There are two standard categories of attacks on such a system:

- **False Data Injection Attacks**. These are attacks on the sensor and its communication channel, such that the controller receives a value that is different from the actual level of the tank.

- **Controller Attacks**. These are penetrations to the computer running the control algorithm. For our purposes it is only necessary to consider attacks that overwrite the value of one of Kp, Ki, or Kd. Any such attack, will cause the controller to calculate an incorrect command.

In either case, the end result is that the level in the water tank will not be correctly maintained. In the first case, the controller calculates a correct response to the distorted sensor value. For example, suppose that the attacker is systematically distorting the sensor value to be too low. In that case, the controller will continuously issue commands to the pump to add water to the tank, eventually causing the tank to overflow. In the second case, a change in value of one of the controller parameters will cause the controller to calculate in an incorrect command. This can have a variety of effects, depending on which parameters are changed.

Monitoring of such a system requires its behavioral specification as shown in Figure 9 (b). The actual system is a *cyber-physical* system, containing both physical components (i.e. the tank, the pump) and computational components (i.e. the controller and the sensor). The monitor model parallels this structure; it contains computational models of the controller and the sensor as well as a computational model of the physical plant. This later model performs a numerical integration of the differential equations describing the physical plant’s behavior, e.g. the dynamics of the pump. The application specification of the controller, essentially mirrors the structure of the algorithm: There is a component that calculates the error term, data-flow links that connect the error term to each of three parallel steps that calculate the Proporational, Integral and Derivative terms, finally there is the summation component that adds the three terms, calculating the command output.

```
(define-component-type controller-step
 :entry-events (controller-step)
 :exit-events (controller-step)
 :allowable-events (update-state accum-error)
 :inputs (set-point sens-val)
 :outputs (com)

  :components
((err-comp :type err-comp :models (normal))
(comp-der :type comp-der :models (normal)) ... )

  :dataflows
((set-point controller-step set-point err-comp)
(the-error err-comp the-error comp-der)...))
```

Figure 7: Decomposition of the Module `controller-step`

The structural model of the controller is shown diagrammatically in Figure 9 (b) (N and C refers to normal and compromised behavior and A refers to possible attacks). The models for the components of the controller are reasonably straightforward. For example, the normal behavioral model for the Kd calculation states that the output of the component is the derivative of the error, weighted by Kd. This is expressed as a post-condition, as shown in Figure 8.

Notice that what the controller calculates is a discrete approximation of the derivative of the error term, which is calculated using the previous and current versions of the error. The value of the error term is conceptually a state variable that is updated between successive iterations of the controller computation. In our specification language, however, we model these as extra inputs and data flows (as we do also for control algorithm parameters such as Kd). For simplicity, we have omitted these extra items from the diagram in Figure 9.

The compromised behavioral model states that any other behavior is acceptable; it does so by stating no post-conditions.

The run-time behavior of the monitor will depend on the strength of the post-conditions; if these are too weak, the monitor may allow undesired behaviors..

3.3 Formal Semantics

In this section, we first give the definition of semantic algebras, then discuss informal description and the formal denotational semantics of the core construct (i.e. behavioral description) of the specification language.

```
(define-component-type comp-der
 :entry-events (compute-derivative)
 :exit-events (compute-derivative)
 :inputs (the-error old-error kd time-step)
 :outputs (der-term)
 :behavior-modes (normal compromised) )

(defbehavior-model (comp-der normal)
 :inputs (the-error the-old-error kd time-step)
 :outputs (der-term)
 :prerequisites ([data-type-of the-error number])
 :post-conditions
  ([and [data-type-of der-term number]
   [equal der-term
    (*kd(/(- new-error old-error) time-step))]]))

 (defbehavior-model (comp-der compromised)
 :inputs (the-error the-old-error kd time-step)
 :outputs (der-term)
 :prerequisites ()
 :post-conditions ())
```

Figure 8: Normal and Compromised Behavior of `comp-der`(kd)

3.3.1 Semantic Algebras

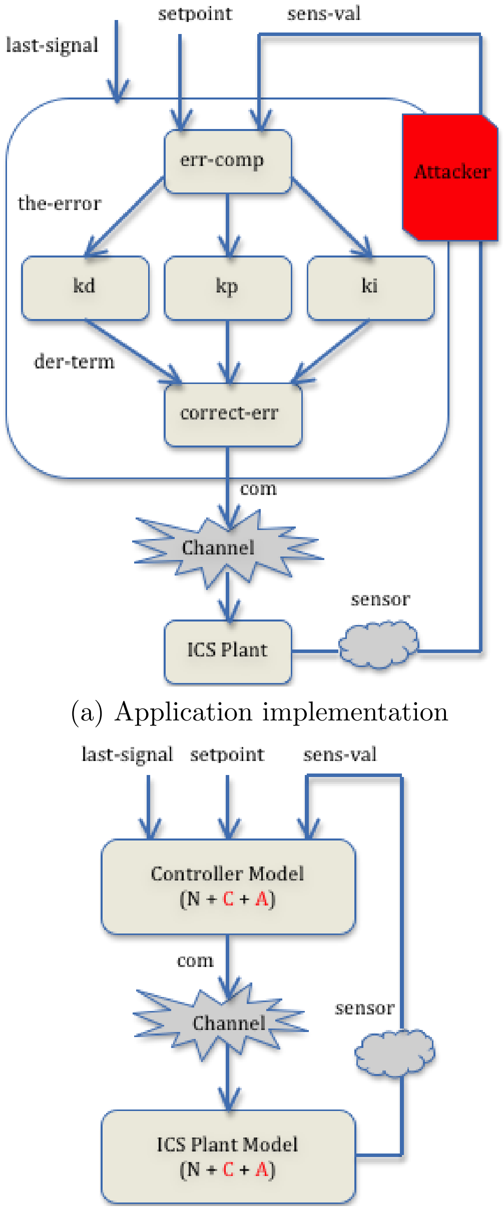
Semantic domains[1] [2] represent a set of elements that share some common properties. A semantic domain is accompanied by a set of operations as functions over the domain. A domain and its operations together form a *semantic algebra* [8]. The domains of our language are similar to the domains of any classical programming/specification language (e.g. Java, JML, ACSL). In the following we declare/define only important semantic domains and their operations.

Environment Values

The domain *Environment* holds the environment values of the language and is formalized as a tuple of domains *Context* (which is a mapping of identifiers to the environment values) and *Space* (that models the memory space). The *Environment* domain includes interesting values, e.g. component, attack plan and resource. Here resource can be binary code in memory, files and

[1] These domains are common to a program to be monitored, its specification language and the monitor.

[2] We use subscript s and r to specify domains for specification and program's runtime resp., e.g. State_s = specification state, State_r = program's runtime state, State = combined monitor state.



(a) Application implementation

(b) Application specification

Figure 9: A Controller Application and its Model

ports etc.
Domain: *Environment*
$Environment := Context \times Space$
$Context := Identifier \rightarrow EnvValue$
$EnvValue := Variable + Component + AtkPlan + Resource + ...$
$Space := \mathbb{P}(Variable)$
Variable := n, where n $\in \mathbb{N}$ represents locations

The domain *Environment* supports typical selection, update and equality operations over its values.

State Values

The domain *State* represents the execution of a program. A *Store* is important element of the state and holds for every *Variable* a *Value*. The *Data* of the state is a tuple of a *Flag* that represents the current status of the state and a *Mode* to represent the current mode of execution of the state of a component.
Domain: *State*
$State := Store \times Data$
$Store := Variable \rightarrow Value$
$Data := Flag \times Mode$
Flag := {running, ready, completed}
Mode := {normal, compromised}

The domain *State* has typical operations, e.g. read and write/update of values, checking equality of *Flag* and *Mode* in a given state, and setting a certain *Flag* and *Mode* of a given state.

Semantic Values

Value is a disjunctive union domain and note that the domain *Value* is a recursive domain.
Domain: *Value*

$$Value := ObsEvent + RTEvent + Component + AtkPlan + ... + Value^*$$

The domain includes semantic values of observable event, a run-time event and attack plan etc. The equality of the given two semantic values can be evaluated.

Component Values

The *Component* formalizes the semantic model of a component as a predicate over decomposition, normal and compromised behavior and a pre-state and a post-state of the component's execution respectively. The predicate is formalized as follows:

$$Component = \mathbb{P}(SBehavior \times NBehavior \times CBehavior \times State \times State_{\perp}{}^{3})$$

where

$$\begin{aligned}&SBehavior := \mathbb{P}(\,Value^{*} \times Value^{*} \times Value^{*} \times State \times State_{\perp})\\&NBehavior = CBehavior := \mathbb{P}(\,Value^{*} \times Value^{*} \times State \times State_{\perp})\end{aligned}$$

Furthermore, *SBehavior* is defined as a predicate over sets of input and output values, set of allowable values, a pre-state and a post-state of the behavior. Also, normal behavior and compromised behavior (*NBehavior* and *CBehavior*) are also defined as predicates over sets of input and output values, a pre-state and a corresponding post-state respectively.

