Formal comparison of separation kernel models
GWV separation versus Rushby's non interference

Garcia Ramirez, A.

Award date:
2014

Technische Universiteit Eindhoven

Master Thesis

Formal Comparison Of Separation Kernel Models: GWV Separation Versus Rushby's Non Interference

Author:
Adrian Garcia Ramirez

Supervisor:
Julien Schmaltz

Assessment Committee:
Julien Schmaltz
Jerry den Hartog
Hans Zantema

A thesis submitted in fulfilment of the requirements for the degree of Master in Computer Science and Engineering.

in the

Department of Mathematics and Computer Science

January 2014

Technische Universiteit
Eindhoven
University of Technology

TECHNISCHE UNIVERSITEIT EINDHOVEN

Abstract

Department of Mathematics and Computer Science

Master in Computer Science and Engineering.

Formal Comparison Of Separation Kernel Models: GWV Separation Versus Rushby's Non Interference

by Adrian Garcia Ramirez

In this document we investigate how formal verification methods can be used to compare and classify separation kernel architectures.

The separation kernels are an integral part of the Multiple Independent Levels of Security (MILS) design pattern. Consisting of relatively small components that run directly over the hardware and whose main task is to create separation or isolation between the different processes running within the host device.

The two main formal models of separation kernels are the one proposed by Greves, Wilding and Vanfleet (GWV) and the one proposed by John Rushby.

In the first part of this dissertation we formalize the original version of the GWV and Rushby models in the Isabelle/HOL theorem prover. Moreover, we present that to our knowledge is the first formal comparison of the aforementioned models.

In the second part, we propose a a mapping between the elements conforming both models, and later we use this mapping to compare the security definitions presented in them.

Finally In the last part we present two examples that illustrate the main differences between the Rushby Non Interference and GWV separation models.

Acknowledgements

Through these lines I want to express my sincere gratitude to all the people whose scientific and human support have collaborated in conducting this research.

I want to thank in a very special manner to my supervisor Julien Schmaltz, for his invaluable help, patience and guidance. His support and advice allowed this project come to fruition. Likewise, I want to thank Jerry den Hartog for the close monitoring he gave to this project and Hans Zantema for being a voting member in the assessment committee.

In general I want to thank all members of the assessment committee for the time spent reviewing this document.

I Also want to thank the "Consejo Nacional De Ciencia Y tecnologia" (CONACYT) and the "Fondo para el Desarrollo de Recursos Humanos" (FIDERH) for economic support provided during the completion of my studies.

Finally, I thank my friends and family for their understanding, support and constant communication.

Contents

List of Figures

List of Tables

Abbreviations

CC	**C**ommon **C**riteria
EAL	**E**valuation **A**ssurance **L**evels
GWV	**G**reve **W**ilding **V**anfleet separation model
MILS	**M**ultiple **I**ndependent **L**evels of **S**ecurity
MLS	**M**ultiple **L**evels of **S**ecurity
NSA	**N**ational **S**ecurity of **A**gency (from the United States of America)
NSA	**N**ational **I**nstitute of **S**tandards and **T**echnology
SKPP	**S**eparation **K**ernel **P**rotection **P**rofile

A mis padres que me han dado el tesoro más valioso que se puede dar a un hijo, amor. Para ellos que no escatimado ningun esfuerzo y han sacrificado gran parte de su vida por sus hijos.

A mi esposa Claudia que ha compartido mis alegrías y preocupaciones, por animarme a superarme todos los días, por su apoyo incondicional y ayuda. En Claudia he encontrado la fuerza necesaria para llegar hasta el final.

Esta tesis está dedicada a ellos.

Chapter 1

Context

There are several security models proposals born hand in hand with the military security classification model [1]. Multiple Levels of Security Models (MLS), correspond to multiple mandatory access control policies[2], in which the data are classified into security levels.

In MLS systems the access to a resource is granted according to the authorization level of the subject trying to access it. The MLS approach has also been called data flow model, because it allows to control the flow of data between different security levels. The MLS best known models are the Bell-La Padula [3] and Biba [4] models.

Multiple Independent Levels of Security (MILS) is an architectural design pattern capable of implementing multi-level security in a simple and adequate manner. The objective of MILS is to ensure the security for high reliability or mission-critical systems.

Figure 1.1 exemplifies the MILS architecture pattern. The main MILS security mechanism, is the separation kernel [5]. The idea behind security kernels consists in isolating all processes and make them appear independent, as if they run on different machines, so they cannot interfere with each other. Interaction between different security levels is only allowed trough channels previously defined.

More background information about MILS and separation kernels can be found in the next Chapter.

Motivation and Contribution

Motivation

Separation kernel is a key element to achieve multiple-level security in MILS systems.

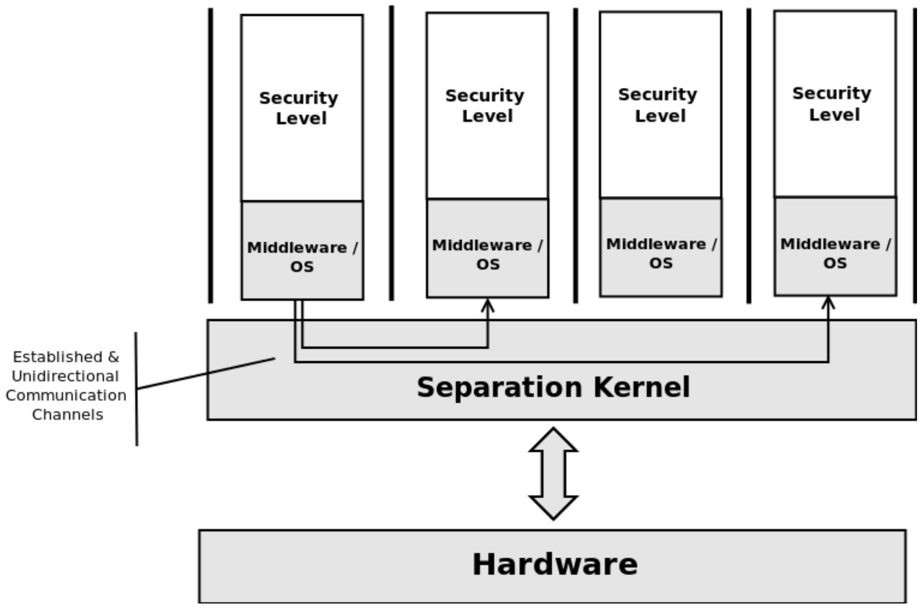

FIGURE 1.1: MILS Architecture Diagram

Formal verification is a fundamental process to ensure the correctness of systems such as separation kernels. However,there is no guideline on how to select the most appropriate separation kernel model.

The two main formal separation kernels models are the GWV model proposed by Greve, Wilding and Vanfleet [6], and the model proposed by John Rushby [7].

The GWV model contains a separation property that can be used to represent the isolation between the separation kernel partitions. Whereas, the Rushby model provides the basis to satisfy a non-interference security policy.

The differences between GWV and Rushby separation kernel models have previously been informally discussed [8]. However, a formal comparison of the concepts in both models has never been performed.

The lack of a formal comparison between the GWV and Rushby models makes it impossible to determine with certainty which of the two proposed security definitions is stronger and what is the exact relationship between the different elements and definitions of both models.

In the present document we do not only formalize the security definition of the GWV and Rushby models, but we also compare the security properties offered by them against each other.

Research Question

The objective of this thesis is to use formal methods to determine the differences between GWV and Rushby's models and to classify these models in terms of their strength and features offered. A model is stronger if it accepts less systems as "secure". A secondary goal is to determine under which conditions a system would be considered secure under both security definitions. Therefore, the main research question to be answered in this thesis is:

> **Research Question:** *"What are the exact differences and similarities between the GWV separation Model and the non-interference model proposed by Rushby?"*.

To reach the this objective, the following sub research questions need to be answered:

- How can we map the concepts of the two models?
- What conditions does the GWV model need to be considered secure under Rushby non-interference definition?
- Which Rushby conditions can be derived from the GWV security definition?
- Can Rushby model ensure security under the GWV definition?
- Is possible to find counterexamples to show the differences of the two models?

Our approach method to answer these questions is to construct a formal proof relating the GWV and Rushby models.

Contribution
The primary contributions of this dissertation can be summarized as follows:

In Chapters 3 and 4, we formalize the original GWV and Rushby Models and reproduce the proofs of the original models in the Isabelle/HOL theorem prover.

Our first four sub-research questions are covered in Chapter 5. In this chapter we provide a mapping between the main elements and functions of both models and we use them to formalize Rushby's reference monitor conditions using the GWV concepts and elements. Later, we prove the relation between Rushby's transitive and intransitive models and GWV.

Our main contribution is pictured in Figure 1.2. This picture illustrates the relation between the different GWV and Rushby elements, line in the figure indicate a relation between definitions, whereas the numbers indicate the Section where the proof of such relation can be found.

As we can see GWV separation definition and one extra condition named *"GWVcond1"* implies Rushby's step consistency definition. Whereas, step consistency implies GWV security (we assume that the action executed in the active partition are the same). In the

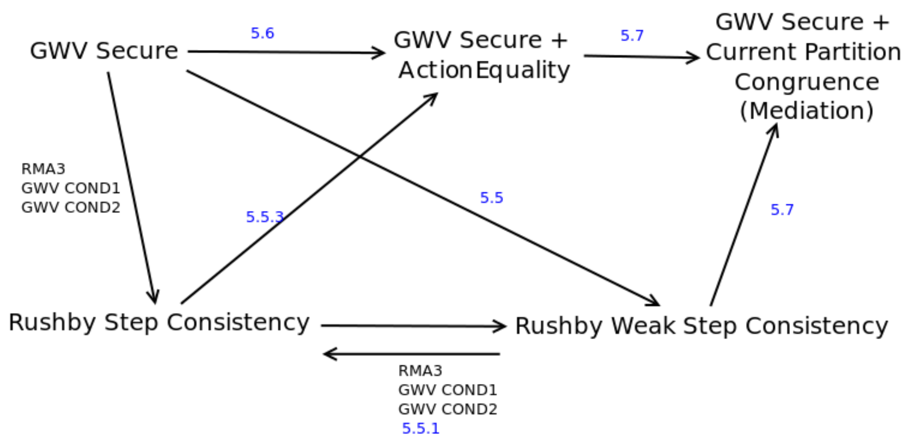
GWV Secure
5.6
GWV Secure + ActionEquality
5.7
GWV Secure + Current Partition Congruence (Mediation)
RMA3
GWV COND1
GWV COND2
5.5.3
5.5
5.7
Rushby Step Consistency
Rushby Weak Step Consistency
RMA3
GWV COND1
GWV COND2
5.5.1

FIGURE 1.2: Relation Among the GWV and Ruhby Security definitions

case of intransitive policies we show that the GWV separation definition implies weakly step consistency, while Rushby's definition can only ensure the property of *mediation* proposed in the original GWV model.

In Chapter 6 we answer the last sub-research question. In this chapter we first present an example that illustrates the need of *GWVcond1* for GWV to be considered secure in the Rushby model. Later, a second example shows that Rushby weakly step consistency cannot ensure GWV security.

Finally, in Chapter 7 the conclusions and some recommendations for further research are described.

Chapter 2

Preliminaries

2.1 Information Security Overview

Information Security is considered a key factor within organizations. However, it has many interpretations and encompasses many different concepts. In a broad context information security is responsible for providing secure and reliable conditions for data processing systems.

The goals of information security are to ensure that information systems resources are available when needed, and to guarantee that access and modification of the information contained in them is only possible within the limits defined by the organization security policies.

The objectives of information security[9] can be summarized by the following principles:

- Confidentiality: Organizations need to protect the privacy of the data stored and processed in computer systems.
- Integrity: A secure system must ensure that the data contained in them can only be modified in appropriate ways, by appropriate people, and that users can rely on the accuracy of the information contained in the system.
- Availability: This refers to continuity of access to the elements of a computer system. Based on this principle, information security must ensure that the services or information supplied by the system are available when required.

2.2 Formal Verification

Commonly, the term verification is used to name the process that aiming at ensuring that a design satisfies the properties that express correct operation of a given system. The formal and precise aspect is achieved by describing the system and its properties in a mathematical language. Verification is the process that ensures, to a certain extent, whether a system meets it desired properties [10].

Formal verification can be useful to prove the correctness of a wide range of systems such as: communication protocols, digital circuits and software. In order to verify a system we need to provide a formal proof of an abstract mathematical model of the system, and demonstrate the correspondence between the mathematical model and the system being developed.

An approach to formal verification is theorem proving[11].

Theorem proving is a technique where the system and its desired properties are expressed in mathematical logic[12]. This logic is given a set of axioms. The theorem proving is then the process of finding a proof of a property using these axioms and inference rules.

Proofs can be obtained by hand, but in recent years there has been an increasing use of theorems provers such as ACL2[13], PVS[14], Coq[15] and Isabele/HOL[16]. Theorem provers can be used as an assistant in the proving process.

Theorem provers vary according to their degree of automation, from fully automatic to interactive provers. The latter, require interaction with a person. On the one hand, human interaction makes them more susceptible to errors. On the other hand, it increases the power of the prover thanks to the contribution that can be made by a human intelligence.

Theorem proving approach requires the user to understand in detail why the system operates properly, and transmit this information to the verification system, either in the form of a sequence of theorems to be proved or in the form of system components specifications.

2.3 Multiple Independent Levels of Security

Multiple independent levels of Security (MILS) is a security model promoted by The Air Force Research Laboratory, in cooperation with the National Security Agency (NSA), Department of Defence contractors, academia and software suppliers, with the goal of ensuring security of systems performing mission-critical functions.

Principles of NEATness

MILS is based on the concepts of separation and information flow control. The goal of this design pattern is that the security mechanisms meet the following properties, appropriately represented by the acronym NEAT [17]:

- **N**on-Bypasable: A security mechanism must be impossible to bypass. Each process by a security mechanism must be obligatorily executed.
- **E**valuatable: It must be possible to show that the security solution was implemented properly. To comply with this principle it should be possible for the MILS designer to demonstrate that the defined security mechanisms functions properly, this can be achieved by using techniques such as code inspection or formal verification
- **A**lways Invoked: Security functions are invoked on every execution of the system.
- **T**amper proof: A security mechanism cannot be modified without authorization. This principle avoids the inclusion of mechanism that could introduce malicious mechanisms to the system.

2.4 The Role of Separation in MILS

Typically the common way to implement security into a system is to add additional elements above the operating system and applications. In contrast, the MILS approach is just the opposite. Systems become secure by simplifying the protection, and as the systems are simpler we can trust them to work properly. Simple means secure [18].

MILS places particular emphasis on the principle of least privilege formulated by Saltszer[19], which states that: "*every program and every user of the system should operate using the least set of privileges necessary to complete the job.*"

To apply the least privilege security principle the designers of MILS compliant systems are required to decompose the obligations of the system. Each obligation is later placed into separate components, in such a way that the resulting components are as simple as possible. This approach involves the advantage that each component can be evaluated and verified in a simple manner.

MILS is a layered approach where the low-level layers provide security services to the upper layers. Under this approach each layer is responsible for the security services only in their own domain.

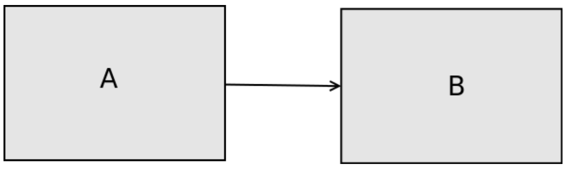

FIGURE 2.1: Simple Security Policy

Imagine for example, a simple security policy like the one shown in Figure 2.1. This policy requires that component A can only send data to B if it has been previously encrypted. Component B should not be able to transmit any data to A.

In the real world policies like this are not as easy to implement as it may seem. As Figure 2.2 shows, components of simple system like this can be very complicated. The only way to ensure that a policy like this is satisfied, is if each one of the elements in A and B have been previously assessed and verified against accidental or malicious mechanisms that can subvert the policy. Therefore we can conclude that to ensure a simple policy in a relatively simple system, is required to verify the security of operating systems, protocols and application software. The costs of such an effort can be really prohibitive.

Now imagine that instead of allowing direct communication between A and B we have a system composed of three different elements connected by specific communication paths as shown in Figure 2.3. In this case the complexity of ensuring the security of the system is simplified, since *crypto* may be a dedicated component whose correctness has been previously verified.

Taking this into consideration it can be said that one way to protect the information from corruption, misuse or unauthorized disclosure is through the separation of the system in different security domains and maintaining the control of information flows between such domains.