Attack Values

The semantics domain *AtkModel* formalizes the attack model and is defined as a predicate over an attack name, probability of the attack and the corresponding vulnerability causing the attack; the attack model is formulated as follows:

$$AtkModel := \mathbb{P}(Identifier \times FVal \times Vulnerability)$$

3.3.2 Signatures of Valuation Functions

A valuation function defines a mapping of a language's abstract syntax structures to its corresponding meanings (semantic algebras) [8]. The valuation function operates on a syntactic construct and returns a function from the environment to a semantic domain.

We define the result of the valuation function as a predicate, e.g. the behavioral relation (BehRelation) is defined as a predicate over an environment, a pre- and a post-state and is defined as follows:

$$\text{BehRelation} := \mathbb{P}(\text{Environment} \times \text{State} \times \text{State}_{\perp})$$

The valuation functions for the abstract syntax domains of specification (ω), behavioral description (β) and attack plans (ϵ) have same signatures. For example, a valuation function signature for β is defined as follows:

$$[\![\beta]\!]\text{: Environment} \rightarrow \text{BehRelation}$$

Based on the above relation and the auxiliary semantic inference rules (see Figure 11), we define valuation functions for β and ϵ in the following subsection.

[3] $State_{\perp} = State \cup \{\perp\}$

3.3.3 Definition of Valuation Functions

Semantically, normal and compromised behavioral models results in modifying the corresponding elements of the environment value *Component* as defined below:

```
$[\![\beta]\!](\mathrm{e})(\mathrm{e'}, \mathrm{s}, \mathrm{s'}) \Leftrightarrow$
 $\mathrm{LET\ c} \in \mathrm{Component}\text{: } [\![\mathrm{CompName}]\!](\mathrm{e})(\mathrm{s}, \mathrm{s'}, \mathrm{inValue(c)})\ \mathrm{IN}$
 $\forall\ \mathrm{e}_1 \in \mathrm{Environment}, \mathrm{nseq} \in \mathtt{set}(\mathrm{EvntName}), \mathrm{b}_1, \mathrm{b}_2\text{: } \mathbb{B},$
   $\mathrm{eseq} \in \mathrm{ObsEvent}^*, \mathrm{iseq}, \mathrm{oseq} \in \mathrm{Value}^*\text{:}$
   $[\![\mathtt{set}(\mathrm{ObjName}_1)]\!](\mathrm{e})(\mathrm{s}, \mathrm{iseq}) \wedge [\![\mathtt{set}(\mathrm{BehCond}_1)]\!](\mathrm{e})\ (\mathrm{s}) \wedge$
   $\mathtt{noatk}(\mathrm{c}, \mathrm{e}, \mathrm{b}_1) \wedge [\![\mathtt{set}(\mathrm{Evnt})]\!](\mathrm{e})\ (\mathrm{e'}, \mathrm{s}, \mathrm{s'}, \mathrm{nseq}, \mathrm{eseq}) \wedge$
   $[\![\mathtt{set}(\mathrm{ObjName}_2)]\!](\mathrm{e'})(\mathrm{s'}, \mathrm{oseq}) \wedge [\![\mathtt{set}(\mathrm{BehCond}_2)]\!](\mathrm{e'})(\mathrm{s},\mathrm{s'}) \wedge$
   $[\![\mathtt{set}(\mathrm{BehCond}_3)]\!](\mathrm{e'})\ (\mathrm{s}, \mathrm{s'}) \wedge \mathtt{noatk}(\mathrm{c}, \mathrm{e'}, \mathrm{b}_2)$
   $\Rightarrow$
   $\mathrm{LET\ v} = \mathrm{b}_1 \wedge \mathrm{b}_2 \wedge \mathrm{eqMode}(\mathrm{s'}, \text{”normal”})\ \mathrm{IN}$
         $\mathtt{update}(\mathrm{c}, \mathrm{e'}, \mathrm{s}, \mathrm{s'}, \mathrm{iseq}, \mathrm{oseq}, \mathrm{v})$
```

where `update` is an auxiliary semantic rule as shown in Figure 11.

In detail, if the semantics of β in an environment e yields environment e' and transforms a pre-state s into a post-state s' then

- the evaluation of inputs `set`(ObjName$_1$) yields a set of values *iseq* in environment e and state s such that the pre-conditions `set`(BehCond$_1$) hold in e and s and the component c has no potential threat (see rule `noatk`) and
- the evaluation of allowable events results in environment e' and given post-state s' with some auxiliary sets *nseq* and *eseq* and
- the evaluation of outputs `set`(ObjName$_2$) yields a set of values *oseq* in e' and s' such that post-conditions `set`(BehCond$_2$) hold in e_1, s and s' and
- the invariant `set`(BehCond$_2$) holds in e', s and s', and the component c has no threat (`noatk`), finally the environment e' can be constructed as follows
  - if the post-state is “normal” then e' is an update to the normal behavior “nbeh” of the component “CompName”
  - otherwise e' is an update to the compromised behavior “cbeh” of the component as shown in the corresponding inference rules of `update`.

Moreover, the valuation function for attack plan is defined as:

$$\frac{t \in \{\texttt{ENTRY}, \texttt{EXIT}, \texttt{ALLOWABLE}, \texttt{NONE}\}}{\texttt{typeOf}(oe, c) \to t}$$

$$\frac{\begin{array}{c}\text{dataArrives}(c, s(i), s'(i)) \qquad \texttt{comp}(c, e(i), e'(i), s(i), s'(i), \texttt{False}, 0) \\ s(\text{i+1}) = s(i) \qquad s'(\text{i+1}) = s(i) \qquad \text{setMode}(s'(\text{i+1}), \text{”compromised”})\end{array}}{\texttt{run}(\texttt{ENTRY}, c, e, e', s, s', i, \texttt{False})}$$

$$\frac{\begin{array}{c}\text{dataArrives}(c, s(i), s'(i)) \quad \texttt{comp}(c, e(i), e'(i), s(i), s'(i), \texttt{True}, 0) \quad \text{setFlag}(s'(\text{i+1}), \text{”running”}) \\ s(\text{i+1}) = s'(i) \quad e(\text{i+1}) = e'(i) \quad \texttt{mon}(\mathit{components}(c), s(\text{i+1}), s'(\text{i+1}), e(\text{i+1}), e'(\text{i+1}))\end{array}}{\texttt{run}(\texttt{ENTRY}, c, e, e', s, s', i, \texttt{True})}$$

$$\frac{\begin{array}{c}\text{dataArrives}(c, s(i), s'(i)) \quad \text{setFlag}(s'(\text{i+1}), \text{”completed”}) \quad \texttt{comp}(c, e(i), e'(i), s(i), s'(i), b, 1) \\ s(\text{i+1}) = s'(i) \quad s'(\text{i+1}) = s'(i) \quad [b = \texttt{False} \Rightarrow \text{setMode}(s'(\text{i+1}), \text{”compromised”})] \\ [b = \texttt{True} \Rightarrow \text{setMode}(s'(\text{i+1}), \text{”normal”})]\end{array}}{\texttt{run}(\texttt{EXIT}, c, e, e', s, s', i, b)}$$

$$\frac{\text{inv}(c, e(i), e'(i), s(i), s'(i), b_1) \quad \text{noatk}(c, e(i), b_1) \quad s(\text{i+1}) = s'(i) \quad e(\text{i+1}) = e'(\text{i+1})}{\texttt{run}(\texttt{ALLOWABLE}, c, e, e', s, s', i, b_1 \wedge b_2)}$$

$$\frac{\text{setMode}(s'(\text{i}), \text{”compromised”} \qquad s(\text{i+1}) = s'(i) \qquad e'(\text{i+1}) = e(\text{i+1})}{\texttt{run}(\texttt{NONE}, c, e, e', s, s', i, \texttt{Flase})}$$

$$\frac{\mathit{nbeh} = \langle \mathit{inseq}, \mathit{outseq}, s, s' \rangle \qquad c' = \langle c[1], \mathit{nbeh}, c[3], s, s' \rangle}{\texttt{update}(c, e_1[\mathit{id}(c) \mapsto c'], s, s', \mathit{inseq}, \mathit{outseq}, \texttt{True})}$$

$$\frac{\mathit{cbeh} = \langle \mathit{inseq}, \mathit{outseq}, s, s' \rangle \qquad c' = \langle c[1], c[2], \mathit{cbeh}, s, s' \rangle}{\texttt{update}(c, e_1[\mathit{id}(c) \mapsto c'], s, s', \mathit{inseq}, \mathit{outseq}, \texttt{False})}$$

$$\frac{a = \langle \mathit{aseq}, \mathit{apseq}, \mathit{vnseq} \rangle}{\texttt{atk}(atkName, e, e[atkName \mapsto a], \mathit{aseq}, \mathit{apseq}, \mathit{vnseq})}$$

$$\frac{b = [\forall at : \text{AtkName} : at = \text{context}(e)(\text{AtkName}) \Rightarrow \text{notcomp}(c, at)]}{\texttt{noatk}(c, e, b)}$$

Figure 10: Auxiliary Semantic Inference Rules (A)

$$\frac{\begin{array}{c}\text{inv}(c, e(i), e'(i), s(i), s'(i), b_1) \quad b_2 = [x = 0 \Rightarrow \text{precond}(c, e(i), e'(i), s(i), s'(i), \texttt{True})] \\ b_3 = [x = 1 \Rightarrow \text{postcond}(c, e(i), e'(i), s(i), s'(i), \texttt{True})] \qquad \texttt{noatk}(c, e(i), b_4)\end{array}}{\texttt{comp}(c, s, s', e, e', b_1 \wedge b_2 \wedge b_3 \wedge b_4, x)}$$

$$\frac{\begin{array}{c}\text{inv}(c, e(i), e'(i), s(i), s'(i), \texttt{False}) \\ \vee \qquad x = 0 \Rightarrow \text{precond}(c, e(i), e'(i), s(i), s'(i), \texttt{False}) \qquad \vee \\ x = 1 \Rightarrow \text{post}(c, e(i), e'(i), s(i), s'(i), \texttt{False}) \qquad \vee \qquad \text{noatk}(c, e(i), \texttt{False})\end{array}}{\texttt{comp}(c, s, s', e, e', \texttt{False}, x)}$$

$$\frac{\begin{array}{c}\exists rte \qquad \text{arrives}(rte, s) \\ \text{monitors}(\text{i+1}, rte, c, e, e'', s, s'') \qquad \texttt{mon}(cseq, s'', s', e'', e', s'', s', i)\end{array}}{\texttt{mon}(c; cseq, s, s', e, e', s, s', i)}$$

Figure 11: Auxiliary Semantic Inference Rules (B)

$\llbracket\delta\rrbracket$(e)(e’, s, s’) $\Leftrightarrow$
$\forall$ s” $\in$ State, aseq, aseq’, vnseq $\in$ ISeq, apseq $\in$ Value*:
$\llbracket$`set`(AtkType)$\rrbracket$(e)(s, inState$_\perp$(s”), aseq, apseq) $\wedge$
$\llbracket$`set`(AtkVulnrabltyMap)$\rrbracket$(e) (s”, s’, aseq’, vnseq) $\wedge$
`atk`(AtkModName, e, e’, aseq, apseq, vnseq)

In detail, the semantics of the domain “δ” updates the environment e with a semantic value of *AtkPlan* such that if

- in a given e and s, the evaluation of “`set`(AtkType)” yields post-state s'', a set of attack types *aseq* and a set of values (conditional probabilities) *apseq* and also
- in given e and s, the evaluation of “`set`(AtkVulnrabltyMap)” yields post-state s', a set of attack types *aseq*$'$ and a set of vulnerabilities *vnseq*, then
- the environment e' is an update of environment e with the semantic value *AtkPlan*, which is a triple of (a) a set of attack types (b) a set of corresponding probabilities and (c) a set of vulnerabilities causing the attack types, respectively.

4 Security Monitor

Based on [3], in this section we discuss the informal behavior of our runtime security monitor whose main goal is to check consistency between a program's run-time *observations* and its specification-based *predictions* and to only raise a flag if any inconsistency is identified. In detail, when the application implementation starts execution, a “startup” event is generated and dispatched to the top level component of the system, which transforms the execution state of the component into “running” mode. The component instantiates its subnetwork (i.e. sub-components) and propagates the data along its data-links by enabling the corresponding control-links (if involved). When the data arrives on the input port of the component, the monitor checks if it is complete; if so, the monitor checks the preconditions of the component for the data and if they succeed, it transforms the state of the component into “ready” mode. Should the conditions fail, it raises a flag.

After the above startup, the execution monitor starts monitoring the arrival of every *observation* (run-time event) as follows:

1. If the event is a “method entry”, then the execution monitor checks if this is one of the “entry events” of the component in the “ready” state; if so, then after receiving the data, the respective preconditions, invariant and absence of attack plans are checked; if they succeed, then the data is applied on the input port of the component and the mode of the execution state is changed to “running”.

2. If the event is a “method exit”, then the execution monitor checks if this is one of the “exit events” of the component in the “running” state; if so, it changes its state into “completed” mode and collects the data from the output port of the component and checks for the corresponding postconditions, invariant and absence of attack plans. Should the checks fail, the monitor raises an alarm.

3. If the event is one of the “allowable events” of the component, if invariant holds and there is no attack plan then it continues execution and finally

4. otherwise, if the event is an none of the above events, then the monitor raises an alarm.

4.1 Formal Semantics

Based on the aforementioned description of the execution monitor, we have formalized the denotational semantics of the monitor by a relation *monitor* that is declare and defined as follows:

$\mathbf{monitor} \subseteq \mathbf{AppImpl} \times \mathbf{AppSpec} \rightarrow \mathbf{Environment} \rightarrow \mathbf{State} \times \mathbf{State}_\perp$
$\text{monitor}(\kappa, \omega)(\text{e})(\text{s, s'}) \Leftrightarrow$
$\forall$ c $\in$ Component, t, t' $\in$ State_s, d, d' $\in$ Environment_s, rte $\in$ RTEvent:
$[\![\omega]\!](\text{d})(\text{d', t, t'}) \wedge [\![\kappa]\!](\text{e}_r)(\text{e}_r\text{', s, s'}) \wedge$ setFlag(s, “running”) $\wedge$
eqMode(s, ”normal”) $\wedge$ arrives(rte, s) $\wedge$ equals(s, t) $\wedge$ equals(e_r, d)

```
$\Rightarrow$
  $\forall$ p, p' $\in$ Environment*, m, n $\in$ State*:
    equals(m(0), s) $\wedge$ equals(p(0), $e_r$) $\wedge$
    $\exists$ k $\in \mathbb{N}$:
      ( $\forall$ i $\in \mathbb{N}_k$: monitors(i, rte, c, p, p', m, n) $\wedge$ equals(s', n(k)) ) $\wedge$
      [( eqMode(n(k), "normal") $\vee$ eqMode(n(k), "compromised")] $\wedge$
      IF eqMode(n(k), "normal") THEN
         eqFlag(n(k), "completed") $\wedge$ equals(s', t')
      ELSE $\neg$ equals(s', t')
```

In detail, the predicate says that if we execute specification (ω) in an arbitrary safe pre-state (s) and environment (d) and execute program (κ) in an arbitrary pre-state (t s.t. s equals t) and environment (e_r s.t. e_r equals d) then there is a finite natural number (k) at which monitor can be observed such that for all iterations until k, the monitor continuous operation. However, at iteration k, either the monitor is in a "normal" mode or in a "compromised" mode. If the mode is "normal", then the component under monitoring has finished its job safely and the post-state of the program execution (t') is equal to post-state (t) of the specification execution, otherwise component is compromised and thus the program execution state (s') and specification execution state (t') are inconsistent. The core semantics of *monitor* is captured by an auxiliary predicate *monitors* that is defined as a relation on

- the number of observation i w.r.t. of a component,
- an observation (run-time event) *rte*, component c being observed,
- sets of pre- and post-environments e and e' resp. and
- sets of pre- and post-states s and s' respectively.

The predicate *monitors* is formalized as follows:

```
$\mathbf{monitors} \subseteq \mathbb{N} \times \mathbf{RTEvent} \times \mathbf{Component}$
                  $\times\, \mathbf{Environment}^* \times \mathbf{Environment}^*$
                  $\times\, \mathbf{State}^* \times \mathbf{State}^*_\perp$
monitors(i, $[\![$rte$]\!]$, $[\![$c$]\!]$, e, e', s, s') $\Leftrightarrow$
eqMode(s(i), "completed")
$\vee$
[ ( eqMode(s(i), "running") $\vee$ eqMode(s(i), "ready") ) $\wedge$
   $\neg$ eqMode(s(i), "compromised") $\wedge$ $[\![$c$]\!]$(e(i))(e'(i), s(i), s'(i)) $\wedge$
   $\exists$ oe $\in$ ObEvent: equals(rte, store($[\![$name(rte)$]\!]$)(e(i))) $\wedge$
      run(type(oe, c), c, e, e', s, s', i, eqMode(s', "normal"))) ]
```

In detail, the predicate *monitors* is defined such that, at any arbitrary observation either the execution is completed and returns or the current execution state $s(i)$ of component c is "ready" or "running" and the current execution state is safe and behavior of the component c has been evaluated and there is a run-time event *oe* that we want to observe (and thus equals an *observation rte*) and then any of the following can happen:

- either the *prediction* resp. *observation* is an entry event of the component c, then it waits until the complete data for c arrives, if so, then
  - either the preconditions and the invariant of "normal" behavior of the component hold and there is no potential attack for the component (as modeled by semantic rule `comp` in Figure 11); if so, then the subnetwork of the component is initiated and the subcomponents in the subnetwork are monitored iteratively with the corresponding arrival of the *observation*
  - or the preconditions and the invariant of "compromised" behavior of the component hold or some attack plan is detected for the component, in this case the state is marked to "compromised" and returns
- or the *observation* is an exit event and after the arrival of complete data, the post-conditions and the invariant hold and if there is no potential threat detected, then the resulting state is marked as "completed"
- or the *observation* is an allowable event, the invariant holds and there is no threat for c, then the c continues the execution
- or the *observation* is an unexpected event (i.e. none of the above holds), then the state is marked as "compromised" and returns.

All of the above choices are modeled by the corresponding semantic inference rule of `run`, see Figure 11.