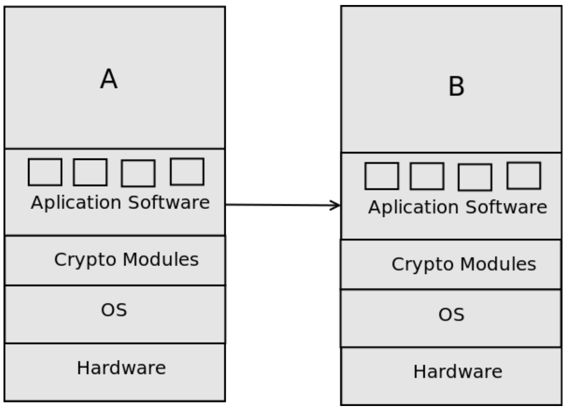

FIGURE 2.2: "Real world" Simple Security Policy

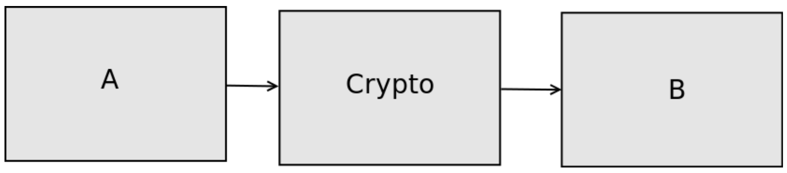

FIGURE 2.3: Simple Security Policy with Duty Separation

Separation is relatively easy to implement if there is an actual physical distance between each of the components, ie each component is really a separate physical device. However, this task is more complex if the system is deployed on a single machine and each component makes use of a single shared processor and resources. Therefore, is duty of MILS designers to establish the mechanisms required to achieve this separation.

2.5 Separation Kernels

From the security standpoint of view, the creation of various independent containers within a single device, enables the possibility of resource isolation. This separation is extremely important for systems requiring multiple levels of security (i. eg. military System) or systems that simultaneously host critical processes along with others applications that may not have been properly verified, may contain errors or even present malicious behaviour.

The idea of splitting a single system on different "virtual" machines to increase their security, was originally proposed by Madnick in 1972 [20]. Nevertheless, existing virtualization systems have not been designed with this goal in mind and cannot ensure the complete separation of the systems hosted in them [21].

Separation kernels were initially proposed by John Rushby [5] as an integral part of the Multiple Independent Levels of Security (MILS) [22] architecture pattern. This special type of security kernel consists of relatively small components that run directly over the hardware and whose main task is to create separation or isolation between the different processes running within the host device.

Jonas Frid identified [23] some attractive features of separation kernels:

Flexibility: using separation kernels it is possible to create systems that range from a few partitions to hundreds of them. Systems can easily be modified by adding more partitions.

Isolation of Security Critical Functions: Owning to the possibility of having secure separated modules, it is possible to extract the security critical functions and allocate

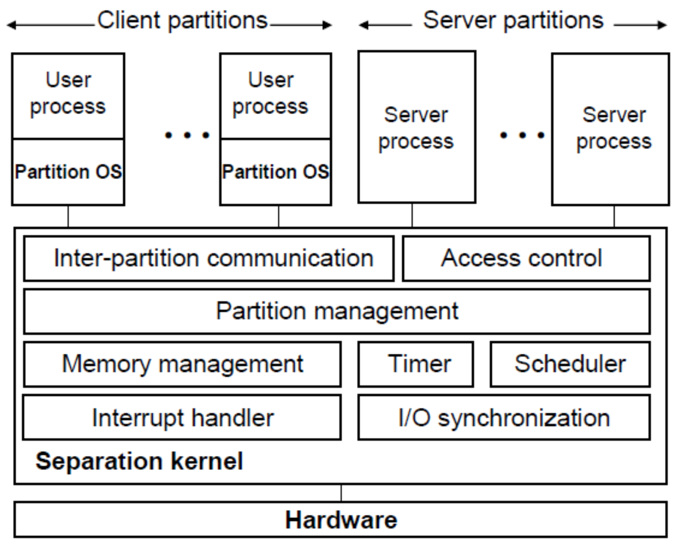

FIGURE 2.4: Architecture of separation-kernel-based system. Figure taken from [24]

them into separate partitions. In consequence security functions can be simpler and specific which leads to easier verification because the modules are smaller.

Figure 2.4 illustrates the general architecture of separation kernels. In a separation kernel, isolation is accomplished through the creation of different security partitions. A certain number of memory segments are assigned to each partition. The processes running within each partition are only allowed to read and write in the segments that correspond to their assigned partition. Meanwhile, the kernel is responsible for controlling the flow of information between the different partitions.

Kei Kawamorita et al. defined three main requirements to ensure the protection of memory in a security kernel[24] that can be listed as follows:

- The memory space of each partition must be isolated from the other partitions, in other words a process being executed in a partition should not be able to access the memory space from other partitions.
- The memory area containing the separation kernel must be only accessible by the same kernel.
- The memory space that contains the operating system of each partition should not be accessible to the user-initiated processes.

Separation kernels are in charge of providing multilevel secure operation on general-purpose systems by creating an indistinguishable environment from that provided by a physically distributed system: to the internal processes each partitions appears as it were a separate, isolated machine.

One of the main advantages of separation kernels is that they can increase the security of the whole system by denying the ability of the processes to interfere with any resource outside their assigned security partition.

To ensure that a separation kernel provides an appropriate level of protection, and presents all its desired properties, it should be possible for the kernel developer to demonstrate that the kernel functions properly, this can be achieved by using techniques such as code inspection or formal verification.

A verifiable secure kernel improves the trustworthiness of the entire system. Separation kernels should be small enough for mathematical verification. This principle removes all superfluous functionality from the kernel, removing with it the problems of complexity and potential security flaws.

The small size and relative simplicity of separation kernels makes them obvious candidates for formal verification thus increasing the trustworthiness of the entire system.

2.5.1 Common Criteria Certification

The Common Criteria (CC) provides a standardized framework (methodology, notation and syntax) to specify and verify security functional requirements to be met by IT systems and assurance measures applied to them, in their different stages of life-cycle.

Common Criteria originated from three standards: ITSEC [25], TCSEC [26] and CTCPEC [27]. The standard ITSEC is an European standard. The TCSEC standard formerly known as the Orange book (DoD 5200.28 Std) was a work of the NSA and NBS (now NIST). The CTCPEC standard was published in Canada.

CC consists of a regulatory framework, adopted as an international standard by ISO in 1999 (ISO 15408) for the security assessment. Common Criteria certification comes in seven predefined assurance packages, which are called Evaluation Assurance Levels (EAL). Table 2.1 offers a small description of each EAL level. As EAL level increases, time, resources and technical requirements needed to achieve the certification also increase.

EAL	Description	Target
EAL 1	Product functional assay	Demonstrate proper operation of the product by independent testing, reviewing the applicability of threats not public domain.
EAL 2	Product structural test	It adds more active cooperation of the developer in terms of product delivery and test results.
EAL3	Test methodical and product validation	In addition to the above, it provides high security guarantee in the product design phase.
EAL4	Methodical design, testing and product validation	To provide a high assurance of safety practices implementation in the development and testing phase of the product. Level up and products suitable for commercial environments.
EAL5	Semi-formal design and product testing	Provides high assurance rigorous application of safety practices in the phase of product development and testing at both the developer how independent.
EAL6	Semi - formally verified design and product testing	Demonstrates proficiency in asset protection against significant threats, where the value of the assets justifies the additional costs of development.
EAL7	Formal design and product testing verified	Provides maximum protection guarantee high-value assets, based on a formal analysis of the product development and testing level.

Table 2.1: Description of Objectives for Each EAL.

2.5.2 Separation Kernel Protection Profile

Common criteria establishes a common language were users can specify their security needs. With this language users can specify what functionality should have the products to meet their needs. User security needs are specified in documents called Protection Profiles.

In 2007, the U.S. National Security Information Security Agency (NSA) released a Separation Kernel Protection Profile (SKPP) [28], that contains a specification of the security requirements for separation kernels. The SKPP describes the fundamental properties of separation kernels and defines the security requirements for their under Common Criteria certification.

The SKPP defines a separation kernel as:

> *"Hardware and/or firmware and/or software mechanisms whose primary function is to establish, isolate and separate multiple partitions and control information flow between the subjects and exported resources allocated to those partitions".*

Although the fulfilment of SKPP requirements does not imply that the evaluated kernel is immediately certified under Common Criteria, the requirements defined by the document were designed to provide an assurance level EAL 6 augmented. Among the main requirements of this protection profile we can find the obligation to present a formal model that describes the functionality of the separation kernel.

Chapter 3

Rushby's Non Interference Formalization

In this chapter we describe the notion of Security (non-Interference) for separation Kernels originally formulated by John Rushby [7]. This chapter has been divided in two sections:

In this Section we present the notion of non-interference for transitive security policies. First, we formalize the system model proposed by Rushby. Next, we discuss the notion of security (non-interference) used to evaluate the security of this model. Later, we prove that the system model proposed by Rushby complies with the notion of non interference. Finally we consider the relation between non-interference and access control formulations, and we identify the "Rushby's reference monitor assumptions" that play a key role to ensure non-interference in access control systems.

3.1 Transitive Non-Interference

An information flow policy defines the types of information that a system can have, and how information can flow between these classes. An information flow policy can be used to express a multi-level security models.

The concept of Non-interference is a very abstract formalization aimed at solving the problem of confidentiality in computer systems. The definition of non-interference was introduced by Goguen and Meseguer in 1982 [29], and later refined by John Rushby [7]. According to the concept of non-interference. A security domain does not interfere with another if the actions executed in the first domain have no effect on what the second can see. In other words, it is as if the first security domain does not exist for the second.

3.1.1 System Model

Non-Interference security policy was defined by Rushby over a system modeled with a finite state automaton (state machine), composed by:

- States. A state contains all the values stored in the machine.
- Actions. An action can be seen as an "input" or "commands" supplied to the machine.
- Outputs. An output is the result returned by the system after feeding it with an action.

With an initial State $s0$ which represents the state of the machine before the execution of any action. In Isabelle/HOL, this model is formalized as follows:

fixes s0 :: ′State

The aforementioned state machine includes functions *step* and *output*, defined as follows:

fixes step :: ′Action => ′State => ′State
fixes output :: ′State => ′Action => ′Output

Please note that from this point on, the first letters of the alphabet $(a, b, ...)$ are used to denote actions, the letters $s, t, ..$ to represent states and the letters of the Greek alphabet $\alpha, \beta, ...$ to indicate sequences of actions.
Function *step*(a, s) returns the state of the machine after action a has been executed in a given state s. Function *output*(a, s) returns the output of the machine after applying the action a over the state s

We make use of an Isabelle/HOL parametrized type $'Domain$ to represent these security domains.
Function *dom* returns the domain associated with the action given as a parameter.

fixes dom :: ′Action => ′Domain

Figure 3.1 shows a representation of the system model proposed by John Rushby, as we can notice, the system is divided in several domains, each action feed in the system is related with a domain

Function *run* represents a sequence of actions applied to the initial state of the machine, and is equivalent to an execution trace.

primrec run :: ′Action list => ′State => ′State **where**

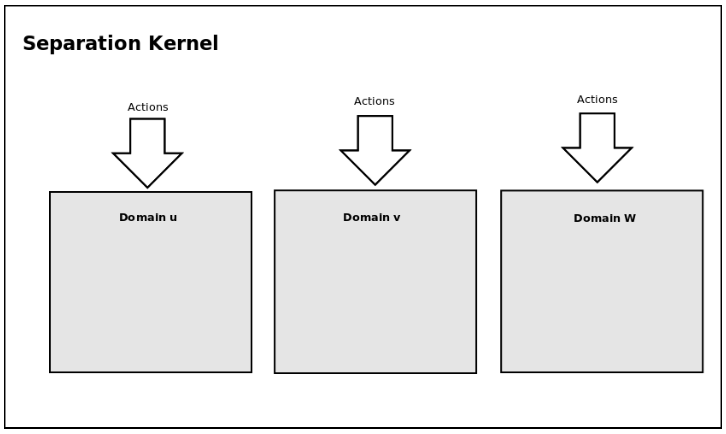

FIGURE 3.1: John Rushby's System Model

runEmpty : run [] s = s|
RunActions : run (a#α) = run α ∘ step a

Because of the frequent use of expressions like $output(run(s0, \alpha), a)$ and $run(s0, \alpha)$, Rushby defined functions *do* and *test* as a convenient way to abbreviate these forms:

fun do :: ′Action list => ′State **where**
  do (α) = run α s0
fun test :: ′Action list => ′Action => ′Output **where**
  test α a = output (do α) a

3.1.2 Security Policy

The core component of Rushby's Security Definition is the introduction of a policy that restricts the flow of information among the different security domains.

This security policy is denoted with two reflexive relations: $\rightsquigarrow$ called "Interference" and $\not\rightsquigarrow$ that denotes the complement relation "Non-Interference", formally defined as:

$$\not\rightsquigarrow = (\text{Domain} \times \text{Domain}) \setminus \rightsquigarrow$$

Where \ denotes set difference.

These relations can be formalized in Isabelle/HOL as follows:
consts interference :: "'Domain => 'Domain => bool" ("(_ $\rightsquigarrow$ _)")

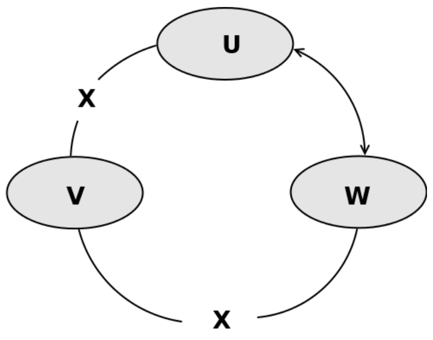

FIGURE 3.2: Information flow Security Policy.

syntax nonInterference :: "'Domain => 'Domain => bool" ("(_\ $\leadsto$ _)")
translations "u \ $\leadsto$ v"$\rightleftharpoons$"¬ (u $\leadsto$ v)"

Intuitively two domains u and v have an "Interference" relation ($u \leadsto v$) if u is allowed to interact with v (the information can flow from one domain to the other). If information is not allowed to flow between these domains they have a "non-Interference" relation ($u \not\leadsto v$).
Using the interference ($\leadsto$) and non-Interference ($\not\leadsto$) relations we can model information flow policies between different security domains.

Figure 3.2 illustrates a policy that can be expressed by:

$$u \leadsto w$$
$$w \leadsto u$$
$$v \not\leadsto u$$
$$v \not\leadsto w$$

This policy allows domains u and w to interact, and restricts any interaction of these domains with domain v.

3.1.3 Purge Function

For a domain v and an action sequence α, we define function $purge(\alpha, v)$. This function returns a subsequence of α, resulting of deleting all the actions of the domains that are not allowed to interact with v. In other words, it removes all actions of domains u that have a "non-Interference" relation with v ($u \not\leadsto v$).

primrec purge :: 'Action list => 'Domain => 'Action list **where**

PurgeemptyCase: purge [] v= []|

PurgeDefList: purge (a#α) v = (if (dom a) $\leadsto$ v then a#(purge α v) else purge α v)

3.1.4 Security Definition

John Rushby[7] defines a system as secure for the policy $\rightsquigarrow$ if:

definition secure::bool **where**
secure ≡ (∀ α a.(test α a) = (test (purge α (dom a)) a))

Under this definition, for a system to be considered as secure, no action performed in a security domain can affect the outputs of any other domain on which the interaction is not allowed.

3.1.5 Conditions to Guarantee Security

Rushby defines several conditions that individual systems have to satisfy to guarantee security.

A system is **view-partitioned** if, for each domain, there is an equivalence relation $\overset{u}{\sim}$ on the states.

consts equivalenceRelation :: "State => Domain => State => bool" ("(_ ~ _ ~ _)")

As we are defining an equivalence relation it must be transitive symmetric and reflexive. We can model these properties in Isabelle/HOL as follows:

definition eqvRelTransitive::bool **where**
eqvRelTransitive ≡ ∀ a b c u. (a ~ u ~ b) ∧ (b ~ u ~ c)
⟶ (a ~ u ~ c)

definition eqvRelSymmetric::bool **where**
eqvRelSymmetric ≡ ∀ s u t. (s ~ u ~ t) ⟶ (t ~ u ~ s)

definition eqvRelreflexive::bool **where**
eqvRelreflexive ≡ ∀ a u. (a ~ u ~ a)

The equivalence relations are said to be **output consistent** if:

definition outputConsistent ::bool **where**
outputConsistent ≡ (∀ s t a. ((s ~ (dom a) ~ t) ⟶ output s a = output t a))

Informally, two states are considered output consistent for domain u if their outputs are the same for any sequence of actions α applied on domain u. This property ensures that the outputs of two states s and t that seem equivalent to a domain u are really indistinguishable no matter what actions are performed on u.