5 Proof of the Soundness

The intent of soundness statement is to articulate whether the system's behavior is consistent with the behavioral specification. Essentially, the goal here is to show the absence of false negative alarm such that whenever the security monitor alarms there is indeed a semantic inconsistency between post-state of the program execution and post-state of the specification execution. The soundness theorem is stated as follows:

Theorem 1 (Soundness of security monitor) *The result of the security monitor is sound for any execution of the target system and its specification, iff, the specification is consistent*[4] *with the program and the program*

[4] See definition of the corresponding predicate *consistent* in 7.

executes in a safe pre-state and in an environment that is consistent with the environment of the specification, then

- *for the pre-state of the program, there is an equivalent safe pre-state for which the specification can be executed and the monitor can be observed and*
- *if we execute the specification in an equivalent safe pre-state and observe the monitor at any arbitrary (combined) post-state, then*
  - *either there is no alarm, and then the post-state is safe and the program execution (post-state) is semantically consistent with the specification execution (post-state)*
  - *or there is an alarm, and then the post-state is compromised and the program execution (post-state) and the specification execution (post-state) are semantically inconsistent.*

Formally, soundness theorem has the following signatures and definition.

$$\begin{array}{l}
\text{Soundness\_ad} \subseteq \mathbb{P}(\text{AppImpl} \times \text{AppSpec} \times \text{Bool}) \\
\text{Soundness\_ad}(\kappa, \omega, \text{b}) \Leftrightarrow \\
\forall\ \text{e}_s \in \text{Environment}_s,\ \text{e}_r, \text{e}_r\text{'} \in \text{Environment}_r,\ \text{s, s'} \in \text{State}_r\text{:} \\
\quad \text{consistent}(\text{e}_s, \text{e}_r) \wedge \text{consistent}(\kappa, \omega) \wedge \\
\quad [\![\kappa]\!](\text{e}_r)(\text{e}_r\text{', s, s'}) \wedge \text{eqMode(s, ”normal”)} \\
\Rightarrow \\
\quad \exists\ \text{t, t'} \in \text{State}_s,\ \text{e}_s\text{'} \in \text{Environment}_s\text{:} \\
\qquad \text{equals(s, t)} \wedge [\![\omega]\!](\text{e}_s)(\text{e}_s\text{', t, t'}) \wedge \text{monitor}(\kappa, \omega)(\text{e}_r\text{;e}_s)(\text{s;t, s';t'}) \wedge \\
\quad \forall\ \text{t, t'} \in \text{State}_s,\ \text{e}_s\text{'} \in \text{Environment}_s\text{:} \\
\qquad \text{equals(s, t)} \wedge [\![\omega]\!](\text{e}_s)(\text{e}_s\text{', t, t'}) \wedge \text{monitor}(\kappa, \omega)(\text{e}_r\text{;e}_s)(\text{s;t, s';t'}) \\
\quad \Rightarrow \\
\quad \text{LET b = eqMode(s', ”normal”) IN} \\
\qquad\qquad \text{IF b = } \texttt{True} \text{ THEN equals(s', t') ELSE } \neg \text{ equals(s', t')} \quad \text{(G)}
\end{array}$$

In detail, the soundness statement says that, if

1. a specification environment (e_s) is *consistent* with a run-time environment (e_r) and
2. a target system (κ) is *consistent* with its specification (ω) and
3. in a given run-time environment (e_r), execution of the system (κ) transforms pre-state (s) into a post-state (s’) and
4. the pre-state (s) is safe, i.e. the state is in ”normal” mode,

then

- there is such pre- and post-states (t and t' respectively) and environment (e_s') of the specification execution such that in a given specification environment (e_s), execution of the specification (ω) transforms pre-state (t) into a post-state (t') and
- the pre-states s and t are *equal* and *monitor*ing of the system (κ) transforms combined pre-state (s;t) into a combined post-state (s';t') and if
- in a given specification environment (e_s), execution of the specification (ω) transforms pre-state (t) into a post-state (t') and
- the pre-states s and t are *equal* and *monitor*ing of the system (κ) transforms pre-state (s) into a post-state (s') then
  - either there is no alarm (b is `True`) and then the post-state s' of program execution is safe and the resulting states s' and t' are semantically *equal*
  - or the security monitor alarms (b is `False`) and then the post-state s' of program execution is compromised and the resulting states s' and t' are semantically not *equal*.

In the following section we present proof of the soundness statement.

5.1 Proof

The proof is essentially a structural induction on the elements of the specification (ω) of the system (κ). We have proved only interesting case β of the specification to show that the proof works in principle. However, the proof of the remaining parts can easily be rehearsed following the similar approach.

The proof is based on certain lemmas (see subsection 8), which are about the relations between different elements of the system and its specification (being at different levels of abstraction). These lemmas and relations can be proved based on the defined auxiliary functions and predicates (see subsection 7) that are based on the method suggested by Hoare [1].

In the following, we start proof with induction on η.

5.1.1 Case (η)

We can re-write (G) as

Soundness_ad(κ, η, b) $\Leftrightarrow$
$\forall$ e_s $\in$ Environment_s, e_r, e_r' $\in$ Environment_r, s, s' $\in$ State_r:
consistent(e_s, e_r) $\wedge$ consistent(κ, η) $\wedge$
$[\![\kappa]\!]$(e_r)(e_r', s, s') $\wedge$ eqMode(s, "normal")

$$
\begin{array}{l}
\Rightarrow \\
\quad \exists\ \text{t, t'} \in \text{State}_s,\ \text{e}_s\text{'} \in \text{Environment}_s\text{:} \\
\qquad \text{equals(s, t)} \wedge [\![\eta]\!](\text{e}_s)(\text{e}_s\text{', t, t')} \wedge \\
\qquad \text{monitor}(\kappa, \eta)(\text{e}_r\text{;e}_s)\text{(s;t, s';t')} \wedge \\
\quad \forall\ \text{t, t'} \in \text{State}_s,\ \text{e}_s\text{'} \in \text{Environment}_s\text{:} \\
\qquad \text{equals(s, t)} \wedge [\![\eta]\!](\text{e}_s)(\text{e}_s\text{', t, t')} \wedge \\
\qquad \text{monitor}(\kappa, \eta)(\text{e}_r\text{;e}_s)\text{(s;t, s';t')} \\
\quad \Rightarrow \\
\quad \text{LET b} = \text{eqMode(s', ”normal”) IN} \\
\qquad\qquad \text{IF b} = \texttt{True}\ \text{THEN equals(s', t')} \\
\qquad\qquad \text{ELSE} \neg \text{ equals(s', t')} \qquad\qquad\qquad \text{(G-1)}
\end{array}
$$

Here, we have two syntactic cases for η but we will show only one case in the following subsection.

5.1.2 Case when $\eta = \beta$

We can re-write (G-1) as

$$
\begin{array}{l}
\text{Soundness\_ad}(\kappa, \beta, \text{b}) \Leftrightarrow \\
\forall\ \text{e}_s \in \text{Environment}_s,\ \text{e}_r,\ \text{e}_r\text{'} \in \text{Environment}_r,\ \text{s, s'} \in \text{State}_r\text{:} \\
\quad \text{consistent}(\text{e}_s, \text{e}_r) \wedge \text{consistent}(\kappa, \beta) \wedge \\
\quad [\![\kappa]\!](\text{e}_r)(\text{e}_r\text{', s, s')} \wedge \text{eqMode(s, ”normal”)} \\
\Rightarrow \\
\quad \exists\ \text{t, t'} \in \text{State}_s,\ \text{e}_s\text{'} \in \text{Environment}_s\text{:} \\
\qquad \text{equals(s, t)} \wedge [\![\beta]\!](\text{e}_s)(\text{e}_s\text{', t, t')} \wedge \\
\qquad \text{monitor}(\kappa, \beta)(\text{e}_r\text{;e}_s)\text{(s;t, s';t')} \wedge \\
\quad \forall\ \text{t, t'} \in \text{State}_s,\ \text{e}_s\text{'} \in \text{Environment}_s\text{:} \\
\qquad \text{equals(s, t)} \wedge [\![\beta]\!](\text{e}_s)(\text{e}_s\text{', t, t')} \wedge \\
\qquad \text{monitor}(\kappa, \beta)(\text{e}_r\text{;e}_s)\text{(s;t, s';t')} \\
\quad \Rightarrow \\
\quad \text{LET b} = \text{eqMode(s', ”normal”) IN} \\
\qquad\qquad \text{IF b} = \texttt{True}\ \text{THEN equals(s', t')} \\
\qquad\qquad \text{ELSE} \neg \text{ equals(s', t')} \qquad\qquad\qquad \text{(F.1)}
\end{array}
$$

From (F.1), we know

- consistent(e_s, e_r) (1)
- consistent(κ, β) (2)
- $[\![\kappa]\!](\text{e}_r)(\text{e}_r\text{', s, s')}$ (3)
- eqMode(s, ”normal”) (4)

We show

- $\exists\, t, t' \in \text{State}_s, e_s' \in \text{Environment}_s\text{: equals}(s, t) \wedge$
  $\llbracket\beta\rrbracket(e_s)(e_s', t, t') \wedge \text{monitor}(\kappa, \beta)(e_r;e_s)(s;t, s';t')$ (G-1.1)

- $\forall\, t, t' \in \text{State}_s, e_s' \in \text{Environment}_s\text{: equals}(s, t) \wedge$
  $\llbracket\beta\rrbracket(e_s)(e_s', t, t') \wedge \text{monitor}(\kappa, \beta)(e_r;e_s)(s;t, s';t')$
  $\Rightarrow$
  LET $b = \text{eqMode}(s', \text{"normal"})$ IN
  IF $b = \texttt{True}$ THEN $\text{equals}(s', t')$
  ELSE $\neg\, \text{equals}(s', t')$ (G-1.2)

Goal: G-1.1

We split the goal (G-1.1) into following three sub-goals:
$\text{equals}(s, t)$ (G-1.1.1)
$\llbracket\beta\rrbracket(e_s)(e_s', t, t')$ (G-1.1.2)
$\text{monitor}(\kappa, \beta)(e_r;e_s)(e_r';e_s', s;t, s';t')$ (G-1.1.3)

Sub-Goal: G-1.1.1

We define

$$t := \text{constructs}(s, \beta) \tag{5}$$

We instantiate Lemma (1) with s as s, t as t, ω as β to get

$$t := \text{constructs}(s, \beta) \Rightarrow \text{equals}(s, t) \tag{I.1}$$

The goal (G-1.1.1) follows from (I.1) and definition (5). □

Sub-Goal: G-1.1.2

We expand definition (2) and get

$\forall\, m, m' \in \text{State}, n, n' \in \text{Environment}:$
$\llbracket\kappa\rrbracket(n)(n', m, m') \wedge \text{eqMode}(m, \text{"normal})$
$\Rightarrow \llbracket\beta\rrbracket(n)(n', m, m')$ (F.2)

We instantiate formula (F.2) with m as s;t, m' as s';t', n as $e_r;e_s$', n' as $e_r';_s$' and κ with κ to get

$\llbracket\kappa\rrbracket(e_r;e_s)(e_r';e_s', s;t, s';t') \wedge \text{eqMode}(s;t, \text{"normal"})$
$\Rightarrow \llbracket\beta\rrbracket(e_r;e_s)(e_r';e_s', s;t, s';t')$ (I.2)

We instantiate Lemma (4) with s as s, s' as s', t as t, t' as t', e_r as e_r, e_r' as e_r', e_s as e_s, e_s' as e_s', κ as κ and get

$\llbracket\kappa\rrbracket(e_r;e_s)(e_r';e_s', s;t, s';t') \Leftrightarrow \llbracket\kappa\rrbracket(e_r)(e_r', s, s')$ (I.3)

We instantiate Lemma (6) with s as s, t as t, t’ and get

$$\text{eqMode}(s;t,\ \text{”normal”}) \Leftrightarrow \text{eqMode}(s,\ \text{”normal”}) \qquad (I.4)$$

From (I.2) with assumptions (3), (4), (I.3) and (I.4) we get

$$\llbracket \beta \rrbracket(e_r;e_s)(e_r';e_s',\ s;t,\ s';t') \qquad (I.2')$$

We instantiate Lemma (5) with s as s, s’ as s’, t as t, t’ as t’, e_r as e_r, e_r’ as e_r’, e_s as e_s, e_s’ as e_s’, ω as β and get

$$\llbracket \beta \rrbracket(e_r;e_s)(e_r';e_s',\ s;t,\ s';t') \Leftrightarrow \llbracket \beta \rrbracket(e_r)(e_r',\ s,\ s') \qquad (I.5)$$

The goal (G-1.1.2) follows from (I.5) with assumption (I.2’).□.

Sub-Goal: G-1.1.3

We instantiate induction assumption (on η) with κ as κ, ω as β, b as b to get

$$\begin{array}{l}
\forall\ e_s \in \text{Environment}_s,\ e_r,\ e_r' \in \text{Environment}_r,\ s,\ s' \in \text{State}_r\text{:} \\
\quad \text{consistent}(e_s,\ e_r) \wedge \text{consistent}(\kappa,\ \beta) \wedge \\
\quad \llbracket \kappa \rrbracket(e_r)(e_r',\ s,\ s') \wedge \text{eqMode}(s,\ \text{”normal”}) \\
\Rightarrow \\
\quad \exists\ t,\ t' \in \text{State}_s,\ e_s' \in \text{Environment}_s\text{:} \\
\qquad \text{equals}(s,\ t) \wedge \llbracket \beta \rrbracket(e_s)(e_s',\ t,\ t') \wedge \\
\qquad \text{monitor}(\kappa,\ \beta)(e_r;e_s)(s;t,\ s';t') \wedge \\
\quad \forall\ t,\ t' \in \text{State}_s,\ e_s' \in \text{Environment}_s\text{:} \\
\qquad \text{equals}(s,\ t) \wedge \llbracket \beta \rrbracket(e_s)(e_s',\ t,\ t') \wedge \\
\qquad \text{monitor}(\kappa,\ \beta)(e_r;e_s)(s;t,\ s';t') \\
\quad \Rightarrow \\
\quad \text{LET } b = \text{eqMode}(s',\ \text{”normal”}) \text{ IN} \\
\qquad\quad \text{IF } b = \texttt{True} \text{ THEN equals}(s',\ t') \\
\qquad\quad \text{ELSE} \neg\ \text{equals}(s',\ t') \qquad\qquad (I.6)
\end{array}$$

We instantiate (I.6) with e_s as e_s, e_s’ as e_s’, e_r as e_r, e_r’ as e_r’, s as s, s’ as s’ to get

$$\begin{array}{l}
\text{consistent}(e_s,\ e_r) \wedge \text{consistent}(\kappa,\ \beta) \wedge \\
\quad \llbracket \kappa \rrbracket(e_r)(e_r',\ s,\ s') \wedge \text{eqMode}(s,\ \text{”normal”}) \\
\Rightarrow \\
\quad \exists\ t,\ t' \in \text{State}_s,\ e_s' \in \text{Environment}_s\text{:} \\
\qquad \text{equals}(s,\ t) \wedge \llbracket \beta \rrbracket(e_s)(e_s',\ t,\ t') \wedge \\
\qquad \text{monitor}(\kappa,\ \beta)(e_r;e_s)(s;t,\ s';t') \wedge \\
\quad \forall\ t,\ t' \in \text{State}_s,\ e_s' \in \text{Environment}_s\text{:}
\end{array}$$

```
      equals(s, t) $\wedge$ $[\![\beta]\!]$(e$_s$)(e$_s$', t, t') $\wedge$
      monitor($\kappa$, $\beta$)(e$_r$;e$_s$)(s;t, s';t')
    $\Rightarrow$
    LET b = eqMode(s', "normal") IN
          IF b = True THEN equals(s', t')
          ELSE $\neg$ equals(s', t')                 (I.6.1)
```

The goal (G-1.1.3) follows from (I.6.1) with assumptions (1), (2), (3), (4). Hence goal (G-1.1) is proved. □

Goal: G-1.2

We know

- equals(s, t) (6)
- $[\![\beta]\!]$(e$_s$)(e$_s$', t, t') (7)
- monitor(κ, β)(e$_r$)(e$_r$', s, s') (8)

We show

```
LET b = eqMode(s', "normal") IN
      IF b = True THEN equals(s', t')
      ELSE $\neg$ equals(s', t')                 (G-1.2')
```

We have two cases here

<u>Case 1: b = `True`</u>

We know

eqMode(s', "normal") (10)

We show

equals(s', t') (G-1.2")

We define

t' := constructs(s', β) (11)

We instantiate Lemma (1) with s as s', t as t' to get

t' := constructs(s', β) $\Rightarrow$ equals(s', t') (I.7)

The goal (G-1.2") follows from (I.7) with def. (11) and (10).□

Case 2: b = False

We know

$$\neg \text{ eqMode(s', ”normal”)} \qquad (12)$$

We instantiate Lemma (7) with s as s' and get

$$\neg \text{ eqMode(s', ”normal”)} \Rightarrow \text{eqMode(s', ”compromised”)} \qquad (\text{I.8})$$

From (I.8) with assumption (12), we know

$$\text{eqMode(s', ”compromised”)} \qquad (13)$$

We show

$$\neg \text{ equals(s', t')} \qquad (\text{G-1.2”'})$$

We instantiate Lemma (2) with s as s, s' as s', t as t, t' as t', e_r as e_r, e_r' as e_r', e_s as e_s, e_s' as e_s', κ as κ, ω as β to get

$[\![\kappa]\!](e_r)(e_r\text{', s, s'}) \wedge [\![\beta]\!](e_s)(e_s\text{', t, t'})$
$\wedge$ equals(s, t) $\wedge$ eqMode(s', “compromised”)
$\Rightarrow$ t' $\neq$ constructs(s', β) (I.9)

From (I.9), with assumptions (3), (7), (6) and (13) we get

$$\text{t'} \neq \text{constructs(s', } \beta) \qquad (14)$$

We instantiate Lemma (3) with s as s', t as t', ω as β to get

$$\text{t'} \neq \text{constructs(s', } \beta) \Rightarrow \neg \text{ equals(s', t')} \qquad (\text{I.10})$$

The goal (G-1.2”') follows from (I.10) with assumption (14). The proof of (G-1.2') and (G-1.2”') implies the goal (G-1.2'). □
Hence, the goal (G-1.2) follows from the proofs of (G-1.2.1) and (G-1.2.2). The premise eqMode(s', ”compromised”) of (I.9) shows that the program execution state s' has been compromised.□

6 Proof of the Completeness

The proof of completeness is very similar to what we have already presented above for the soundness. However, the proof differs only for the goal (G-1.2) whose proof is presented in the previous subsection.