Rushby proved [7] that given a policy $\rightsquigarrow$ and a view-partitioned, output consistent system such that

$$do(\alpha) \overset{u}{\sim} do(purge(\alpha, u)).$$

Then such system is secure for $\rightsquigarrow$

This lemma can be proven in Isabelle/HOL as follows:

```
  lemma lemma1:
    assumes 1: ∀ u α.((do α)~ u ~ (do (purge α u)))
    and     2: outputConsistent
  shows 3:secure
  proof –
    from assms show ?thesis
    unfolding outputConsistent-def secure-def test.simps
    by simp
  qed
```

3.1.6 Security for Single Step Transitions.

Rushby also defines the conditions the single step transitions have to fulfill in order to comply with the security definition described as follows:

Let a system be a view-partitioned system and $\rightsquigarrow$ a policy. We say that the system locally respects $\rightsquigarrow$ if:

```
  definition LocalRespects :: bool where
      LocalRespects ≡ (∀ a u s. ((dom a) ⋌↝ u) ⟶ (s ~ u ~ (step a s)))
```

System M is step consistent if:

```
  definition stepConsistent :: bool where
      stepConsistent ≡ ∀ a u s t . (s ~ u ~ t) ⟶ ((step a s) ~ u ~ (step a t))
```

"Locally respects" means that if an action is executed in a domain with non-Interference relation with u then domain u cannot distinguish the states before and after the execution of the action.

Finally **"step consistency"** requires that if two states are equal for a domain u and we execute the same action on both states, the outputs will still be same after the execution of the action.

After defining step consistency and locally respects rushby proves that any system that is step consistent and locally respects a policy is also secure.

Theorem 1 (Unwinding Theorem) Let $\rightsquigarrow$ be a policy in a view-partitioned system that is

- output consistent,
- step consistent, and
- locally respects $\rightsquigarrow$.

Then the system is secure for $\rightsquigarrow$.
This theorem can be formalized in Isabelle/HOL as follows:

theorem simple-noninterference:
  assumes outputConsistent:outputConsistent
  and stepConsistent: stepConsistent
  and LocalRespects: LocalRespects
  The following premises are included to include the properties of the equivalence relations
  and eqvRelSymmetric:eqvRelSymmetric
  and eqvRelTransitive:eqvRelTransitive
  and eqvRelreflexive:eqvRelreflexive
shows 5:secure

3.1.7 Rushby Model and Access Control Interpretation

The original Rushby model describes the conditions that an abstract system has to comply to be considered secure. However to consider non-Interference in actual systems he recognized the necessity to provide the system with a more elaborated state structure (*Structured State System*).
First we can consider that partitions will be composed of several memory locations called by Rushby as objects. Each one of these object stores a value.

We use two Isabelle/HOL parametrized types to represent this inner structure:

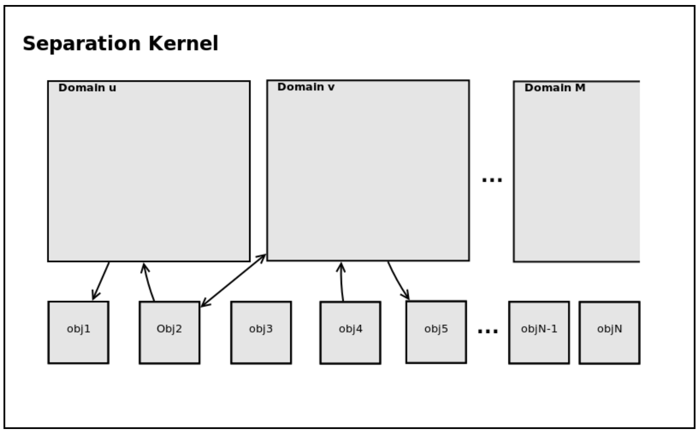

FIGURE 3.3: John Rushby's Structured State System

- 'Object
- 'Value

The system also defines a function *contents* which returns the value of a object in a given state, formalized as:

fixes contents :: ′State => ′Object => ′Value

3.1.8 Access Control Security Policy

The information flow is controlled with functions *observe* and *alter*, that determine the objects on which a given domain is capable to read or write. Function *observe* returns the set of objects that can be read by the domain given as a parameter.
This function can be modeled with the following expression:
fixes observe::′Domain => ′Object set

Function alter returns the set of objects that can be modified by the domain given as a parameter.

fixes alter::′Domain => ′Object set

Functions *alter* and *observe* are used to encode an "Access Matrix" used to represent the security policy of the system. Figure 3.3 depicts a graphical representation of a Structured State System with a security policy encoded by the Table 3.1.

D/O	Obj1	Obj2	Obj3	Obj4	Obj5
Domain u	w	r			
Domain v		rw		r	w

TABLE 3.1: John Rushby's Structured System Access Matrix

For a domain u and two states s and t, Rushby defines relation $\overset{u}{\sim}$ by the following formalization:

```
  assumes acRelation:
        ∀ s u t.(s ~ u ~ t) ⟷ (∀ n ∈ observe u. contents s n = contents t n)
```

It is easy to prove that $\overset{u}{\sim}$ have the Transitive, Symmetry and Reflexive properties from the equivalence relations:

```
lemma acRelationisTransitive:
      shows eqvRelTransitive
      proof –
      from acRelation
           show ?thesis
           unfolding eqvRelTransitive-def  by auto
      qed

lemma acRelationIsSymmetric:
       shows eqvRelSymmetric
     proof –
          from acRelation
          show ?thesis
          unfolding eqvRelSymmetric-def by auto
     qed

lemma  acRelationIsReflexive:
       shows eqvRelreflexive
     proof –
        from acRelation
          show ?thesis
        unfolding eqvRelreflexive-def by auto
     qed
```

3.1.9 Access Control Reference Monitors.

A State Structured System enforces a security policy when the following conditions are satisfied:

1.The outputs of an action a must depend only of the objects that can be read by the domain u , thus is required that:

definition(in AC) rma1 ::bool **where**
rma1 ≡ (∀ s t a. ((s ~ (dom a) ~ t) ⟶ output s a = output t a))

This condition can be seen as an interpretation of the "output consistency " property for state structured systems. This condition requires that for a domain u the values stored in the objects are the same for any sequence of actions α applied on domain u. This property ensures that the values of two states s and t that seem equivalent to a domain u are really indistinguishable no matter what actions are performed on u.

2.When an action is applied to the system, the values of all the modified objects must depend only of the values on the object on which $dom(a)$ has read access:

```
definition rma2 ::bool where
  rma2 ≡ (∀ s t a n.( (s ~ (dom a) ~ t)
                        ∧ (contents (step a s) n ≠ (contents s n)
                        ∨ contents (step a t) n ≠ (contents t n))
        ⟶ (contents (step a s) n = contents (step a t) n )))
```

This condition requires that if one object have the same value for two different states s and t for a domain u. If we execute an action a in both states, the value of the object shall be modified in the same way in the resulting states.

3. If an action a changes the value of object n then $dom(a)$ must have alter access to n:

```
definition rma3 ::bool where
  rma3 ≡ (∀ a n s .(contents (step a s) n ) ≠ (contents s n)
        ⟶ n ∈ alter (dom a))
```

Rushby names this three conditions the "Reference Monitor Assumptions" and states a theorem to relate the non-Interference relation and the Access Control System we just described.

Theorem: A system with structured state that satisfies the Reference Monitor Assumptions and the following two conditions.

- 1 $u \rightsquigarrow v \supset observe(u) \subseteq observe(v)$, and

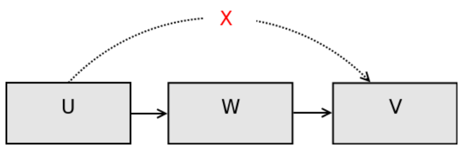

FIGURE 3.4: John Rushby Security Policy Representation

- 2 $n \in alter(u) \wedge n \in observe(v) \supset u \rightsquigarrow v$.

Is secure for the policy $\rightsquigarrow$.

This theorem can be formalized in Isabelle/HOL as follows:

theorem accessControlSystemIsSecure:
assumes 1:∀ u v. ((u ⇝ v) ⟶ observe u ⊆ observe v)
and 2:∀ u v n. ((n ∈ alter u) ∧ (n ∈ observe v)) ⟶ (u ⇝ v)
and rma1:rma1
and rma2:rma2
and rma3:rma3
shows secure

On this proof we have formalized Rushby intransitive non-interference model for access control systems. In the next Chapter we will study the GWV separation model.

3.2 Intransitive Non-Interference

The notion of basic non-Interference discussed in Section 2.5.2 is excessively restrictive in some cases; Imagine for example that we have a security policy like the one represented by Figure 3.4 and composed by the following assertions:

$$1.u \not\rightsquigarrow v$$
$$2.u \rightsquigarrow w$$
$$3.w \rightsquigarrow v$$

Expression 1 ($u \not\rightsquigarrow v$) forbids the communication between domains u and v and requires that there is no way for domain v to observe any change on the system caused by the actions applied to u. However, this condition is too restrictive on this case and would consider the assertions 2 and 3 as insecure, because the domain v could observe changes caused by actions of u trough w.

This kind of policies can be useful for applications such as the one illustrated by Figure 3.5. In this Figure the right arrows connecting the "Downgrader" domain represent intransitive information flows because, although the iformation can flow without restriction from the lower levels to the higher level, they need to use the "Downgrader" to send the information the other way around.

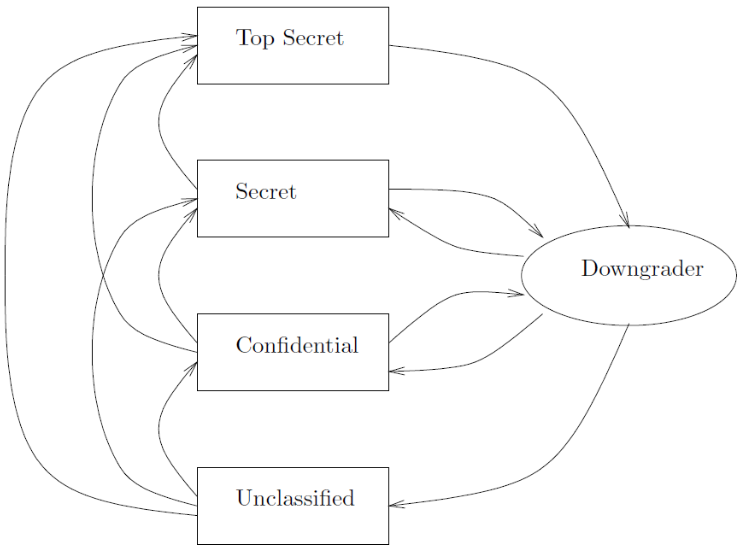

FIGURE 3.5: John Rushby [7]. Controlled Downgrading

To solve this kind of problems Haigh and Young [30] proposed a variant of intransitive non Interference based on an "intransitive purge" function wich was later refined by John Rushby [7].

Rushby intransitive policies are defined on the same state machine we described in Chapter 3 with some additional elements we describe in the next sections.

3.2.1 Sources Function

Rushby specify a *sources* function. *sources* receives a list of actions "α" and a domain U as parameters, and returns a set of domains. *sources* function is formally defined as follows:

```
primrec sources :: 'Action list => 'Domain  => 'Domain set where
  SourcesEmptyCase:sources [] u = {u}
 | SourcesDefList :sources (a#α) u = (if (∃ v. v ∈ (sources α u) ∧ ((dom a) ⇝ v) )
                 then (sources α u) ∪ {dom a}
                 else sources α u)
```

The goal of *sources* function is to determine the Security domains that are allowed to interfere with the domain given as a parameter. In other words $U \in sources(\alpha, V)$ either means that $U = V$ or that there is a subsequence of steps in α such that $U \rightsquigarrow D_1 \rightsquigarrow D_2 .... \rightsquigarrow D_n \rightsquigarrow V$. It is important to point out that the order on which the actions are executed is important for the sources function. Consider for example the following intransitive security policy:

$$A \rightsquigarrow B$$

$$B \rightsquigarrow C$$
$$A \not\rightsquigarrow C$$

Defined for a system with a set of actions $\{a, b\}$ such that $dom(a) = A$, $dom(b) = B$. If we apply *sources* function to the two step sequences $\alpha_1 = \{a, b\}$ and $\alpha_2 = \{b, a\}$ for the domain C we would notice that:

$$sources(\alpha_1, C) = \{A, B, C\} \text{ , while}$$
$$sources(\alpha_2, C) = \{B, C\}$$

The reason for this difference is that an α_2 action a is executed after action b. In intransitive policies like this C should not be able to notice changes from A that has not been followed by an action on B.

3.2.2 Ipurge Function

Using *sources* function it is possible to define an *ipurge* function. *ipurge* represents an intransitive version of *purge* function we described in Section 3.1.3.

ipurge can be defined as follows:

```
primrec ipurge :: 'Action list => 'Domain  => 'Action list where
  iPurgeEmptyCase:ipurge [] u= [] |
  iPurgeDefList  :ipurge (a#α) u = (if ((dom a) ∈ (sources (a # α) u )) then
                                     a#(ipurge α u)
                                    else ipurge α u)
```

The goal of *ipurge* function is to return a subsequence of α, resulting of deleting all the action that are not allowed to interact with u.

3.2.3 Security Definition for Intransitive Policies.

John Rushby defines an intransitive machine as secure for the policy $\rightsquigarrow$ if:

```
definition isecure::bool where
  isecure≡(∀ α a.(test α a)  = (test (ipurge α (dom a) ) a ))
```

Under this definition, for a system to be considered secure, no action performed in a security domain can affect the outputs of any other domain on which the interaction is not allowed by a direct interference relation $U \rightsquigarrow V$ or by a intransitive policy $(A \rightsquigarrow B, B \rightsquigarrow C)$.

Rushby showed that given an intransitive policy $\rightsquigarrow$ and a view-partitioned, output consistent system such that

$do(\alpha) \overset{u}{\sim} do(ipurge(\alpha, u))$.

Then such system is secure for $\rightsquigarrow$

This lemma can defined in Isabelle/HOL as follows:

```
lemma lemma2:
  assumes ∀ u α.((do α) ~ u ~ (do (ipurge α u)))
  and     outputConsistent
  shows isecure
 proof – {
    from assms show ?thesis
    unfolding outputConsistent-def isecure-def test.simps
    by auto}
 qed
```

Rusbhy defines a new equivalence relation. Given a set of security domains C relation and two states s and t $s \overset{C}{\approx} t$ is defined as follows:

```
definition iequivalenceRelation :: 'State => 'Domain set=> 'State => bool ((- ≈ - ≈ -))
  where iequivalenceRelation s c t ≡ ∀ u ∈ c.(s ~ u ~ t)
```

Informally the relation is true when the states s and t look the same for all the members of C.

3.2.4 Weakly step consistency

Intransitive policies requires a weaker version of the step consistency condition we described in Section 3.1.6. A system is weakly step consistent if the states that result from executing an action in equivalent states are equivalent.

Weakly step consistency can be defined as follows:

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \supset step(a, s) \overset{u}{\sim} step(a, t)$$

Isabelle/HOL

```
definition weaklyStepConsistent:: bool
  where weaklyStepConsistent ≡
     ∀ s u t a.(s ~ u ~ t) ∧ (s ~ (dom a) ~ t)⟶ ((step a s) ~ u ~(step a t))
```

Weak step consistency [31]: can be described informally as follows: If two states look equal to the evaluated partition (u), and also look the same to the partition executing the action ($dom(a)$), then resulting states must also look the same to the evaluated partition (u).

3.2.5 Security of Weak Step Consistent Systems for Intransitive policies

Rushby defines three lemmas to prove the security of weak step consistent systems.

A formal verification of the lemmas has been performed using Isabelle/HOL. The Isabelle/HOL file containing the following proofs cna be found in [URL]. The mechanically checked proof of Rushby's Lemmas 3 to 5 and Theorem 7 follow the original approach very closely.

Lemma 3 Let $\rightsquigarrow$ be a policy and M a weakly step consistent view-partitioned system, and locally respects $\rightsquigarrow$. Then

$$s \overset{sources(a\circ\alpha,u)}{\approx} t \supset step(s,a) \overset{sources(a\circ\alpha,u)}{\approx} step(s,t)$$

This lemma can be formalized in Isabelle/HOL as follows:

lemma *lemma3*:
  assumes *weaklyStepConsistent*:*weaklyStepConsistent*
  and *LocalRespects*:*LocalRespects*
  and *eqvRelSymmetric*:*eqvRelSymmetric*
  and *eqvRelTransitive*:*eqvRelTransitive*
  shows
  $\forall\ s\ a\ \alpha\ u\ t.(s \approx (sources\ (a\ \#\ \alpha)\ u\ ) \approx t) \longrightarrow ((step\ a\ s) \approx (sources\ \alpha\ u\ ) \approx (step\ a\ t))$

Rushby basically proves that in weakly step consistent system, if all the Domains that can modify a domain U are equal in two states s and t, They are going to remain equal after executing the same action in both states.

Lemma 4 Let $\rightsquigarrow$ be a policy and M a view-partitioned system that locally respects $\rightsquigarrow$. Then

$$dom(a) \notin sources(a \circ \alpha, u) \supset s \overset{sources(\alpha,u)}{\approx} step(a,s)$$

This lemma can be formalized in Isabelle/HOL as follows:

lemma *lemma4*:
assumes *LocalRespects*:*LocalRespects*
shows $\forall\ a\ \alpha\ s\ u.\ ((dom\ a) \notin sources(\ a\ \#\ \alpha\ )\ u) \longrightarrow (s \approx (sources\ \alpha\ u) \approx (step\ a\ s)\ )$

The purpose of this lemma is to show that if the domain of an action a is not allowed to alter a domain u, then the values of all the domains that alter u will remain the same after performing the action a, In other words domain U will be able to observe any action perform by domains that are not allowed to interact with it.