In the following, first we formulate the completeness theorem:

Theorem 2 (Completeness of security monitor) *The result of the security monitor is complete for a given execution of the target system and its specification, iff, the specification is consistent with the program and the program executes in a safe pre-state and in an environment that is consistent with the environment of the specification, then*

- *for the pre-state of the program, there is an equivalent safe pre-state for which the specification can be executed and the monitor can be observed and*
- *if we execute the specification in an equivalent safe pre-state and observe the monitor at any arbitrary (combined) post-state, then*
  - *either the program execution (post-state) is semantically consistent with the specification execution (post-state), then there is no alarm and the program execution is safe*
  - *or the program execution (post-state) and the specification execution (post-state) are semantically inconsistent, then there is an alarm and the program execution has been compromised.*

Formally, completeness theorem has the following signatures and definition.

```
Completeness_ad $\subseteq \mathbb{P}$(AppImpl $\times$ AppSpec $\times$ Bool)
Completeness_ad($\kappa$, $\omega$, b) $\Leftrightarrow$
$\forall$ $e_s \in$ Environment$_s$, $e_r$, $e_r$' $\in$ Environment$_r$, s, s' $\in$ State$_r$:
  consistent($e_s$, $e_r$) $\wedge$ consistent($\kappa$, $\omega$) $\wedge$
  $[\![\kappa]\!]$($e_r$)($e_r$', s, s') $\wedge$ eqMode(s, ”normal”)
$\Rightarrow$
   $\exists$ t, t' $\in$ State$_s$, $e_s$' $\in$ Environment$_s$:
      equals(s, t) $\wedge$ $[\![\omega]\!]$($e_s$)($e_s$', t, t') $\wedge$
      monitor($\kappa$, $\omega$)($e_r$;$e_s$)(s;t, s';t') $\wedge$
   $\forall$ t, t' $\in$ State$_s$, $e_s$' $\in$ Environment$_s$:
      equals(s, t) $\wedge$ $[\![\omega]\!]$($e_s$)($e_s$', t, t') $\wedge$
      monitor($\kappa$, $\omega$)($e_r$;$e_s$)(s;t, s';t')
   $\Rightarrow$
      IF equals(s', t') THEN
          b = True $\wedge$ b = eqMode(s', “normal”)
      ELSE b = False $\wedge$ b = eqMode(s', ”normal”)      (G')
```

In detail, the completeness statement says that, if

1. a specification environment (e_s) is *consistent* with a run-time environment (e_r) and
2. a target system (κ) is *consistent* with its specification (ω) and

3. in a given run-time environment (e_r), execution of the system (κ) transforms pre-state (s) into a post-state (s') and
4. the pre-state (s) is safe, i.e. the state is in "normal" mode,

then

- there is such pre- and post-states (t and t' respectively) and environment (e_s') of specification execution such that in a given specification environment (e_s), execution of the specification (ω) transforms pre-state (t) into a post-state (t') and
- the pre-states s and t are *equal* and *monitor*ing of the system (κ) transforms combined pre-state (s;t) into a combined post-state (s';t') and if
- in a given specification environment (e_s), execution of the specification (ω) transforms pre-state (t) into a post-state (t') and
- the pre-states s and t are *equal* and *monitor*ing of the system (κ) transforms pre-state (s) into a post-state (s'), then
  - either the resulting two post-states s' and t' are semantically *equal* and there is no alarm
  - or the resulting two post-states s' and t' are semantically not *equal* and then the security monitor alarms.

In the following, we discuss proof of the completeness statement.

6.1 Proof

6.1.1 Case when $\eta = \beta$

We can re-write (G') as

$$\begin{array}{l}
\text{Soundness\_ad}(\kappa, \beta, \text{b}) \Leftrightarrow \\
\forall\ \text{e}_s \in \text{Environment}_s,\ \text{e}_r,\ \text{e}_r\text{'} \in \text{Environment}_r,\ \text{s, s'} \in \text{State}_r\text{:} \\
\quad \text{consistent}(\text{e}_s, \text{e}_r) \wedge \text{consistent}(\kappa, \beta) \wedge \\
\quad [\![\kappa]\!](\text{e}_r)(\text{e}_r\text{', s, s'}) \wedge \text{eqMode(s, "normal")} \\
\Rightarrow \\
\quad \exists\ \text{t, t'} \in \text{State}_s,\ \text{e}_s\text{'} \in \text{Environment}_s\text{:} \\
\qquad \text{equals(s, t)} \wedge [\![\beta]\!](\text{e}_s)(\text{e}_s\text{', t, t'}) \wedge \text{monitor}(\kappa, \beta)(\text{e}_r\text{;e}_s)\text{(s;t, s';t')} \wedge \\
\quad \forall\ \text{t, t'} \in \text{State}_s,\ \text{e}_s\text{'} \in \text{Environment}_s\text{:} \\
\qquad \text{equals(s, t)} \wedge [\![\beta]\!](\text{e}_s)(\text{e}_s\text{', t, t'}) \wedge \text{monitor}(\kappa, \beta)(\text{e}_r\text{;e}_s)\text{(s;t, s';t')} \\
\quad \Rightarrow \\
\qquad\ \text{IF equals(s', t') THEN b} = \texttt{True} \wedge \text{b} = \text{eqMode(s', "normal")} \\
\qquad\ \text{ELSE b} = \texttt{False} \wedge \text{b} = \text{eqMode(s', "normal")} \qquad\qquad \text{(F'.1)}
\end{array}$$

From (F'.1), we know

- consistent(e_s, e_r) (1')
- consistent(κ, β) (2')
- $[\![\kappa]\!]$(e_r)(e_r', s, s') (3')
- eqMode(s, "normal") (4')

We show

- $\exists$ t, t' $\in$ $State_s$, e_s' $\in$ $Environment_s$: equals(s, t) $\wedge$
  $[\![\beta]\!]$(e_s)(e_s', t, t') $\wedge$ monitor(κ, β)(e_r;e_s)(s;t, s';t') (G'-1.1)
- $\forall$ t, t' $\in$ $State_s$, e_s' $\in$ $Environment_s$: equals(s, t) $\wedge$
  $[\![\beta]\!]$(e_s)(e_s', t, t') $\wedge$ monitor(κ, β)(e_r;e_s)(s;t, s';t')
  $\Rightarrow$
  IF equals(s', t') THEN b = `True`$\wedge$b=eqMode(s', "normal")
  ELSE b = `False` $\wedge$ b = eqMode(s', "normal") (G'-1.2)

Goal: G'-1.1

The proof is similar to as for the soundness goal (G.1.1) as discussed in the subsection. □

Goal: G'-1.2

We know

- equals(s, t) (5')
- $[\![\beta]\!]$(e_s)(e_s', t, t') (6')
- monitor(κ, β)(e_r)(e_r', s, s') (7')

We show

IF equals(s', t') THEN b = `True` $\wedge$ b = eqMode(s', "normal")
ELSE b = `False` $\wedge$ b = eqMode(s', "normal") (G'-1.2')

We have two cases here

<u>Case 1: equals(s’, t’) holds</u>

We know

$$\text{equals(s', t')} \qquad\qquad (8')$$

We show

$$b = \texttt{True} \wedge b = \text{eqMode(s', ”normal”)} \qquad\qquad \text{(G'-1.2”)}$$

To prove the goal, it suffices to show

$$\text{eqMode(s', ”normal)} = \texttt{True} \qquad\qquad \text{(G'-1.2”.1)}$$

We instantiate Lemma (8) with s as s’ and t as t’ and get

$$\text{equals(s', t')} \Rightarrow \text{eqMode(s', ”normal”)} \qquad\qquad \text{(I'.1)}$$

The goal (G’.1.2”.1) follows from (I’.1) with assumption (8’). Hence the goal (G’.1.2”) is proved. □

Furthermore, the goal shows that there is no alarm when the two post-states (s’ and t’) are equivalent and are not compromised.

<u>Case 2: ¬ equals(s’, t’) holds</u>

We know

$$\neg\, \text{eqMode(s', ”normal”)} \qquad\qquad (9')$$

We show

$$b = \texttt{False} \wedge b = \text{eqMode(s', ”normal”)} \qquad\qquad \text{(G'-1.2”)}$$

To prove the goal, it suffices to show

$$\text{eqMode(s', ”normal)} = \texttt{False} \qquad\qquad \text{(G'-1.2”.1)}$$

We instantiate Lemma (9) with s as s’ and t as t’ and get

$$\neg\, \text{equals(s', t')} \Rightarrow \neg\, \text{eqMode(s', ”normal”)} \qquad\qquad \text{(I'.3)}$$

The goal (G’.1.2”.1) follows from (I’.3) with assumption (9’). Hence the goal (G’.1.2”) is proved. □
Furthermore, we instantiate Lemma (7) with s as s’ to get

$$\neg\, \text{eqMode(s', ”normal”)} \Rightarrow \text{eqMode(s', ”compromised”)} \quad \text{(I'.4)}$$

From (I’.4) with the proved goal (G’.1.2”) we get

$$\text{eqMode(s', ”compromised”)}$$

that shows that the alarm is generated when the post-states (s’ and t’) are semantically not equal. Furthermore, from the assumption (2’) if follows that indeed the program execution (post-state) is compromised.

7 Auxiliary Functions and Predicates

In this section, we declare respectively define auxiliary functions and predicates that are used in the proof of soundness and completeness above.

- $\textbf{constructs : State}_r \times \textbf{AppSpec} \rightarrow \textbf{State}_s$
  $\text{constructs(s, } \omega) = \text{t,}$
  $\quad \text{s.t. t} = \text{build}(\omega) \wedge \text{eqMode(s, "normal")} \wedge \text{abstract(s, t)}$

- $\textbf{constructs : Environment}_r \times \textbf{AppSpec} \rightarrow \textbf{Environment}_s$
  $\text{constructs(e, } \omega) = \text{e', s.t. e'} = \text{build}(\omega) \wedge \text{abstract(e, e')}$

- $\_\,;\,\_ : \textbf{State}_r \times \textbf{State}_s \rightarrow \textbf{State}$
  $\text{s;t} = \text{state}(\{\langle \text{I:v} \rangle \in \text{store(s)} : \neg\exists \langle \text{I:v'} \rangle \in \text{store(t)}\} \cup$
  $\quad \{\langle \text{I:v'} \rangle \in \text{store(t)} : \neg\exists \langle \text{I:v} \rangle \in \text{store(s)}\} \cup$
  $\quad \{\langle \text{I:v"} \rangle : \exists \text{ v": } \langle \text{I:v} \rangle \in \text{store(s)} \wedge \langle \text{I:v'} \rangle \in \text{store(t)} \wedge$
  $\qquad \text{v"} = \text{super(v, v')}\}, \text{flag(s))}$

- $\_\,;\,\_ : \textbf{Environment}_r \times \textbf{Environment}_s \rightarrow \textbf{Environment}$
  $\text{e;e'} = \text{environment}(\{\langle \text{I:v} \rangle \in \text{context(e)} :$
  $\quad \neg\exists \langle \text{I:v'} \rangle \in \text{context(e')}\} \cup$
  $\quad \{\langle \text{I:v'} \rangle \in \text{context(e')} : \neg\exists \langle \text{I:v} \rangle \in \text{context(e)}\} \cup$
  $\quad \{\langle \text{I:v"} \rangle : \exists \text{ v": } \langle \text{I:v} \rangle \in \text{context(e)} \wedge$
  $\qquad \langle \text{I:v'} \rangle \in \text{context(e')} \wedge \text{v"} = \text{super(v, v')}\}$
  $\qquad \text{, space(e))}$

- $\textbf{super : Value}_r \times \textbf{Value}_s \rightarrow \textbf{Value}$
  $\text{super(v, v')} = \text{v , if } \llbracket \text{v} \rrbracket \subseteq \llbracket \text{v'} \rrbracket$
  $\qquad \text{v', if } \llbracket \text{v'} \rrbracket \subseteq \llbracket \text{v} \rrbracket$

- $\textbf{super : EnvVal}_r \times \textbf{EnvVal}_s \rightarrow \textbf{EnvVal}$
  $\text{super(v, v')} = \text{v , if } \llbracket \text{v} \rrbracket \subseteq \llbracket \text{v'} \rrbracket$
  $\qquad \text{v', if } \llbracket \text{v'} \rrbracket \subseteq \llbracket \text{v} \rrbracket$

- $\textbf{equals} \subseteq \mathbb{P}(\textbf{State}_r \times \textbf{State}_s)$
  $\text{equals(s, t)} \Leftrightarrow$
  $\forall \text{ c:Component}_s, \omega\text{:AppSpec}, \kappa\text{: AppImpl:}$
  $\quad \text{c} \in \omega \wedge \text{c} \in \kappa \wedge \llbracket \text{c} \rrbracket(\text{e}_r)(\text{s, s', e}_r\text{'})$
  $\Rightarrow \llbracket \text{c} \rrbracket(\text{e}_s)(\text{t, t', e}_s\text{'}) \wedge$
  $\quad \forall \text{ id: Identifier}_s\text{, v: Value}_s\text{: } \langle \text{id, v} \rangle \in \text{store(t)}$
  $\quad \Rightarrow \langle \text{id, v'} \rangle \in \text{store(s)} \wedge \text{abstract(v, v')}$

- $\textbf{consistent} \subseteq \mathbb{P}(\textbf{Environment}_r \times \textbf{Environment}_s)$
  $\text{consistent(e}_r\text{, e}_s) \Leftrightarrow$
  $\forall \text{ id:Identifier, v: Value}_s\text{, v': Value}_r\text{:}$
  $\quad \langle \text{id, v} \rangle \in \text{context(e}_s) \Rightarrow \langle \text{id, v'} \rangle \in \text{context(e}_r) \wedge \text{abstract(v, v')}$

- **consistent** $\subseteq$ **$\mathbb{P}$(AppImpl $\times$ AppSpec)**
  consistent(κ, ω) $\Leftrightarrow$ the safe execution of ”κ” meets ”ω” and ”ω” always executes in a safe state, that can be formulated as follows:
  $\forall$ s, s’ $\in$ State, e, e’ $\in$ Environment:
  $[\![\kappa]\!]$(e)(e’, s, s’) $\wedge$ eqMode(s, ”normal”) $\Rightarrow$ $[\![\omega]\!]$(e)(e’, s, s’) $\wedge$
  $\forall$ t, t’ $\in$ State$_s$, d, d’ $\in$ Environment$_s$:
  $[\![\omega]\!]$(d)(d’, t, t’) $\wedge$ eqMode(t, ”normal”) $\Rightarrow$ eqMode(t’, ”normal”)

  Semantically, the predicate “consistent” returns `True` iff only such pair of states (s and s’) are related by ”κ” which is also related by ”ω”. Here the states and environment are combined of two corresponding abstractions of specification and implementation respectively. Furthermore, execution of ”ω” in a safe pre-state always yields a safe post-state.

- **abstract** $\subseteq$ **$\mathbb{P}$(State$_r$ $\times$ State$_s$)**
  abstract(s, t) $\Leftrightarrow$
  $\forall$ i:Identifier, v:Value$_s$:
  $\langle$i, v$\rangle$ $\in$ store(t) $\Rightarrow$ $\exists$ v’:Value$_r$:$\langle$i, v’$\rangle$ $\in$ store(s) $\wedge$
  abstract(v, v’)

- **abstract** $\subseteq$ **$\mathbb{P}$(Value$_r$ $\times$ Value$_s$)**
  abstract(v, v’) $\Leftrightarrow$
  $\forall$ τ, τ’:Type, s:State$_r$, t:State$_s$:
  equals(s, t) $\wedge$ $[\![$v$]\!]$(s, τ) $\wedge$ $[\![$v’$]\!]$(t, τ’) $\Rightarrow$ $[\![\tau’]\!]\subseteq$ $[\![\tau]\!]$

- **abstract** $\subseteq$ **$\mathbb{P}$(EnvVal$_r$ $\times$ EnvVal$_s$)**
  abstract(v, v’) $\Leftrightarrow$
  $\forall$ τ, τ’:Type, e:Environment$_s$, e’:Environment$_r$:
  consistent(e, e’) $\wedge$ $[\![$v$]\!]$(e, τ) $\wedge$ $[\![$v’$]\!]$(e’, τ’) $\Rightarrow$ $[\![\tau’]\!]\subseteq$ $[\![\tau]\!]$

8 Lemmas

In this section, we give definitions and corresponding proof hints of lemmas that were used in the proofs above.

Lemma 1

$\forall$ s $\in$ State$_r$, t $\in$ State$_s$: t = constructs(s) $\Rightarrow$ equals(s, t)

Lemma 2

$\forall$ s, s’ $\in$ State$_r$, t, t’ $\in$ State$_s$,
κ $\in$ AppImpl, ω $\in$ AppSpec,
e$_r$, e$_r$’ $\in$ Environment$_r$, e$_s$, e$_s$’ $\in$ Environment$_s$:
$[\![\kappa]\!]$(e$_r$)(e$_r$’, s, s’) $\wedge$ $[\![\omega]\!]$(e$_s$)(e$_s$’, t, t’)

$\wedge$ equals(s, t) $\wedge$ eqMode(s', “compromised”)
$\Rightarrow$ t' $\neq$ constructs(s')

Proof Hints In principle, from a compromised program state, an equivalent specification safe state cannot be constructed because the program state may have inconsistent values for certain variables or new variables etc.

Lemma 3

$\forall$ s $\in$ State$_r$, t $\in$ State$_s$: t $\neq$ constructs(s) $\Rightarrow \neg$ equals(s, t)

Lemma 4

$\forall$ s, s' $\in$ State, t, t' $\in$ State$_s$,
e$_r$, e$_r$' $\in$ Environment$_r$, e$_s$, e$_s$' $\in$ Environment$_s$,
$\kappa \in$ AppImpl:
$\llbracket\kappa\rrbracket$(e$_r$;e$_s$)(e$_r$';e$_s$', s;t, s';t') $\Leftrightarrow$ $\llbracket\kappa\rrbracket$(e$_r$)(e$_r$', s, s')

Proof Hints The goal follows from the semantics of κ.