Lemma 5 Let $\rightsquigarrow$ be a policy and M a view-partitioned system which is weakly step consistent, and locally respects $\rightsquigarrow$. Then

$$s \overset{sources(\alpha,u)}{\approx} step(a,s) \supset run(\alpha,s) \overset{u}{\sim} run(ipurge(\alpha,u)t)$$

We formalized this lemma in Isabelle/HOL as follows:

lemma *lemma5*:
  assumes *weaklyStepConsistent*:*weaklyStepConsistent*
  and *LocalRespects*:*LocalRespects*
  and *eqvRelSymmetric*:*eqvRelSymmetric*
  and *eqvRelTransitive*:*eqvRelTransitive*
  shows $\forall\ \alpha\ s\ t\ u.(s \approx (sources\ \alpha\ u) \approx t) \longrightarrow ((run\ \alpha\ s) \sim u \sim (run\ (ipurge\ \alpha\ u)\ t))$

Using lemmas 3 to 5, John Rushby proves that the proposed intransitive model is secure

Theorem 7 (Unwinding Theorem for Intransitive Policies) Let $\rightsquigarrow$ be a policy and M a view-partitioned system that is

1. is output consistent,
2. weakly step consistent, and
3. locally respects $\rightsquigarrow$.

Then M is secure for $\rightsquigarrow$.

We formalized this theorem in Isabelle/HOL as follows:

theorem *Theorem7*:
  assumes *outputConsistent*:*outputConsistent*
  and *weaklyStepConsistent*:*weaklyStepConsistent*
  and *LocalRespects*:*LocalRespects*
  and *eqvRelSymmetric*:*eqvRelSymmetric*
  and *eqvRelTransitive*:*eqvRelTransitive*
  and *eqvRelreflexive*:*eqvRelreflexive*
shows *isecure*

3.2.6 Security of Intransitive Access Control Systems

Finally,John Rushby proved that the access control system described in Section 3.1.7 works for intransitive noninterference policies as well as for transitive ones.

Theorem 8 Let M be a system with structured state that satisfies the Reference Monitor Assumptions and the condition

$$n \in alter(u) \wedge n \in observe(v) \supset u \rightsquigarrow v$$

is secure for $\rightsquigarrow$.

The aforementioned theorem is translated in Isabelle/HOL as follows:

theorem *Theorem8*:
  assumes *1*:$\forall\ n\ u\ v.(n \in alter\ u \wedge n \in observe\ v) \longrightarrow (u \rightsquigarrow v)$
  and *rma1*:*rma1*

```
    and      rma2:rma2
    and      rma3:rma3
  shows isecure
```

Chapter 4

GWV Security Policy

In this chapter we describe the Greve, Wildling, Vanfleet (GWV) separation Kernel Formal Security Policy[6]. The original GWV model was formalized in ACL2. In this dissertation we translate the original formalization into Isabelle/HOL.

ACL2 and Isabelle/HOL support different styles of specification and proof. The ACL2 logic is unquantified and untyped, Isabelle/HOL is not only quantified but higher-order (it allows quantification over functions and predicates, not just individuals). However, we were able to successfully translate the original security definition and show that GWV model presents the properties of Exfiltration, Mediation and Infiltration.

4.1 System Model

The GWV formal security policy describes an abstract model of a Separation Kernel. Such model divides a state machine in several memory segments and ensures its separation by controlling the flow of information between them. Figure 4.1 depicts the main components of the GWV system model.

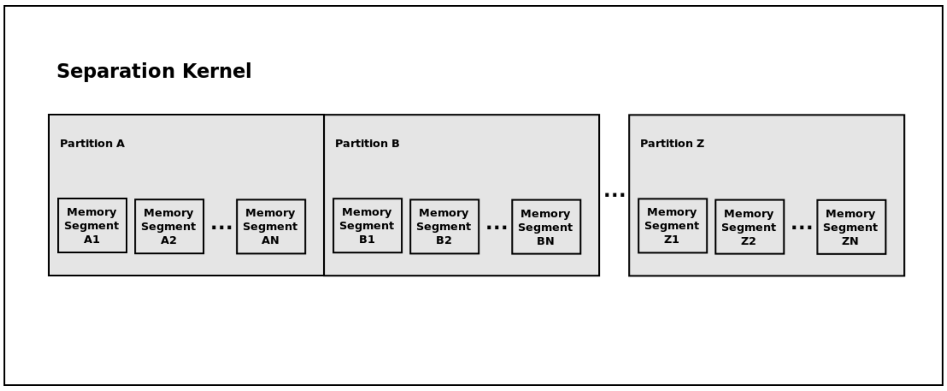

FIGURE 4.1: GWV Separation kernel main components

It is worth pointing out that states in the GWV model are not completely equivalent to the ones described in the Rushby model. In the Rushby model an action a is applied to a state s to change the current state. Whereas, the GWV model does not define a concept for actions. Thus, we consider that a GWV state is composed by a Rushby State and an Action.

As it was mentioned before the GWV security policy is defined over a machine that supports several partitions, with one of them defined as the current partition. Function *current* returns the current active partition given a memory state as parameter.

fixes current :: ($'$State, $'$Action) memorystate ⇒ $'$Domain

Each partition has several memory segments associated with it. Function *segs* takes a partition as parameter and returns the memory segments associated with it. (Note that function *segs* that segments associated to a partition are not modified with current memory state of the machine).

fixes segs :: $'$Domain ⇒ $'$Object set

The values stored in a memory segment are extracted by function *select*. *Select* takes two arguments: a segment and a machine memory state and returns all the values stored in the memory segment in the given memory state.

fixes select :: ($'$State, $'$Action) memorystate ⇒ $'$Object ⇒ $'$Value

The change between memory states is modeled with function *next*, that represents a computation on the state machine. This function takes as an argument a memory state and returns the new memory state of the machine after the execution of a single execution step.

fixes Next :: ($'$State, $'$Action) memorystate ⇒ ($'$State, $'$Action) memorystate

4.2 Clarification of Next and Current Functions

The way the states are defined within the model GVW can be a bit confusing at first. However, Alvez [8] offers a clarification about the correct interpretation of the *Current* and *Next* functions.

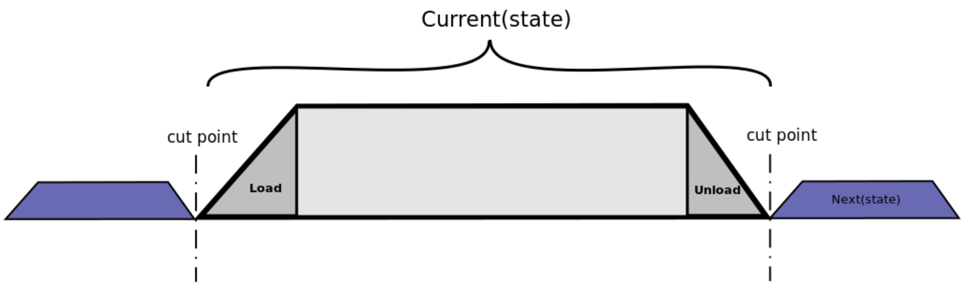

FIGURE 4.2: GWV state change

Figure 4.2 illustrates how the change of states is performed in the GWV model. As it can be seen, when the Next function is executed the kernel invokes, what the author calls a cut point. The cut point is the time the previous state of the kernel has already been saved in memory, but the values of the current state are not yet loaded into the kernel work area. It is at this time when the security policy is evaluated.
When the next function is executed, the separation kernel has the function to execute several steps. First, the values of the current state are loaded to the Kernel working area. Once the data is loaded , the action that corresponds to the current state is executed. When the task is completely executed the partition status is saved to memory. Finally the kernel work area is cleaned before loading the next state.

4.3 Security Policy (Separation)

The GWV Separation Kernel model enforces a communication policy between memory segments. The basic idea is to control the information flow between memory segments. This policy is modeled with function "Direct Interaction Allowed" *dia*, which takes as an argument a memory segment and returns a list of memory segments that are allowed to interact with it.

fixes dia :: 'Object ⇒ 'Object set

Figure 4.3 illustrates how GWV security policies are defined using *dia*. It's worth notice that the only segments which can effect segment A are those returned from *dia*, on this case:

$$dia\,(A) = \{B, E, D\}$$

Finally function *equals* is defined to test if the values of a given set of segments match for two different memory states. Please notice that this function is not present in the

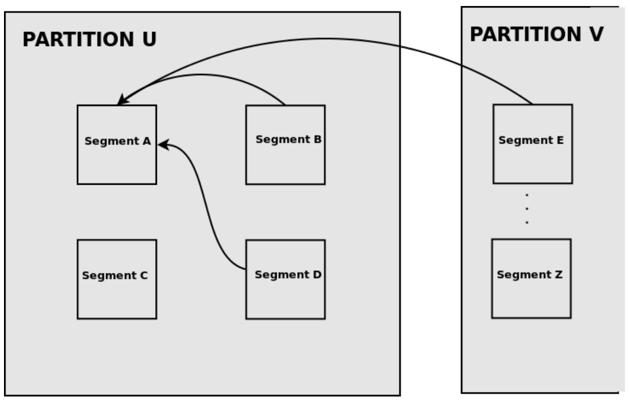

FIGURE 4.3: Information Flow Defined with *dia*

original model. However it greatly increases the readability of the separation lemma described later.

fun equals :: 'Object set ⇒ ('State, 'Action) memorystate ⇒ ('State, 'Action) memorystate ⇒ bool
  where
 equals A sa sb = (∀ a ∈ A. select sa a = select sb a)

The security policy requires that any arbitrary memory segment *seg* is only affected by the set of memory segments that are allowed to interact with *seg* and at the same time are associated with the current partition.

definition GWV-secure:: bool **where**
  GWV-secure ≡ ∀ st1 st2 seg.
          current st1 = current st2 ∧
          select st1 seg = select st2 seg ∧
          equals ((dia seg) ∩ (segs (current st1))) st1 st2
          ⟶ (select (Next st1) seg = select (Next st2) seg)

GWV security Policy states that for any given segment, *seg*, the values of the segment are only affected by memory segments that are allowed to communicate with it and that are part of the currently executing partition. If the separation assumption is preserved, then the only apparent way that a given segment could change is from interaction with segments that are allowed to affect it and that are in $dia(seg)$.

4.4 Relationship with other formalizations

It is proven [6] that any system that meets the requirements for the *separation* security policy also present three desired properties of the separation kernels, namely Exfiltration, Mediation and Infiltration. In this section, these properties are presented and proved under the assumption that *separation* holds:

4.4.1 Exfiltration.

Lemma *exfiltration* states that when a step is executed in the current partition, the memory segments can only be affected in a way that is consistent with the communication policy expressed with function *dia*.

```
lemma exfiltration:
  assumes GWV-secure
  shows ∀ st1 st2 seg. current st1 = current st2 ∧
                     select st1 seg = select st2 seg ∧
                     (dia seg) ∩ (segs (current st1)) = {}
                     ⟶ select (Next st1) seg = select (Next st2) seg
  proof –
  from assms show ?thesis unfolding GWV-secure-def by auto
  qed
```

For this lemma we assume that *dia* for a considered memory segment does not intersect with the segments of the current partition

4.4.2 Infiltration

Lemma *infiltration* states that the values in the current partition are not affected by the data in the segments associated with other partitions.

```
lemma infiltration:
assumes GWV-secure
shows ∀ st1 st2 seg. current st1 = current st2 ∧
                   seg ∈ segs (current st1) ∧
                   equals (segs (current st1)) st1 st2
                   ⟶ select (Next st1) seg = select (Next st2) seg
proof –
  from assms show ?thesis unfolding GWV-secure-def by auto
qed
```

4.4.3 Mediation

When a process is executed in the current partition, the effect on a segment does not depend on anything else than the segment's original value and the values of the current partition.

lemma mediation:
assumes GWV-secure
  shows ∀ st1 st2 seg.
        current st1 = current st2 ∧
        select st1 seg = select st2 seg ∧
        equals (segs (current st1)) st1 st2
        ⟶
        select (Next st1) seg = select (Next st2) seg
proof –
from assms **show** ?thesis **unfolding** GWV-secure-def **by** auto
qed

We have proven that our formalization of the GWV separation model holds the $exfiltration$, $infiltration$ and $mediation.$ properties In the next Chapter we will compare the models we just described to find their main similarities and differences.

Chapter 5

Non-Interference vs GWV

The structure of this chapter is illustrated in Figure 5.1. First we present a mapping between the concepts of the GWV and Rushby models, the mapping includes a definition of the Rushby reference monitor conditions in GWV terms. Next, we use this mapping to check if we can derive the original reference monitor definitions from the GWV versions. Later, we prove that a system with the GWV reference monitors complies with the non-Interference security definition. After that, we show that GWV security definition implies weakly step consistency.

Once we have showed that GWV secure implies weakly step consistency we can use this proof to show that a GWV secure system can be Rushby secure for the transitive and intrasitive Rushby security definitions.

Finally, we show that Rushby's step consitency (transitive model) implies GWV security (with action equality), but Rushby weak step consistency (intransitive model) only implies one of the properties of GWV security known as *Mediation*.

5.1 Mapping Between GWV and Rushby Concepts

The first problem while trying to compare "GWV" and Rushby basic "non-Interference" is that the concepts and components of the two models are not equivalent. In this section we try to find the similarities between these elements and map these equivalences. Table 6.1 presents the mappings between GWV and Rushby functions.

One of the most important similarities between the two models is the way the system is divided. The two models provide separation between processes on different security levels. However, such division is achieved by defining *Partitions* in GWV and security *Domains* in Rushby model. For the rest of this document we consider these concepts as equivalent.

We define function *GWVoutput-f* which takes a GWV memory state as input and returns the output of that state.

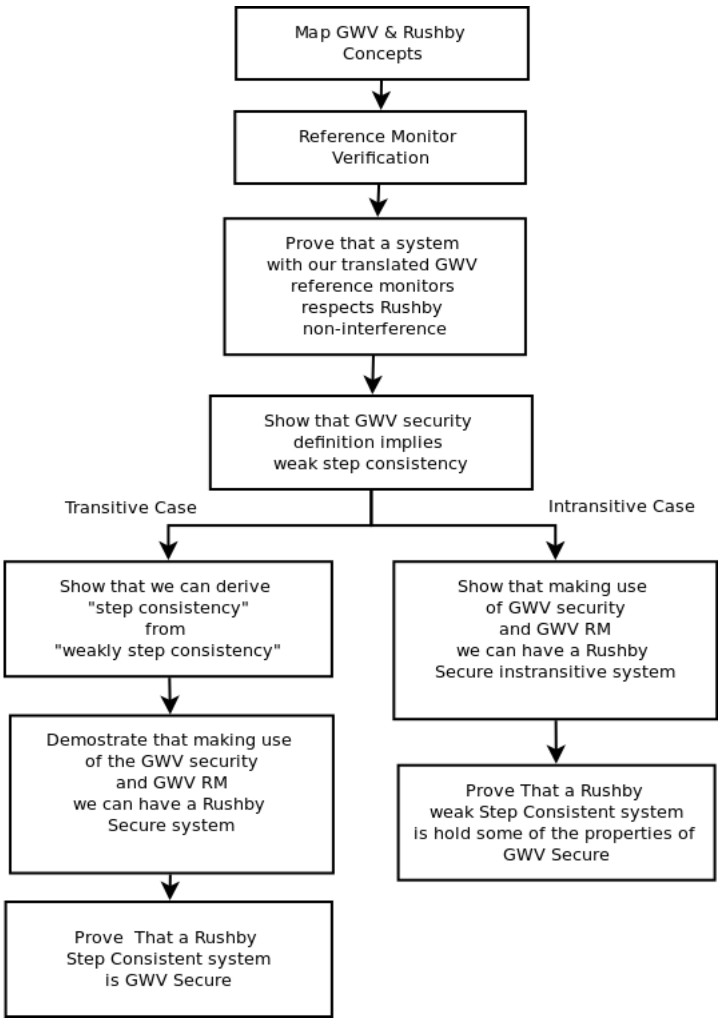
Map GWV & Rushby Concepts
Reference Monitor Verification
Prove that a system with our translated GWV reference monitors respects Rushby non-interference
Show that GWV security definition implies weak step consistency
Transitive Case
Intransitive Case
Show that we can derive "step consistency" from "weakly step consistency"
Demostrate that making use of the GWV security and GWV RM we can have a Rushby Secure system
Prove That a Rushby Step Consistent system is GWV Secure
Show that making use of GWV security and GWV RM we can have a Rushby Secure instransitive system
Prove That a Rushby weak Step Consistent system is hold some of the properties of GWV Secure

FIGURE 5.1: Overview of the proofs in Chapter 5

fixes GWVoutput-f :: ('State, 'Action) memorystate => 'Output

GWV does not define an interference relation at the level of partitions. It defines *dia*, which returns pairs of interfering segments. We now provide a definition of an interference relation in the world of GWV.

definition GWVia :: 'Domain => 'Domain => bool **where**
GWVia u v ≡ (∃ s s'. s ∈ segs v ∧ s' ∈ dia s ∧ s' ∈ segs u)

We define the mapping between *GWVia* and Rushby's interference relation. This definition basically reads, there is an interference from u to v, if there are two objects s and s' such that s is a segment of v, s' is a segment of u, and s' is allowed to directly interfere with s. In other words, two partitions have an interference relation if they have interfering segments.

definition intMapping:: bool **where**
intMapping ≡ ∀ u v. (u ⇝ v) ⟷ GWVia u v

GWV	Rushby	Comments
('State, 'Action) memorystate	state	GWV states are not equivalent to Ruhsby states. States in GWV contain the Rushby state and the action to be executed.
Partition	Domain	
Segment	Object	
GWVoutput_ f	Output	We define a new function GWVoutput_ f to represent the output of the machine in a given state.
('State, 'Action) memorystate	action	GWV states contains the action that is going to be performed when the next function is called.
$current(segment)$	$dom(dimension)$	
$segs(partition)$	$alter(domain)$	
$select(State, Segment)$	$contents(State, Object)$	
$next(State)$	$step(State)$	
$segs(u)$	$observe(u)$	
$GWVvpeq(s, u, t)$	$s \overset{u}{\sim} t$	We define the $GWVvpeq$ function to represent the "view partitioned" equivalence relation.
$GWVia(u, v)$	$u \rightsquigarrow v$	$GWVia$ function provides a definition of interference relation in the world of GWV.

TABLE 5.1: Mapping between the elements of GWV and Rushby models

In Rushby's model, the equivalence relation "view partitioned" plays a central role. There is no similar relation defined in the context of GWV. Therefore we define such a relation with function *GWVveq*

```
definition GWVvpeq :: ('State, 'Action) memorystate ⇒
                 'Domain ⇒ ('State, 'Action) memorystate ⇒ bool
   where
  GWVvpeq s p t ≡
                 (∀ seg ∈ segs p. select s seg = select t seg)
                  ∧ Action s = Action t
```

According to John Rushby, function *"observe(u) is the set of locations whose values can be observed by domain u."* [7]. Whereas in the GWV model, a *partition* is able to read from all the segments segments assigned to it.

The new definition *observeMapping* maps the Rushby's observe function in the terms of GWV model.

```
definition observeMapping:: bool where observeMapping ≡ ∀ u. observe u = segs u
```

In the GWV context a partition can modify ("*alter*") all its assigned partitions.

```
definition alterMapping:: bool where alterMapping ≡ ∀ u. alter u = segs u
```

We assume that the values stored in a segment *n* in the GWV state *s* are equal to the values stored in a similar state in the Rushby model.

```
definition selectMapping:: bool where
   selectMapping ≡ (∀ n memoryState.
                 select memoryState n = contents (State memoryState) n)
```

A mapping between the Rushby *output* and the GWV *GWVoutput-f* functions is defined as follows:

```
definition outputMapping::bool where
         outputMapping ≡ (∀ memoryState.(GWVoutput-f memoryState) =
                 (output (State memoryState) (Action memoryState)))
```

The active partition returned by function *current* for a state *s*, is equal to the domain returned by the Rusby function *dom* for the state *s*

```
definition currentMapping::bool where
         currentMapping ≡ ∀ memoryState. current memoryState
                          = dom (Action memoryState)
```

The state obtained after executing the GWV function *next* on a state *s* will be equal to the one obtained from the Rushby *step* function for the action contained within the GWV state.

```
definition stepMapping:: bool where
        stepMapping ≡ ∀ s. State (Next s) = step (Action s) (State s)
```

5.2 Equivalence Relation Mapping

We define a equivalent to the Rushby state equivalence relation in GWV terms as follows:

```
definition GWVvpeqtoRelation::bool where
        GWVvpeqtoRelation ≡ ∀ s u t. GWVvpeq s u t
                     ⟷  ((State s) ~  u ~ (State t))
```

Rushby's $s \overset{u}{\sim} t$ relation denotes an equivalence relation. We prove that the GWV mapping holds the *transitivity*, *symmetry* and *reflexivity* properties

```
definition GWVvpeq-transitive::bool where
      GWVvpeq-transitive  ≡ ∀ a b c u. (GWVvpeq a u b) ∧ (GWVvpeq b u c)
       ⟶ (GWVvpeq a u c)

lemma GWVvpeqIstransitive:
  shows GWVvpeq-transitive
  proof –
      show ?thesis
      unfolding GWVvpeq-def GWVvpeq-transitive-def  by auto
  qed

definition GWVvpeq-symmetric::bool where
    GWVvpeq-symmetric  ≡ ∀ a b u. (GWVvpeq a u b) ⟶ (GWVvpeq b u a)

lemma GWVvpeqIsSymmetric:
  shows GWVvpeq-symmetric
proof –
     show ?thesis unfolding GWVvpeq-def GWVvpeq-symmetric-def by auto
qed

definition GWVvpeq-reflexive::bool where
    GWVvpeq-reflexive ≡ ∀ a u. (GWVvpeq a u a)

lemma GWVvpeqIsReflexive:
  shows GWVvpeq-reflexive
proof –
```

```
        show ?thesis unfolding GWVvpeq-def GWVvpeq-reflexive-def by auto
    qed
```

5.3 Reference Monitor Mappings

Now that we have mapped all the Rushby concepts to GWV concepts, we can start phrasing Rushby's *accessControlSystemIsSecure* obligations and reference monitor conditions in terms of GWV concepts.

Theorem *accessControlSystemIsSecure* have 3 reference monitor obligations to be discharged.

The first reference monitor:

$$s \overset{dom(a)}{\sim} t \supset output(s,a) = output(t,a)$$

Can be expressed in GWV terms, by simply replacing $GWVia.$ as follows:

```
    definition GWVrma1::bool where
        GWVrma1 ≡ ∀ s t. (GWVvpeq s (current s) t)
                    ⟶ GWVoutput-f s = GWVoutput-f t
```

The second reference monitor defined as:

$$\begin{aligned} s \overset{dom(a)}{\sim} t \wedge (&contents(step(s,a),n) \neq contents(s,n) \\ &\vee contents(step(t,a),n) \neq contents(t,n)) \\ \supset contents(&step(s,a),n) = contents(step(t,a),n). \end{aligned}$$

Can be expressed in GWV terms with the following definition:

```
    definition GWVrma2::bool where
        GWVrma2 ≡ ∀ s t n u.((GWVvpeq s u t)
                ∧( select (Next s) n ≠ select s n
                ∨ select (Next t) n ≠ select t n))
                ⟶ select (Next s) n = select (Next t) n
```

Finally the third reference monitor:

$$contents(step(a,s),n) \neq contents(step(a,t),n) \supset n \in alter(dom(a)).$$

can be formalized in the GWV world as follows:

```
    definition GWVrma3::bool where
        GWVrma3 ≡ ∀ n s. select (Next s) n ≠ select s n ⟶ n ∈ segs (current s)
```

The *accessControlSystemIsSecure* theorem additionally defines two extra conditions that systems must comply to be considered secure:

- 1 $u \leadsto v \supset observe(u) \subseteq observe(v)$, and
- 2 $n \in alter(u) \wedge n \in observe(v) \supset u \leadsto v$.

The aforementioned conditions are expressed with the following definitions:

```
definition GWVcond1::bool where
      GWVcond1 ≡ ∀ u v. GWVia u v ⟶  observe v ⊇ observe u

definition GWVcond2::bool where
      GWVcond2 ≡ ∀ u v n.(n ∈ alter u) ∧ (n ∈ observe v) ⟶ GWVia u v
```

5.4 Reference Monitor Verification

In this section we verify the mappings between GWV and Rushby reference monitor obligations, proving that the new GWV definitions can be translated into the ones defined by Rushby.

First we prove the translation of the proof obligation 1. The goal of this lemma is to demostrate that if a domain "u" can alter the values of other domain v , then that all objects observable by u are also observable by v

```
lemma proofobligation1Mapping:
    assumes GWVcond1:GWVcond1
    and intMapping: intMapping
  shows ∀ u v. (u ⇝ v) ⟶  observe v ⊇ observe u
proof -
  from GWVcond1 and intMapping show ?thesis
    unfolding intMapping-def  GWVcond1-def by metis
```

This lemma is proven directly from the definition of the first condition and the mappings

```
qed
```

Now we verify the mapping for proof obligation 2. The purpose of this lemma is to prove that if an object can be written by a domain u and read by a domain v then we have an interference relation $u \leadsto v$ between u and v

```
lemma proofobligation2Mapping:
      assumes GWVcond2:GWVcond2
      and intMapping:intMapping
shows
      ∀ u v n. ((n ∈ alter u) ∧ (n ∈ observe v)) ⟶ (u ⇝ v)
  proof -
```

```
      We use the mappings to replace the GWV terms with the Rushby ones
      thm GWVcond2-def
        from GWVcond2   and GWVia-def  and intMapping show
          ?thesis
          unfolding intMapping-def GWVcond2-def by auto
          The lemma is proven from the mapping and the definition of GWVcond2
    qed
```

The next step is to prove the GWV reference monitor for the Rushby proof obligation 3. We have separated this proof in two lemmas:

lemma *proof-obligation3* proves that *GWVrma1* can be mapped to an intermediate expression similar to *rma1* but using GWV states.

```
  lemma GWVproofobligation3:
    assumes GWVrma1:GWVrma1
    and     outputMapping:outputMapping
    and     currentMapping:currentMapping
    and     GWVvpeqtoRelation:GWVvpeqtoRelation
   shows     ∀ s t. ((State s) ~ dom (Action s) ~ (State t))
          ⟶ output (State s) (Action s) = output (State t) (Action s)
     proof –
      {
        fix s t::('State, 'Action) memorystate

        We make use of the definition of GWVrma1 and the mappings to
        get the intermediate expression.
        from  assms
        have  ((State s) ~ (dom (Action s)) ~ (State t))⟶
               output (State s) (Action s) = output (State t) (Action s)
        unfolding GWVrma1-def GWVvpeqtoRelation-def  currentMapping-def
        outputMapping-def GWVvpeq-def by metis
      }
     then show ?thesis by (metis select-convs(1) select-convs(2))
  qed
```

The second lemma *GWVrma1ToRma1* invokes *proof-obligation3* to prove that *GWVrma1* implies the First Rushby's reference monitor (*rma1*) making use of the new intermediate expression.

```
 lemma GWVrma1ToRma1:
   assumes GWVrma1:GWVrma1
```

```
  and     outputMapping:outputMapping
  and     currentMapping:currentMapping
  and     GWVvpeqtoRelation:GWVvpeqtoRelation
shows rma1
```

rma1 states that two equivalent states have the same output

```
proof –
{
  from assms and GWVproofobligation3 have
        ∀ (s::'State) (t::'State) (a::'Action).

              ((State (|State = s, Action = a|)) ~
              dom (Action (|State = s, Action = a|)) ~
              (State (|State = t, Action = a|)))

          ⟶ output (State (|State = s, Action = a|)) (Action (|State = s, Action = a|))
            = output (State (|State = t, Action = a|)) (Action (|State = s, Action = a|))
  by metis
  }
```

The proof directly follows from the intermediate definition.

```
    then show rma1
    unfolding rma1-def
    by (metis  select-convs(1) select-convs(2))
  qed
```

In a similar way, we can prove that our new definition implies the second Rushby's reference monitor, separating this proof in two lemmas:

lemma *proof-obligation4* proves that *GWVrma2* can be mapped to an intermediate expression similar to *rma2* but using GWV states.

```
lemma proof-obligation4:
    assumes GWVrma2:GWVrma2
    and     selectMapping:selectMapping
    and     stepMapping: stepMapping
    and    GWVvpeqtoRelation:GWVvpeqtoRelation

shows (∀ (s::('State, 'Action) memorystate) (t::('State, 'Action) memorystate)
n.(  ((State s) ~  (dom (Action s))~ (State t))
          ∧ (contents (step (Action s) (State s)) n ≠ contents (State s) n
          ∨  contents (step (Action t) (State t)) n ≠ contents (State t) n)
         ⟶ contents (step (Action s) (State s)) n
```

```
          = contents (step (Action t) (State t)) n)
         )
  proof –
    {
      fix s t::('State, 'Action) memorystate
      fix n :: 'Object

      from assms have
      (((State s) ~ (dom (Action s)) ~ (State t)) ∧
         (contents (step (Action s) (State s)) n ≠ contents (State s) n
        ∨  contents (step (Action t) (State t)) n ≠ contents (State t) n))
       ⟶
        contents (step (Action s) (State s)) n
       = contents (step (Action t) (State t)) n
```

We prove this lemma simply by using the definition of the *GWVrma2* and the mappings

```
       unfolding GWVrma2-def selectMapping-def  stepMapping-def
                 GWVvpeqtoRelation-def
      by auto

    }
   then show ?thesis by auto
 qed
```

The second lemma *GWVrma2ToRma2* uses *proof-obligation4* to show that *GWVrma2* implies the second Rushby's reference monitor (*rma2*) making use of the new intermediate expression.

```
lemma GWVrma2ToRma2:
    assumes GWVrma2:GWVrma2
    and     selectMapping:selectMapping
    and     stepMapping: stepMapping
    and    GWVvpeqtoRelation:GWVvpeqtoRelation
  shows  rma2

  proof –
```

This expression will be used to transform the GWVrma2 into the conclusion of *proof-obligation4* lemma.

```
   from assms and  proof-obligation4 have
```

```
(∀ (s::'State) (t::'State) (a::'Action) (n::'Object).

      ((State (|State = s, Action = a|) ) ~
       (dom (Action (|State = s, Action = a|)))~
       (State (|State = t, Action = a|)))

      ∧ (contents (step (Action (|State = s, Action = a|))
                        (State (|State = s, Action = a|))) n
          ≠ contents (State (|State = s, Action = a|)) n
      ∨  contents (step (Action (|State = t, Action = a|))
                        (State (|State = t, Action = a|))) n
          ≠ contents (State (|State = t, Action = a|)) n)
      ⟶ contents (step (Action (|State = s, Action = a|))
                        (State (|State = s, Action = a|))) n
       = contents (step (Action (|State = t, Action = a|))
                        (State (|State = t, Action = a|))) n
      )
 unfolding selectMapping-def stepMapping-def by metis
```

Then the proof falls directly from this transformation and the mappings

from this **and** selectMapping **and** stepMapping **and** select-convs(1) **and** select-convs(2)

```
 show ?thesis
 unfolding currentMapping-def  selectMapping-def  rma2-def  stepMapping-def
 by auto
qed
```

Finally we prove that the mapping for the Third reference monitor is correct, by making use of a similar strategy to the previous two monitors.

lemma *proof-obligation5* shows that *GWVrma3* can be mapped to an intermediate expression similar to *rma3* with GWV states.

```
lemma proof-obligation5:
    assumes GWVrma3:GWVrma3
    and     selectMapping:selectMapping
    and     stepMapping:stepMapping
    and     alterMapping:alterMapping
    and     currentMapping:currentMapping
  shows  ∀ a n s. contents (step (Action s) (State s)) n ≠ contents (State s) n
                ⟶ n ∈ alter (dom (Action s))
    proof –
    {
        fix s t::('State, 'Action) memorystate
        fix n :: 'Object

       We prove the intermediate expression by making use of the mappings and the

        GWVrma3 definition
       from GWVrma3 and selectMapping and stepMapping and currentMapping
        and alterMapping
        have
        contents (step (Action s) (State s)) n ≠ contents (State s) n
            ⟶  n ∈ alter (dom (Action s))
     unfolding  GWVrma3-def selectMapping-def stepMapping-def currentMapping-def
        alterMapping-def
        by (metis (full-types))

     }
    then show ?thesis by (metis (mono-tags) select-convs(1) select-convs(2))
  qed
```

The lemma *GWVrma3ToRma3* uses *proof-obligation5* to show that *GWVrma3* implies the last Rushby's reference monitor (*rma3*).

```
 lemma GWVrma3ToRma3:
    assumes GWVrma3:GWVrma3
    and     selectMapping:selectMapping
    and     stepMapping:stepMapping
    and     alterMapping:alterMapping
    and     currentMapping:currentMapping
  shows rma3
    proof –
    {
     We define a transformation expression to convert GWVrma3 into rma3
     by invoking the proof-obligation5 lemma.
     from assms and  proof-obligation5
     have
     ∀ (s::'State) (a::'Action) (n::'Object).
        contents (step (Action (|State = s, Action = a|))
                      (State (|State = s, Action = a|))) n
```

```
          ≠ contents (State (|State = s, Action = a|)) n
          ⟶ n ∈ alter (dom (Action (|State = s, Action = a|)))
          unfolding GWVrma3-def
          by metis
          }
```