Lemma 5

$\forall$ s, s' $\in$ State, t, t' $\in$ State$_s$,
e$_r$, e$_r$' $\in$ Environment$_r$, e$_s$, e$_s$' $\in$ Environment$_s$,
$\omega \in$ AppSpec:
$\llbracket\omega\rrbracket$(e$_r$;e$_s$)(e$_r$';e$_s$', s;t, s';t') $\Leftrightarrow$ $\llbracket\omega\rrbracket$(e$_s$)(e$_s$', t, t')

Lemma 6

$\forall$ s $\in$ State, t $\in$ State$_s$:
eqMode(s;t, ”normal”) $\Leftrightarrow$ eqMode(s, ”normal”)

Lemma 7

$\forall$ s $\in$ State$_r$:
$\neg$ eqMode(s', ”normal”) $\Leftrightarrow$ eqMode(s', ”compromised”)

Lemma 8

$\forall$ s $\in$ State$_r$, t $\in$ State$_s$: equals(s, t) $\Rightarrow$ eqMode(s, ”normal”)

Proof Hint The definition of *equals* enables to show the goal. Also because of the fact, that two states are only equal if they can be constructed in a safe mode.

Lemma 9

$\forall$ s $\in$ State$_r$, t $\in$ State$_s$:
$\neg$ equals(s, t) $\Rightarrow \neg$ eqMode(s, ”normal”)

9 Conclusion

We have presented a sound and complete run-time security monitor for application software, which avoids false alarms (positive or negative). The monitor implements run-time software verification, comparing an executable application specification with the execution of its implementation at run-time. Our main contribution, the proof of soundness and completeness, establishes an *assume/guarantee*-based contract between the *security monitor* and its user, i.e. the designer of the application to be monitored. Specifically, if the user establishes the *assumptions* of the proof, then the monitor *guarantees* to detect all deviations of the executions behaviour relatively to the behaviour defined in the application specification and will never produce any false alarm at run-time. Importantly, the proof strategy can be a fundamental building block for:

1. any proof that shows that an abstract description/specification (non-determinism) of a program is consistent with its concrete description/implementation (determinism/instance),
2. transformation rules to automatically generate sound and complete monitors (for program execution) from specification and
3. developing proof tactics to prove such tedious goals semi-automatically, significantly reducing human effort.

Our future work includes the mechanization of this proof in a proof assistant, specifically Coq, targeting the development of a generic library based on our proof strategy so that the proof can be applied to any given specification and implementation.

References

[1] C.A.R. Hoare. *Proof of Correctness of Data Representations.* Acta Informatica, 1(4):271–281, 1972.

[2] Muhammad Taimoor Khan, Dimitrios Serpanos, and Howard Shrobe. *On the Formal Semantics of the Cognitive Middleware AWDRAT.* Technical Report MIT-CSAIL-TR-2015-007, Computer Science and Artificial Intelligence Laboratory, MIT, USA, March 2015.

[3] Howard Shrobe, Robert Laddaga, Bob Balzer, Neil Goldman, Dave Wile, Marcelo Tallis, Tim Hollebeek, and Alexander Egyed. *AWDRAT: A Cognitive Middleware System for Information Survivability*'. In Proceedings of the 18th Conference on Innovative Applications of Artificial Intelligence - Volume 2, IAAI'06, pages 1836–1843. AAAI Press, 2006.

[4] E. Borger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[5] Hannan, John and Miller, Dale. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Mathematical Structures in Computer Science:2(4), pages 415–459, 1992.

[6] Barringer, Howard and Goldberg, Allen and Havelund, Klaus and Sen, Koushik. *Program Monitoring with LTL in EAGLE.* In Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), RISC Report Series, TR-12-08, pages 26–30, IEEE Computer Society, USA, 2004.

[7] Bauer, Andreas and Leucker, Martin and Schallhart, Christian. *Runtime Verification for LTL and TLTL.* In ACM Transactions on Software Engineering and Methodology:20(4), pages 14:1–14:64, 2011.

[8] Schmidt, David A. *Denotational Semantics: a methodology for language development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[9] H. Barringer, D. Rydeheard, K. Havelund. *Rule systems for run-time monitoring: from Eagle to RuleR.* In Journal of Logic and Computation:20(3), pages 675–706, 2010.

[10] George Spanoudakis and Christos Kloukinas and Khaled Mahbub. *The SERENITY Runtime Monitoring Framework.* In Security and Dependability for Ambient Intelligence, Chapter 13, pages 213–237, Advances in Information Security Series, Springer, 2009.

[11] Shrobe, Howard E. *Dependency Directed Reasoning for Complex Program Understanding.* Technical report, 1979.

[12] Langner, Ralph. *Stuxnet: Dissecting a Cyberwarfare Weapon.* In IEEE Security and Privacy, Volume 2, No. 3, pages 49–51. May 2011.

[13] F. Chen, G. Rosu. *MOP: An Efficient and Generic Runtime Verification Framework.* In 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA 07), pages 569–588. ACM, 2007.

[14] Wolfgang Schreiner, Temur Kutsia, Michael Krieger, Bashar Ahmad, Helmut Otto and Martin Rummerstorfer. *Securing Device Communication by Predicate Logic Specifications.* In Proceedings of the Embedded World Conference 2015, Design&Elektronik, pages 9. Nürnberg, Germany, February 24-26 2015.

[15] Kaiser, Gail and Gross, Phil and Kc, Gaurav and Parekh, Janak and Valetto, Giuseppe. *An Approach to Autonomizing Legacy Systems.* In Proceedings of the Workshop on Self-Healing, Adaptive and Self-MANaged Systems, June 2002.

[16] Temur Kutsia, Wolfgang Schreiner. *Verifying the Soundness of Resource Analysis for LogicGuard Monitors (Revised Version).* In RISC Report Series, TR-14-08, JKU, Austria, 2014.

[17] Temur Kutsia, Wolfgang Schreiner. *Logic Guard Abstract Language.* In RISC Report Series, TR-12-08, JKU, Austria, 2012.

[18] Wasserman, Hal and Blum, Manuel. *Software Reliability via Run-time Result-checking.* In Journal of ACM:44(6), pages 826–849, ACM, 1997.

[19] Barnett, Mike and Schulte, Wolfram. *Runtime Verification of .NET Contracts.* In Journal of Systems and Software: 65(3), pages 199–208, Elsevier Science Inc., 2003.

[20] Jesper G. Henriksen, Ole J.L. Jensen, Michael E. Jrgensen, Nils Klarlund, Robert Paige, Theis Rauhe and Anders B. Sandholm. *MONA: Monadic Second-Order Logic in Practice.* In Tools and Algorithms for the Constructive and Analysis of Systems, LNCS 1019, Springer-Verlag, 1995.

[21] Chupilko, Mikhail M. and Kamkin, Alexander S.. *Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces.* In Proceedings Eighth Workshop on Model-Based Testing, EPTCS, pages 67–81, 2013.

[22] Haogang Chen, Daniel Ziegler, Adam Chlipala, Nickolai Zeldovich, Frans Kaashoek. *Using Crash Hoare Logic for Certifying the FSCQ File System.* In Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15). October 2015.

[23] Vern Paxson. *Bro: A System for Detecting Network Intruders in Real-time.* In Proceedings of the 7th conference on USENIX Security Symposium - Volume 7, USENIX Association, Berkeley, USA. 1998.

[24] Martin Roesch. *Snort - Lightweight Intrusion Detection for Networks.* In Proceedings of the 13th USENIX conference on System administration (LISA '99). USENIX Association, Berkeley, CA, USA. 1999.

[25] S. Kim, A. L. N. Reddy, and M. Vannucci. *Detecting Traffic Anomalies through Aggregate Analysis of Packet Header Data.* In Networking. 2004.

[26] Lakhina, Anukool and Crovella, Mark and Diot, Christophe. *Mining Anomalies Using Traffic Feature Distributions.* In Proceedings of ACM SIGCOMM. 2005.

[27] Victoria Hodge and Jim Austin. *Adaptive, Model-based Monitoring for Cyber Attack Detection.* In Artificial Intelligence Review. 22(2), pages 85–126. October 2004.

[28] Valdes, A. and Skinner, K. *Mining Anomalies Using Traffic Feature Distributions.* In Proceedings of the 3rd International Workshop on Recent Advances in Intrusion Detection. Springer-Verlag, pages 80–92. 2000.

[29] Watterson, C. and Heffernan, D.. *Runtime Verification and Monitoring of Embedded Systems.* In Software, IET , Volume 1(5), pages 172–179. October 2007.

[30] Ji Zhang and Betty H.C. Cheng. *AMOEBA-RT: Run-Time Verification of Adaptive Software.* In Lecture Notes in Computer Science (Models in Software Engineering), Springer Berlin Heidelberg, Volume 5002. 2008.

[31] D. Drusinsky and J.L. Fobes. *Executable Specifications: Language and Applications.* In Department of Defense Crosstalk Magazine, Journal of Defense Software Engineering. September 2004.