The proof directly follows from the mapping and this transformation expression

```
      from this show ?thesis unfolding rma3-def
      by (metis  select-convs(1) select-convs(2))
  qed
```

We now show that a system that satisfies GWV reference monitors we just formulated can be considered secure under Rushby's definition.

```
  lemma GWVmapsAreRushbySecure:
    We use the definitions for the GWV reference monitors
    assumes GWVrma1:GWVrma1
    and GWVrma2:GWVrma2
    and GWVrma3
    and GWVcond1
    and GWVcond2
    And we use the mappings
        and     selectMapping:selectMapping
        and     stepMapping:stepMapping
        and     outputMapping:outputMapping
        and     alterMapping:alterMapping
        and     currentMapping:currentMapping
        and     GWVvpeqtoRelation:GWVvpeqtoRelation
        and     GWVvpeqtoRelation:GWVvpeqtoRelation
        and     intMapping
    shows secure
    proof –
        from assms
        and GWVrma1ToRma1
        and GWVrma2ToRma2
        and GWVrma3ToRma3
        and proofobligation1Mapping
        and proofobligation2Mapping
```

We just invoke the Rushby's *accessControlSystemIsSecure* lemma to complete the proof

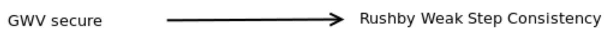

FIGURE 5.2: Relation Between GWV secure and Weakly Step Consistency definitions.

```
    and accessControlSystemIsSecure show secure by auto
qed
```

5.5 Proof of GWV Step Consistency

In the previous section we shown that a system modeled with with GWV components can be considered secure under Rushby "non-Interference" definition. Figure 5.2 illustrates the relation between the GWV security and Rushby's step consistency definitions. We show that the GWV definition of security *GWV-secure* implies Rushby "Weak Step Consistency" (*weaklyStepConsistent*).

```
lemma GWV-WeakStepConsitent:
  assumes GWV-secure:GWV-secure
  and     currentMapping:currentMapping
  and    ActionStep:∀ s t u . GWVvpeq s u t ⟶ Action (Next s) = Action ( Next t)
shows ∀ s u t. GWVvpeq s u t ∧ GWVvpeq s (current s) t ⟶ GWVvpeq (Next s) u (Next t)
 proof (auto)
 {
  fix s t::('State, 'Action) memorystate
  fix u::'Domain
  assume uEquiv:GWVvpeq s u t
  assume currentEquiv:GWVvpeq s (current s) t
  show GWVvpeq (Next s) u (Next t)

  unfolding GWVvpeq-def
   proof {
     From the definition of GWVvpeq we need to prove 2 goals:
GWVvpeq (Next s) u (Next t)  1. ∀ seg∈segs u. select (Next s) seg = select (Next t) seg  2. Action (Next s) = Action (Next t)
       show ∀ n ∈segs u. select (Next s) n = select (Next t) n
          proof {
           fix n::'Object
           assume observe:n ∈ segs u
```

```
        show select (Next s) n = select (Next t) n
         this is the conclusion of the GWV-secure definition
          proof - {
           We get each one of the premises of GWV-secure
              from GWVvpeq relation and the mappings.

           from currentEquiv and currentMapping have sec1:current s = current t
            unfolding GWVvpeq-def currentMapping-def by auto

            from currentEquiv and observe have sec2:select t n = select t n
            unfolding GWVvpeq-def by auto

            from currentEquiv have sec3:Action s = Action t
            unfolding GWVvpeq-def by auto

            from currentEquiv and observe
            have sec4:equals (dia n  ∩ segs (current s)) s t
            unfolding  GWVvpeq-def
                by auto
            the proof of this goal, then directly follows the above facts and
              $GWV_secure$
            from sec1 sec2 sec3 sec4 and GWV-secure
            and observe currentEquiv GWVvpeq-def
            show  select (Next s) n = select (Next t) n
            unfolding GWV-secure-def  GWVvpeq-def
            by (metis GWVvpeq-def uEquiv)

           } qed
        } qed

      next
      The proof of the second goal is trivial from our ActionStep assumption.
      from currentEquiv and ActionStep show Action (Next s) = Action (Next t)
      unfolding GWVvpeq-def by auto
    }
   qed
  }
 qed
```

Now we prove that Weak Step Consistency (under GWV terms) can also be transformed into an expression under Rusbhy terms.

lemma GWV-RusbhyWeakStepConsitent:
  assumes GWV-secure: GWV-secure
  and currentMapping:currentMapping
  and ActionStep:∀ s t u . GWVvpeq s u t ⟶ Action (Next s) = Action (Next t)
  and GWVvpeqtoRelation:GWVvpeqtoRelation
  and stepMapping:stepMapping
shows weaklyStepConsistent

5.5.1 "Strengthening" Weakly Step Consistency.

Figure 5.3 shows the relation between Rushby's "step consistency" and "weak step consistency" definitions. As the following lemma shows is trivial to prove that step consistency implies "weakly step consistency".

lemma
assumes stepConsistent
shows weaklyStepConsistent
proof –
  from assms **show** ?thesis
  unfolding stepConsistent-def **and** weaklyStepConsistent-def
  by auto
qed

To show the reverse implication we propose the following lemma.

Lemma: A Weakly Step Consistent system that satisfies the third Reference Monitor Assumption we described in Section 3.1.9, and the following two conditions (originally proposed by Rushby).

- 1 $u \rightsquigarrow v \supset observe(u) \subseteq observe(v)$, and
- 2 $n \in alter(u) \land n \in observe(v) \supset u \rightsquigarrow v$.

is also step Consistent.

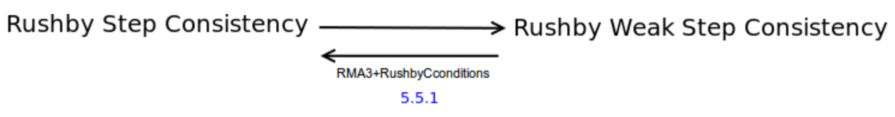

Figure 5.3: Relation Between Weakly Step Consistency and Step Consistency definitions.

In the next proof we demonstrate this lemma by separating the proof in two steps; first we prove the case when there is no change in the values of the state after executing the a step in the state machine, the proof for this case is trivial and fall directly from the equality properties of the state equivalence relation. For the second case we have a change in the values of the evaluated memory segment, for this case we use the third reference monitor $\forall\ a\ n\ s\ .(contents\ (step\ a\ s)\ n\ ) \neq (contents\ s\ n) \longrightarrow n \in alter\ (dom\ a)\}$" to prove that having this additional condition we can ansure step consistency.

```
lemma weakStepConsistencyToStepConsistent:
    assumes 1:∀ u v. ((u ⇝ v) ⟶ observe u ⊆ observe v)
    and     2:∀ u v n. ((n ∈ alter u) ∧ (n ∈ observe v)) ⟶ (u ⇝ v)
    and     weaklyStepConsistent:weaklyStepConsistent
    and     rma3:rma3
    shows stepConsistent
  proof –{
        fix a::′Action
        fix s t::′State
        fix u::′Domain
        fix n::′Object

        —To prove step consistency we can rewrite its definition as:
        have ((s ~ u ~ t) ∧ (n ∈ observe u)) ⟶
              (contents (step a s) n = contents (step a t) n)

        To prove this expression we have two cases to consider:
        proof (case-tac contents (step a t) n = contents t n
                   ∧ contents (step a s) n = contents s n){
          {
             {
            —for the first case:
               contents (step a t) n = contents t n
               ∧ contents (step a s) n = contents s n,
             we have contents s n = contents t n then the proof of this case is trivial

            assume case1:contents (step a t) n = contents t n
                        ∧ contents (step a s) n = contents s n
             show ((s ~ u ~ t) ∧ (n ∈ observe u))
                   ⟶ (contents (step a s) n = contents (step a t) n)
             proof {
```

```
      assume hyp:(s ~ u ~ t) ∧ (n ∈ observe u)
        from hyp and acRelation have contents s n = contents t n by auto
        from this and case1
        show (contents (step a s) n) =(contents (step a t) n) by simp
      }qed
  }
next
  {
    —In the second case we have a change in any of the states after a step
      is executed
    assume case2:¬ (contents (step a t) n = contents t n
              ∧ contents (step a s) n = contents s n)
    show  ((s ~ u ~ t) ∧ (n ∈ observe u))
        ⟶ contents (step a s) n = contents (step a t) n
    proof  {
        assume hyp:(s ~ u ~ t) ∧ (n ∈ observe u)
        —This case can be rewritten as follows for simplicity:
          (contents (step a t) n ≠ contents t n)
          ∨ (contents (step a s) n ≠ contents s n)
        from case2 have
        case2Rew:(contents (step a t) n ≠ contents t n)
              ∨ (contents (step a s) n ≠ contents s n)
        by simp

        —From the third reference monitor we have n ∈ alter (dom a)
        from this and rma3 have  n ∈ alter (dom a)
        unfolding rma3-def by auto

        —Using the first and second conditions of the theorem we
        have that : observe (dom a) ⊆ observe u
        from this  and 1 and 2 and hyp
        have observe (dom a) ⊆ observe u by metis

        —Then we have that (s ~ u ~ t) implies s ~(dom a) ~ t
        from this and acRelation and hyp have s ~(dom a) ~ t by auto

        –finally using the weak Step Consistency definition we finish the
        proof for this case.
        from this and weaklyStepConsistent
```

```
                  show contents (step a s) n = contents (step a t) n
                  unfolding weaklyStepConsistent-def
                  by (metis acRelation hyp)
                } qed end of the second case
              }
            }
          }
       qed end of the step consistency proof
    }
    from this and acRelation show ?thesis unfolding stepConsistent-def
    by smt
  qed
```

Using this lemma we show Weakly Step Consistency can be "strengthen" to the original Step Consistency definition if we use some of the other conditions on the Rushby Intransitive model.

5.5.2 Security of Weak Step Consistent Systems for Transitive Policies

Using previous lemma we can show that:

Theorem: A Weakly Step Consistent system with structured state that satisfies the third Monitor Assumption and the following two conditions.

- 1 $u \rightsquigarrow v \supset observe(u) \subseteq observe(v)$, and
- 2 $n \in alter(u) \wedge n \in observe(v) \supset u \rightsquigarrow v$.

Is secure for the policy $\rightsquigarrow$.

We prove this lemma in Isabelle/HOL as follows:

```
theorem weakStepConsistentSystemIsSecure:
   assumes 1:∀ u v. ((u ⇝ v) ⟶ observe u ⊆ observe v)
   and     2:∀ u v n. ((n ∈ alter u) ∧ (n ∈ observe v)) ⟶ (u ⇝ v)
   and     rma1:rma1
   and     weaklyStepConsistent:weaklyStepConsistent
   and     rma3:rma3
     shows secure
proof –
```

To prove this theorem we need to show thatoiur assumptions imply *outputConsistent*, *stepConsistent* and *LocalRespects*

Output consistency (*outputConsistent*) can be proven directly form the first reference monitor *rma1*

from rma1 **have** outputConsistent:outputConsistent
unfolding rma1-def **and** outputConsistent-def **by** auto

To show that the system holds the locally respects condition (*LocalRespects*), we need to show: ($s \sim u \sim (step\ a\ s)$) that can be proven from the definition of $\overset{u}{\sim}$ and the third reference monitor (*rma3*)

have LocalRespects:LocalRespects
unfolding LocalRespects-def
proof (auto)**{**
fix a::′Action
fix s::′State
fix u::′Domain
assume notInt:(dom a) $\not\rightsquigarrow$ u
from notInt **show** (s ~ u ~ (step a s))
by (metis 2 acRelation rma3 rma3-def)
}
qed

to show step consistency we invoke the weakStepConsistencyToStepConsistent lemma

from assms **have** stepConsistent:stepConsistent **using** weakStepConsistencyToStepConsistent
by auto

Once we have proven that the AC system complies with the *outputConsistent*, *stepConsistent* and *LocalRespects* conditions, we simply invoke the *simple-noninterference* to show this system is secure.

from outputConsistent **and** LocalRespects **and** stepConsistent **and** simple-noninterference
show ?thesis **by** auto
qed

5.5.3 Non-interference for GWV Secure Systems

Finally in the following theorem we prove a new version of Rusbhy's Theorem 2 (*accessControlSystemIsSecure*). For this Theorem we do not assume GWVrma2 but we derive stepConsistent from $GWVsecure$, Weak Step Consistency and $GWVcond1$.

theorem GWVSystemIsSecure:

We use the definitions for the GWV reference monitors

assumes GWVrma1:GWVrma1

and GWV-secure:GWV-secure

and GWVrma3:GWVrma3

and GWVcond1:GWVcond1

and GWVcond2:GWVcond2

and ActionStep:$\forall$ s t u . GWVvpeq s u t $\longrightarrow$ Action (Next s) = Action (Next t)

And we use the mappings

and selectMapping:selectMapping

and stepMapping:stepMapping

and outputMapping:outputMapping

and alterMapping:alterMapping

and currentMapping:currentMapping

and GWVvpeqtoRelation:GWVvpeqtoRelation

and intMapping

shows secure

proof –

for this theorem we need to prove that the three GWV reference monitors and the two

additional conditions imply *outputConsistent*, *stepConsistent* and *LocalRespects*

we use the GWV reference monitor definitions and the mapping to get the original Rushby reference monitors

from assms **and** GWVrma1ToRma1 **have** rma1:rma1 **by** auto

from assms **and** GWVrma3ToRma3 **have** rma3:rma3 **by** auto

from assms **and** proofobligation2Mapping

have 2:$\forall$ u v n. ((n $\in$ alter u) $\wedge$ (n $\in$ observe v)) $\longrightarrow$ (u $\rightsquigarrow$ v) **by** auto

from assms **and** proofobligation1Mapping

have 1:$\forall$ u v. ((u $\rightsquigarrow$ v) $\longrightarrow$ observe u $\subseteq$ observe v) **by** auto

from assms **and** GWV-RusbhyWeakStepConsitent **have**

weakStepConsistent:weaklyStepConsistent

by auto

Output consistency (*outputConsistent*) can be proven directly from the first reference monitor *rma1*

```
from rma1  have outputConsistent:outputConsistent
unfolding rma1-def and outputConsistent-def by auto
```

To show that the system respects the locally respects condition (*LocalRespects*), we need to show: $(s \sim u \sim (step\ a\ s))$ that can be proven from the definition of $\overset{u}{\sim}$ and the third reference monitor (*rma3*)

```
have LocalRespects:LocalRespects
unfolding LocalRespects-def
  proof (auto){
    fix a::'Action
    fix s::'State
    fix u::'Domain
    assume notInt:(dom a) \~> u
    from notInt show (s ~ u ~ (step a s))
      by (metis 2 acRelation rma3 rma3-def)
      }
  qed
```

Now we invoke the *weakStepConsistencyToStepConsistent* lemma to show step Consistency

```
from weakStepConsistent and 1 and 2 and rma3
have stepConsistent:stepConsistent
using weakStepConsistencyToStepConsistent
by auto
```

Once we have proven that we comply with the *simple-noninterference* is easy to show that the system is secure.

```
 from  acRelationisTransitive have eqvRelTransitive:eqvRelTransitive by auto
 from acRelationIsSymmetric have eqvRelSymmetric:eqvRelSymmetric by auto
 from  acRelationIsReflexive have eqvRelreflexive:eqvRelreflexive by auto

from outputConsistent  and LocalRespects and stepConsistent and simple-noninterference
  and eqvRelTransitive and eqvRelSymmetric and eqvRelreflexive
  show ?thesis by auto
qed
```

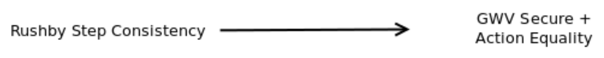

FIGURE 5.4: Relation Between Weakly Step Consistency and GWV secure definitions.

5.6 Comparing GWV Secure vs Rushby's Step Consistency

In this section we show that GWV security notion is contained by Rushby Step Consistency definition. In other words that a Step Consistent Rushby system is also GWV Secure. This relation is illustrated by Figure 5.4.

lemma *GWVstepConsistent* is used to translate Rushby step consistentcy $s \sim u \sim t \longrightarrow (step\ a\ s) \sim u \sim (step\ a\ t)$ into an equivalent expression given under GWV terms: $GWVvpeq\ s\ u\ t \longrightarrow GWVvpeq\ \ (Next\ s)\ u\ (\ Next\ t)$

One primary difference between the Rushby and GWV models, is result of the way of how their state machines are defined. On one hand, the Rushby model defines state transitions through *step* function:

-step : Action × State → State

On the other hand, GWV model defines state changes with *next* function:

-Next : State → State

Taking a look on $Next$ definition, we can notice that GWV states transitions are deterministic. In other words, A state s will always result in the same state s' after executing *next* function. Whereas, in the Rushby model an state can execute different actions, that will affect the outcome of the *step* function.

As a result of this difference, to deduce the GWV separation theorem from the Rushby model, we need to prove that the action executed by the states on GWV separation theorem is the same. This is impossible to prove because in the same current partition there can be different active tasks. To avoid it, we introduced a notion of action equality to GWV model.

definition GWV-secure-wAction:: bool **where**
  GWV-secure-wAction ≡ ∀ st1 st2 seg.
      current st1 = current st2 ∧
      select st1 seg = select st2 seg ∧
      Action st1 = Action st2 ∧

```
equals ((dia seg) ∩ (segs (current st1))) st1 st2
⟶ (select (Next st1) seg = select (Next st2) seg)
```

We define an Isabelle/HOL lemma to show that the original GWV secure definition implies our new formulation including action equality:

```
lemma GWVsecure-wAction-isSecure:
  assumes GWV-secure:GWV-secure
  shows GWV-secure-wAction
proof –
  from assms show ?thesis unfolding GWV-secure-def GWV-secure-wAction-def
  by auto
qed
```

We now prove that Rushby step consistency implies our new definition of GWV secure with action equality, for this theorem we needed to add an extra hypothesis named *hyp*:

$$hyp\text{: } \forall\ n.\ \exists\ p.\ n \in segs\ p \wedge (\forall\ n'.\ n' \in segs\ p \longrightarrow n' = n$$

This extra hypothesis is composed by two parts $\forall\ n.\ \exists\ p.\ n \in segs\ p$ states that every partition should be assigned at least one partition, and $(\forall\ n'.\ n' \in segs\ p \longrightarrow n' = n)$ requires that every partition has (only) one memory segment assigned to it.

```
theorem stepConsistencyImpliesGWVSecure:
  assumes  1: stepConsistent
  and      2: stepMapping
  and      3: selectMapping
  and      4: observeMapping
  and      5: GWVvpeqtoRelation
  and    hyp: ∀ n. ∃ p. n ∈ segs p ∧ (∀ n′. n′ ∈ segs p ⟶ n′ = n)

shows  GWV-secure-wAction

proof –
  show ?thesis
  unfolding  GWV-secure-wAction-def

  proof auto{
    fix st1 st2::(′State, ′Action) memorystate
    fix seg::′Object
    we use the hypotesis of the GWV-secure definition.
    assume GWVhyp2:select st1 seg = select st2 seg
    assume GWVhyp3:Action st1 = Action st2


    show select (Next st1) seg = select (Next st2) seg
      proof–{

        Using the GWVstepConsistent lemma and our premises we prove that
        having two equivalent states and a segment that belongs to a partition and
        we apply the same action to the states, then the values will remain the same
        after executing a step on the state machine.
        have GWVstepconst:∀ u. GWVvpeq st1 u st2 ∧ seg ∈ segs u
        ⟶ (select (Next st1) seg = select (Next st2) seg)
        using GWVstepConsistent and assms
        unfolding GWVvpeq-def
        by auto

        The conclusion directly falls from the last fact and the action equality
       of the third GWV premise GWVhyp3
       then show ?thesis using hyp GWVhyp2 GWVhyp3 unfolding GWVvpeq-def
by metis
        }
      qed
  }qed
qed
```

As the reader can notice, so far we have proven the relation between the different formulations of security defined in transitive version of the GWV and Rushby models. now we proceed with the intransitive case.

5.7 Rushby Intransitive Non-Interference vs GWV

In this section we compare the Rushby intransitive non Interference described in Section 3.2 versus GWV security model.

As we proved in Section 5.5 GWV security definition implies weak step consistency, which is a key concept for Rushby's intransitive non Interference.

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \supset step(a,s) \overset{u}{\sim} step(a,t)$$

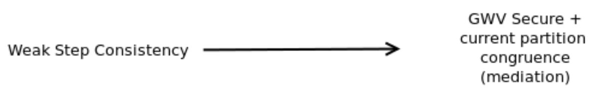

FIGURE 5.5: Relation Between GWV Mediation and Weakly Step Consistency definitions.

As consequence we can use the GWV security definition to show that a GWV secure system that complies with the other Rusbhy reference monitor conditions is secure for Rushby non interference:

theorem *GWVIsiSecure*:

We use the definitions for the GWV reference monitors

assumes GWVrma1:GWVrma1

and GWV-secure:GWV-secure

and GWVrma3:GWVrma3

and GWVcond2:GWVcond2

and ActionStep:∀ s t u . GWVvpeq s u t ⟶ Action (Next s) = Action (Next t)

And we use the mappings

and selectMapping:selectMapping

and stepMapping:stepMapping

and outputMapping:outputMapping

and alterMapping:alterMapping

and currentMapping:currentMapping

and GWVvpeqtoRelation:GWVvpeqtoRelation

and intMapping

shows isecure

Rushby Weak Step Consistency cannot really ensure GWV separation but a weaker property known as Mediation. The reason for this will be discussed in the next Chapter. Figure 5.5 shows the relation between Rushby step consistency and GWV mediation definitions. We add a definition for mediation (including equality of actions) to prove this.

definition GWV-secureEPartition:: bool **where**

GWV-secureEPartition ≡ ∀ st1 st2 seg.

current st1 = current st2 ∧

select st1 seg = select st2 seg ∧

Action st1 = Action st2 ∧

equals ((segs (current st1))) st1 st2

⟶ (select (Next st1) seg = select (Next st2) seg)

Finally we can prove that Rushby weak step consistency implies our new definition of mediation.

```
 theorem weaklyStepConsistencyImpliesGWVSecure:
    assumes  1: weaklyStepConsistent
    and      2: stepMapping
    and      3: selectMapping
    and      4:currentMapping
    and      5: observeMapping
    and      6: GWVvpeqtoRelation
    and    hyp: ∀ n. ∃ p. n ∈ segs p ∧ (∀ n′. n′ ∈ segs p ⟶ n′ = n)
shows GWV-secureEPartition
 proof –
   show ?thesis
   unfolding GWV-secureEPartition-def
   proof auto{
    fix st1 st2::(′State, ′Action) memorystate
    fix seg::′Object

    we use the hypotesis of the GWV-secure definition.
    assume GWVhyp1:current st1 = current st2
    assume GWVhyp2:select st1 seg = select st2 seg
    assume GWVhyp3:Action st1 = Action st2
    assume GWVhyp4:∀ a∈ segs (current st2). select st1 a = select st2 a

    show select (Next st1) seg = select (Next st2) seg
       proof–{

       Using the GWVstepConsistent lemma and our premises we prove that having
       two equivalent states and a segment that belongs to a partition and we apply
       the same action to the states, then the values will remain equal after executing
       a step on the state machine.
          have GWVstepconst:∀ u.
               GWVvpeq st1 u st2
               ∧ GWVvpeq st1 (current st1) st2
               ∧ seg ∈ segs u
               ∧ Action st1 = Action st2
          ⟶ (select (Next st1) seg = select (Next st2) seg)
          using GWVWeaklyStepConsistent and assms
```

```
        unfolding GWVvpeq-def
        by auto

        from GWVhyp4 and GWVhyp3 and GWVhyp1
        have h2:GWVvpeq st1 (current st1) st2
        unfolding GWVvpeq-def by simp

        then  show ?thesis using hyp GWVhyp2 GWVstepconst GWVhyp3
        unfolding GWVvpeq-def by blast
        }
      qed
  }qed
qed
```

So far we have proven the relation between the GWV and Rushby models. In the next Chapter we discuss about the differences between the two models and explain the need of the extra conditions we need to add into the original formulations.

Chapter 6

Discussion

There are several fundamental differences between the GWV and Rushby models. These differences complicate the task of comparing them.

Figure 6.1 recapitulates the relation between the security formulations in the GWV and Rushby models. In Chapter 5, we proved that GWV separation definition and one extra condition we named (GWVcond1) implies Rushby's non-interference step consistency definition. Whereas, step consistency implies GWV-secure (we assume that the action executed in the active partition are the same).

If we consider intransitive policies we show that GWV separation definition implies weakly step consistency, while Rushby's definition can only ensure the concept of mediation proposed in the original model GWV.

This kind of fundamental differences results in the existence of some cases where a policy is valid in a model but not for the other.

In this chapter we explore some of these instances, and discuss about the conditions that must be fulfilled by each model to avoid such cases.

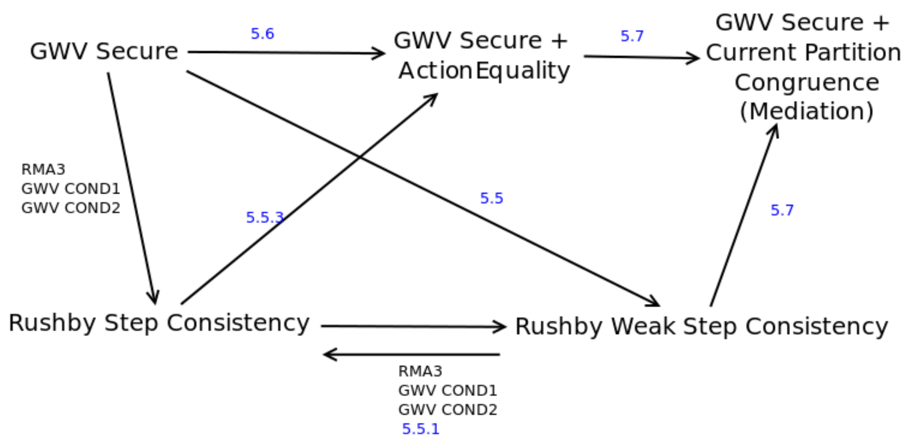
GWV Secure
5.6
GWV Secure + ActionEquality
5.7
GWV Secure + Current Partition Congruence (Mediation)
RMA3
GWV COND1
GWV COND2
5.5.3
5.5
5.7
Rushby Step Consistency
Rushby Weak Step Consistency
RMA3
GWV COND1
GWV COND2
5.5.1

FIGURE 6.1: Relation Among the GWV and Rushby Security definitions

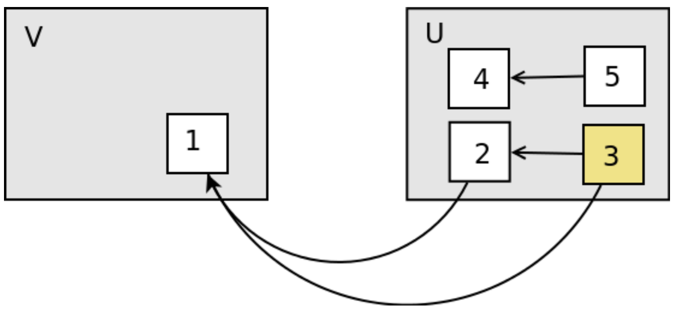

Figure 6.2: GWV Secure Policy

6.1 Example 1

For our first example we study one of the conditions added to the GWV model in Section 5.3:

> definition GWVcond1::"bool" where
>
> GWVcond1 ≡ ∀ u v. GWVia u v ⟶ observe v ⊆ observe u

Consider a system M as depicted in Figure 6.2. System M consists of two partitions U and V. These partitions have several memory segments assigned to them.

The sole purpose of partition V is to read information from the system. Partition V does not write any data. Partition V contains a memory segment "1", which is used to obtain information from partition U.

On the other hand, partition U has four assigned memory segments $\{2, 3, 4, 5\}$. Segment 1 can only be modified with information from segments 2 and 3. Meanwhile, segment 4 can only be modified with information from segment 5.

System M can be modelled using the following GWV security policy:

$$
\begin{gathered}
segs(V) = \{1\} \\
segs(U) = \{2, 3, 4, 5\} \\
dia(1) = \{2, 3\} \\
dia(2) = \{3\} \\
dia(4) = \{5\}
\end{gathered}
$$

In this particular case, we assume system M is secure for the GWV separation definition. That is, M meets all the conditions of the Separation theorem we discussed in Section 4.3:

> (1) current s = current t ∧
>
> (2) select s seg = select t seg ∧
>
> (3) equals ((dia seg) ∩ (segs (current s))) s t
>
> ⟶ (select (Next s) seg = select (Next t) seg)

GWV separation theorem definition states that for any two states "s" and "t", and a given partition "seg" where the following conditions are met:

- (1)The Active partition is the same in both states.
- (2)The value of the segment "seg" is equal in the both states.
- (3)The values of the segments that can modify "seg" and are assigned to the active partition equal.

The value of "seg" will not be modified after performing a step on the state machine.

It is important to emphasize that according to GWV separation definition; a partition can read (observe) from all its assigned memory segments($segs(V)$) [8]. Furthermore, GWV security definition prohibits the information flow between segments that do not have a *dia* relation.

In this example, partition U can read from segments $\{2, 3, 4, 5\}$, but the system can only change segment 1 with the information from segments 2 and 3. In other words, according to the GWV model, system M forbids the information flow from segments 4 and 5 to segments 1,2,3.

On the other hand, system M we just described would be defined in the Rushby model by the following sequence of statements:

$$\begin{gathered} Alter(U) = \{1\} \\ Observe(U) = \{2, 3, 4, 5\} \\ Observe(V) = \{1\} \\ Alter(V) = \varnothing \end{gathered}$$

In this case, an interference relation between U and V ($U \rightsquigarrow V$) is implied by system M. Segment 1 can be modified by U and observed by V. However, if we observe carefully the Rushby **transitive** access control theorem 3.1.8:

Theorem: A system with structured state that satisfies the Reference Monitor Assumptions and the following two conditions.

- *1* $u \rightsquigarrow v \supset observe(u) \subseteq observe(v)$, *and*
- *2* $n \in alter(u) \wedge n \in observe(v) \supset u \rightsquigarrow v$.

Is secure for the policy $\rightsquigarrow$.

In particular the first theorem condition ($u \rightsquigarrow v \subset observe(u) \subseteq observe(v)$), states that when two partitions U and V have an interference relation, partition V must be able to read from all the segments that U can read.

As we can see, system M clearly violates the aforementioned condition. Partition U can read from the segments $\{2, 3, 4, 5\}$, and partition V can only read from segment 1.

In other words, in a system like this, not everything that can be observed by U can also be observed by the partition V. Thus violating the Rushby condition.

Intra-partition flow restrictions (like the one required for the partition U) are in practice complex and difficult to ensure. One reason for this, is that all memory segments assigned to the same partition are loaded in the same kernel working area, and can be read for all the processes running in the partition. Jim Alvez [8] discusses a possible attack on the GWV model consisting on modifying the value of a segment, and returning it to its previous value before the evaluation of the security policy (cut point).

A GWV system is expected to prevent the flow from segments $\{4, 5\}$ to segments $\{2, 3\}$. For commercial processors, this capability is too powerful. Custom made hardware is needed to provide the ability to restrict information flow from a specific memory segment to a particular destination segment.

For example, during a partition's execution an untrusted process could copy information from either segments $\{4, 5\}$ into the segments $\{2, 3\}$ and restore the previous value before the policy evaluation (cut-point). The only way to stop the copy from $\{4, 5\}$ is to label the information based on its original source and prevent the data flow during execution time.

May seem like this discrepancy could be caused by our choice to use a strict definition for the *observe* mapping. We considered to use a weaker definition: $\forall u.observe(u) = (\{s'.(\exists s.s \in segsu \wedge s' \in dia(s))\} \cup segs(u)$. To include not only memory segments assigned to a partition but also all segments having a *dia* relation with them. However, even with this expanded definition the difference between the models remains.

6.2 Example 2

In the next example we are going to explore the Rushby security condition, and describe one instance where a Rushby security definition allows valid memory change that violates GWV's separation notion.

Consider a system N like the one depicted in Figure 6.3. System N represents an implementation of an intransitive security policy composed by three classification domains: High, Downgrader and Low, enclosed in the partitions H, D and L respectively.

Partition H can read data from memory segment 1 and send information to partition D using the memory segments $\{2, 3\}$.

In this system, we assume that the Downgrader partition (D) has two specific goals. The first goal is to declassify the information obtained from the High H domain, and

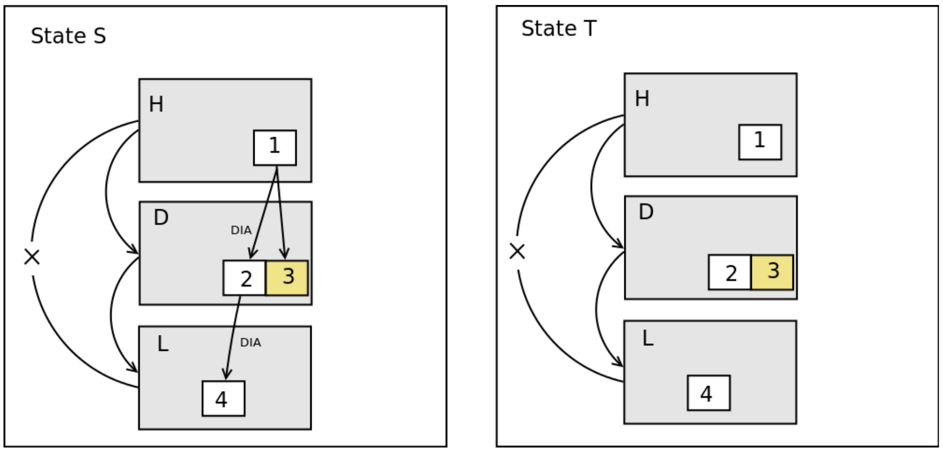

FIGURE 6.3: Policy expressed with GWV and Rushby Models

send the declassified information to partition L. The second goal is to store a copy of the original data for auditing purposes. Partition D stores the original data in segment 3, while, the declassified data are stored in segment 2, and sent to partition L using segment 4.

This kind of intransitive security policies can be written under Rushby terms as follows:

$$H \rightsquigarrow D$$
$$D \rightsquigarrow L$$
$$H \not\rightsquigarrow L$$

A possible implementation of this system can be made with the following object distribution:

$$Observe(H) = \{1\}$$
$$Alter(H) = \{2,3\}$$
$$Observe(D) = \{2,3\}$$
$$Alter(D) = \{4\}$$
$$Observe(L) = \{4\}$$
$$Alter(L) = \varnothing$$

Imagine that system N has two states s and t as illustrated in Figure 6.3. Segment 3 has a different value in states s and t ($contents(s,n) \neq contents(t,n)$).

Next suppose we execute an action a such that $dom(a) = \{D\}$ in both states. After executing action a, we evaluate the security policy for partition L in the resulting states s' and t', where $s' = step(a,s)$ and $t' = step(a,t)$.

If we take a look at the definition of weakly step consistency for Rushby intransitive policies:

$$s \overset{l}{\sim} t \wedge s \overset{dom(a)}{\sim} t \longrightarrow step(a,s) \overset{l}{\sim} step(a,t)$$

As we can notice, expression $s \overset{dom(a)}{\sim} t$ is not true. The value of segment 3 is not equal in both states. In consequence, Rushby model does not guarantee that the value of Segment 4 remains identical in states s' and t' after executing a.

However, this situation does not violate the condition of weak step consistency. Moreover, the value of Segment 3 may change without implying that the system is insecure.

On the other hand, if we consider the same policy implemented in GWV as follows:

$$\begin{aligned} segs(H) &= \{1\} \\ segs(D) &= \{2,3\} \\ segs(L) &= \{4\} \\ dia(2) &= \{1\} \\ dia(3) &= \{1\} \\ dia(4) &= \{2\} \end{aligned}$$

Taking a closer look on the GWV separation theorem we discussed in section 4.3:

(1) current s = current t $\wedge$
(2) select s seg = select t seg $\wedge$
(3) equals ((dia seg) $\cap$ (segs (current s))) s t
$\longrightarrow$ (select (Next (s)) seg = select (Next (t)) seg)

If we evaluate the separation definition for the segment 4 ($seg = 4$) and the states s and t (with different values for segment 3), we can notice that all the GWV conditions are met:

- We apply the same action a to the states S and t. Hence, both states have the same current partition (D).
- The value of segment 4 is equal in the states s and t, thus the second condition is satisfied.
- The intersection of the $dia(4) \cap segments(D) = 2$ have the same value on the s t states.

However, if the value of segment 4 is different after the execution of action a (as in the Rushby model). It would violate the GWV separation definition as we would have $(select(Next(s))seg \neq select(Next(t))$.

From this example, we can conclude that the only way for an intransitive Rushby system to comply with the GWV security definition, is if all segments read by the active partition have the same value (not only the intersection with the *dia* function). This condition matches the *Mediation* definition of the original GWV model.

The Theorem weaklyStepConsistencyImpliesGWVSecure we discussed in Section 2.5.2 shows that *weakly step consistency* implies Mediation.

Chapter 7

Conclusions and Future Work

Summary and Conclusions

In this document we emphasized the importance of Multiple levels of Security models as an appropriate way to ensure Information Security in computer systems. We explored MILS architecture and concluded that it is a suitable way to implement MLS models. A key concept in the MILS architecture is the separation kernel. Separation kernels are in charge of providing and ensuring isolation between the different security domains in MILS compliant systems.

A Separation Kernel Protection Profile containing the security requirements for high robustness separation kernels was published by the NIST and the NSA. Among the requirements of the SPPK we can find the need to verify the correctness of the separation kernel by means of formal methods.

There are several formal models that can be used to verify security kernels. The best known models are GWV separation and Rushby non-Interference models. However, in the SKPP there is not any guideline on how to select the appropriate model.

This brought up the following question: "what are the exact differences and similarities between the GWV separation model and the non-interference model proposed by Rushby."

This work contains to our knowledge, the first publicly available formal comparison among GWV and Rushby separation kernel models. The main results of this work is the creation of a mapping relating the concepts of the GWV and Rushby separation kernel models as well as the accompanying proofs relating the security definition in both models.

As we discussed in Chapter 5 we provided a mapping of the methods and properties between we found and proven the relation of the security definition in both models.

In conclusion, we can state that both models have a great number of similarities; both of them define security domains, and control the data flow between the such domains according to a predefined security policy. However, they present a range of fundamental differences between them:

- GWV and Rushby models are composed of several components that despite being similar to one another are not completely equivalent.
- Rushby controls the flow of information between different security domains, while the GWV controls the flow between different memory segments. This results in a significant difference in the granularity of both models.

Although, none of the aforementioned differences imply an evident security issue, we could find cases cases were a system is considered secure for the GWV model but not under Rushby's model and vice versa.

Formally determine the differences between different separation kernel models could bring several practical advantages. For example:

- Find the similarities and differences between the two models can provide a starting point to identify the strengths and weaknesses of each model;
- clarify the strengths and weaknesses of each model can serve as a guide to ease the choice of the most suitable formal model in accordance to the specific needs of the separation kernel to be evaluated, and
- Serve as base to estimate if it is feasible to design a system that complies with both security definitions.

Future Work

In this document we have compared the original formulations for the Rushby and GWV separation kernel models. However, both models have suffered several revisions and refinements to extend and clarify its functionality. The coming of these new revisions and models open a promising and interesting field for further research to determine the exact relation between them.

On the one hand, Greve proposed a later revision of the GWV model called GWVr1 [32]. GWVr1 defines the notion of agents with the purpose of adding accountability properties into the original model. Another generalization called GWVr2[33] was subsequently defined to cover security policies for dynamic and distributed systems.

On the other hand, David Oheim proposed a generalization of the original Rushby non-interference model to cover non-deterministic state machines. While Eggert and Van der Meyden [34] argued that the classic notion of intransitive non-interference (*ipurge*) allows some cases where a low security domain can infer information from higher security domains without the intermediation of the trusted downgraders and prosed a new definition of security named $TA-security$.

Appendix A

Isabelle/HOL

Isabelle [35, 36] is a generic system for interactive theorem proving. It was implemented in the functional programming language ML, and developed by Larry Paulson and Tobias Nipkow. It is available in isabelle(http://isabelle.in.tum.de).

Isabelle is based on tactics, characteristic inherited from the LCF system (Logic for Computable Functions)[37], which allows building tactics in order to simplify the mechanical application rules for the inference deduction process. Isabelle supports formal reasoning in First Order Logic (FOL) and Higher-Order logic (HOL).

In particular Isabelle/HOL [38] is the specialization of Isabelle for Higher-Order Logic. Isabelle/HOL is an interactive prover assistant, based on Gordon's HOL system[39], which implements an extension of Church's [40] Higher-Order logic. HOL has a large library of theories including set theory, real and complex numbers, abstract algebra, etc.

Isabelle/HOL has been successfully used for formal reasoning in various areas of knowledge: Pure Mathematics, verification of computing systems, programming languages, etc.

An Archive of Formal proofs is available with a variety of theories and examples corresponding to scientific developments that have been formally verified in Isabelle and especially Isabelle/HOL.

Basic Elements of Isabelle/HOL

In this section we illustrate some important aspects of Isabelle/HOL described by Nipkow and Paulson [38]. Please note this document does not intend to completely describe the capacities and features offered by Isabelle/HOL and its multiple extensions, but to provide a quick overview of its main language components.

Theories

Isabelle/HOL formalizations are defined in theories. Theories are modules that contain definitions, terms, formulas, data types, functions, and theorems, etc., that describe the solution of a problem.

When we use Isabelle/HOL, we define theories as extensions of other theories and thus we "import" the data types and structures defined in the imported theories. Isabelle/HOL contains a *Main* theory that includes predefined basic theories, such as the arithmetic of natural numbers, lists and sets. The general syntax to define a theory is:

```
Theory
        import T1. . . Tn
begin
        Type declarations, function definitions and proofs
end
```

Where T_1... T_n are the names of existing theories for which the new theory is defined.

In the *declarations*, *definitions*, *and proofs* section we introduce new concepts used to solve a problem such as: types, functions, theorems, lemmas, and their respective proofs.

Sometimes it is useful to introduce , through declarations, new concepts to expand the theories already established. Functional programming needs datatypes and functions. Both of them can be defined in Isabelle/HOL.

Type Declarations

The general datatype syntax in Isabelle has the form

$$\textbf{datatype}\ (\alpha_1 \ldots\ \alpha_n) = C_1..C_n$$

Where α represents the name of the datatype and C_1 represents the constructors of the type.

Function Definition

In isabelle/HOL We declare a definition by defining a name, a type, and a set of defining equations. Functions can be defined with the following keywords[41]:

definition: A function without recursion.

fun: For cases of recursion where termination can be automatically proved. For Isabelle automatically prove the completion of a function, it is necessary that the recursive calls of the arguments of the function on the right side of each equation are strictly smaller than the arguments corresponding to the left.

primrec: For primitive recursion. In HOL all functions defined by $func$ are fully recursive. When it is not possible to define a fully recursive function primitive recursion (**primrec**) must be used.

Isar Structured Testing

Isabelle/HOL can be used to prove lemmas and theorems, in this section we introduce the features of the ISAR (Intelligible semi-automated reasoning) proof environment. ISAR offers a framework for human readable structured proofs. This Section is based on [42], ISAR proofs examples can be found in http://isabelle.in.tum.de/Isar/.

The proof environment in ISAR makes use of the following syntax:

```
|lemma|theorem| [<name>:]

[assumes "formula"]

show "formula"
proof [method] statement qed | by method


method = (simp...)|(blast...)|(rule...)|...

statement =     fix variables |
                        assume prop(==>)|
                        [from fact+] (have|show) prop proof|
                        next (separates subgoals)

prop = [name:] "formula"

fact = name|name[OF fact+]|‘formula’
```

Isabelle proofs starts with the *lemma* or *theorem* keywords, always followed by a goal to prove (*shows* formula). Please note that we only need to provide a name(label) to the lemma/theorem is we want to reference it in the future.

In ISAR syntax *"assumes"* keyword is used to specify the hypotheses of the statement we want to prove.

An ISAR a proof environment consists of a single method preceded by the *by* word or by a $proof-qed$ block composed by zero or more statements. A block may optionally start with the declaration of a method in order to indicate how start the test, for example, (*inductn*)

A statement consists of one of two kinds of propositions, a hypothesis together with your demonstration.

The optional word *from* indicates which facts or hypotheses are used in the demonstration. The intermediate propositions begin with the word *have* and the main proposition with *show*.

A statement can introduce new local variables with the keyword *fix*.

The propositions are formulas preceded optionally a name (label) that subsequently allows the formula to refer to corresponding an assertion preceded with the word *from*.

Bibliography

[1] Carl E. Landwehr, Constance L. Heitmeyer, and John D. McLean. A security model for military message systems: Retrospective. In *ACSAC*, pages 174–190. IEEE Computer Society, 2001. ISBN 0-7695-1405-7. URL http://dblp.uni-trier.de/db/conf/acsac/acsac2001.html#LandwehrHM01.

[2] Sylvia Osborn. Mandatory access control and role-based access control revisited. In *IN PROCEEDINGS OF THE 2ND ACM WORKSHOP ON ROLE-BASED ACCESS CONTROL*, pages 31–40. ACM Press.

[3] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical report, MITRE Corporation, March 1973.

[4] J. K. Biba. Integrity considerations for secure computer systems, 1977.

[5] John Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles (SOSP)*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[6] David Greve, Matthew Wilding, and W. Mark Vanfleet. A separation kernel formal security policy. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '03)*, July 2003.

[7] John Rushby. Noninterference, transitivity and channel-control security policies. Technical report, Computer Science Laboratory, SRI international, 1992.

[8] Jim Alves-foss and Carol Taylor. An analysis of the gwv security policy. In *In 5th Internat. Workshop on ACL2 Prover and Its Applications*, pages 2–2004, 2004.

[9] M.E. Whitman and H.J. Mattord. *Principles of Information Security*. Course Technology, 2010. ISBN 9781111138219. URL http://books.google.nl/books?id=L3LtJAxcsmMC.

[10] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, October 2001. ISSN 0925-9856. doi: 10.1023/A:1011254632723. URL http://dx.doi.org/10.1023/A:1011254632723.

[11] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009. ISSN 0256-2499. doi: 10.1007/s12046-009-0001-5. URL http://dx.doi.org/10.1007/s12046-009-0001-5.

[12] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28:626–643, 1996.

[13] Matt Kaufmann and J. Strother Moore. An acl2 tutorial. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 17–21. Springer, 2008. ISBN 978-3-540-71065-3. URL http://dblp.uni-trier.de/db/conf/tphol/tphol2008.html#KaufmannM08.

[14] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. URL http://www.csl.sri.com/papers/cade92-pvs/.

[15] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[17] Mark W. Vanfleet, Jahn A. Luke, William R. Beckwith, Carol Taylor, Ben Calloni, and Gordon Uchenick. MILS:Architecture for High-Assurance Embedded Computing. *CrossTalk: Journal of Defence Software Engineering*, 18(8):12–16, 2005. URL http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.pdf.

[18] Objective Interface Systems Joe Jacob. Mils: High-assurance security at affordable costs. *"COTS journal, the Journal of Military electronics and Computing"*, 28, November 2005. URL http://www.cotsjournalonline.com/articles/view/100423.

[19] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems, 1975.

[20] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 210–224, New York, NY, USA, 1973. ACM. doi: 10.1145/800122.803961. URL http://doi.acm.org/10.1145/800122.803961.

[21] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382230. URL http://doi.acm.org/10.1145/2382196.2382230.

[22] John Rushby. Separation and integration in mils (the mils constitution. Technical report, Computer Science Laboratory, SRI international, 2008.

[23] Jonas Frid. Security critical systems in software, 2010.

[24] Kei Kawamorita, Ryouta Kasahara, Yuuki Mochizuki, and Kenichiro Noguchi. Application of formal methods for designing a separation kernel for embedded systems. *World Academy of Science, Engineering and Technology*, (44):1313–1321, 2010.

[25] Provisional Harmonised Criteria. Information technology security evaluation criteria (itsec), 1991.

[26] Department of Defense. *Trusted Computer System Evaluation Criteria*. December 1985.

[27] Canadian System Security Centre. *The Canadian Trusted Computer Product Evaluation Criteria*. Canadian System Security Centre, Communications Security Establishment, Government of Canada, 1991. URL http://books.google.nl/books?id=m6g5nQEACAAJ.

[28] Information Assurance Directorate. U.s. government protection profile for separation kernels in environments requiring high robustness, 2007.

[29] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[30] J. Thomas Haigh and William D. Young. Extending the noninterference version of mls for sat. *IEEE Trans. Software Eng.*, 13(2):141–150, 1987.

[31] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[32] Michael W. Whalen, David A. Greve, and Lucas G. Wagner. *Model Checking Information Flow*. Springer-Verlag, Berlin Germany, March 2010.

[33] David Greve, Matthew Wilding, Raymond Richards, and W. Mark Vanfleet. Formalizing security policies for dynamic and distributed systems. *Unpublished*, 2004. URL http://hokiepokie.org/docs/sstc05.pdf.

[34] Sebastian Eggert, Ron van der Meyden, Henning Schnoor, and Thomas Wilke. Complexity and unwinding for intransitive noninterference. *CoRR*, abs/1308.1204, 2013.

[35] Lawrence C. Paulson. Introduction to Isabelle. Technical Report UCAM-CL-TR-280, University of Cambridge, Computer Laboratory, January 1993. URL http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-280.dvi.gz.

[36] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994. ISBN 3-540-58244-4.

[37] Mike Gordon. From lcf to hol: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 169–186. The MIT Press, 2000. ISBN 978-0-262-16188-6. URL http://dblp.uni-trier.de/db/conf/birthday/milner1999.html#Gordon00.

[38] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[39] M. J. C. Gordon and A. M. Pitts. The HOL logic and system. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*, chapter 3, pages 49–70. Elsevier Science B.V., 1994.

[40] Alonzo Church. A formulation of the simple theory of types. *Jurnal of Symbolic Logic*, 5(2):56–68, June 1940.

[41] Alexander Krauss. Defining recursive functions in isabelle/hol.

[42] Markus Wenzel and Tu München. The isabelle/isar reference manual. 1999.