Security Tagging for a Real-Time Zero-Kernel Operating System: Implementation and Verification

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctorate of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Jia Song

October 2014

Major Professor: Jim Alves-Foss, Ph.D.

Authorization to Submit Dissertation

This dissertation of Jia Song, submitted for the degree of Doctorate of Philosophy with a Major in Computer Science and titled "Security Tagging for a Real-Time Zero-Kernel Operating System: Implementation and Verification," has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor _____Date_____
Jim Alves-Foss, Ph.D.

Committee Members _____Date_____
Daniel Conte de Leon, Ph.D.

_____Date_____
Clinton Jeffery, Ph.D.

_____Date_____
Paul Oman, Ph.D.

Department Administrator _____Date_____
Gregory Donohoe, Ph.D.

Discipline's College Dean _____Date_____
Larry Stauffer, Ph.D.

Final Approval and Acceptance by the College of Graduate Studies

_____Date_____
Jie Chen, Ph.D.

Abstract

This dissertation is a continuation of an Air Force Research Laboratory (AFRL) project focused on the development of a new security tagged microprocessor architecture and a supporting operating system. Security tagging schemes are promising mechanisms for enhancing the security of computer systems. The idea behind security tagging schemes is to attach metadata tags to memory and registers to carry information about the data. These tags are then used to protect the system and user software from security attacks and invalid information access. Research has shown that tagging schemes can be used to enhance the traditional protection mechanisms of microprocessors, going beyond basic memory management and supervisor/user mode (and even beyond protection rings). This dissertation summarizes the major security tagging schemes proposed in recent years, introduces a security tagging scheme, and proposes new research to implement, evaluate and improve the security tagging scheme.

The AFRL project used the open source Real-Time Executive for Multiprocessor Systems (RTEMS), a single-user and multi-threaded runtime executive as the base operating system in the security tagged architecture. Therefore, the design of the new security tagging scheme addressed key features of RTEMS that required modification in order to provide enhanced security. The tag checking and propagation rules are designed for both C language level and assembly instruction level programs. A SPARC instruction simulator has been improved to simulate instruction execution as well as tag checking and propagation. To ensure tagging rules were properly designed and that tag propagations were implemented correctly, several test cases were developed. In addition, RTEMS was expanded from a single user system to a multiuser system, and an advanced tagging scheme was designed to support multiple users. Lastly, this dissertation presents a discussion of a formal model of the security policy enforced by the tagged system and proves security properties of the proposed formalization.

Acknowledgements

I would first like to thank my advisor, Dr. Jim Alves-Foss, for his support, encouragement, and patient guidance throughout my graduate studies.

I would also like to thank my other committee members, Dr. Daniel Conte de Leon, Dr. Clinton Jeffery and Dr. Paul Oman, for their valuable comments on my dissertation.

I would like to thank all my instructors Dr. Jim Alves-Foss, Dr. Clint Jeffery, Dr. Paul Oman, and Dr. Bob Rinker for providing me with knowledge about Computer Science and building my research skills.

I would like to thank the department chair, Dr. Gregory Donohoe, and Mrs. Arvilla Daffin, Mrs. Rhonda Zenner, Mrs. Darby Baldwin, and other staff members in the department of Computer Science for their help during my study in the department.

I would like to thank all the staff of the College of Graduate Studies for their help and support throughout my study at UI.

I am indebted to all of the team members who worked with me in the past and present. I would like to thank Jessica Smith, Xiaohui He, Achala Aryal, Qinghua Tian, Cynthia Rempel, Saeede Zakeri, Stuart Steiner, and Abhay Patil for their input at various stages of my research.

I would like to thank many other friends who made my life at UI very enjoyable.

I wish to acknowledge the United States Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), for supporting me during the course of my graduate studies through grant number FA8750-11-2-0047.

Last, but certainly not least, I would like to thank my parents and my husband, Yinghao Wang, for their understanding, support, encouragement, and love, all of which have helped me make this dissertation a reality.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Computers play an increasingly important role in modern life and it is now widely recognized that people need to pay more attention to computer security issues. Over the years, researchers and developers have devised various techniques including encryption, firewalls, and virus scanners to provide secure computing environments. The idea of enhancing these mechanisms by using hardware to provide security features for operating systems and user applications is not new. Decades ago researchers proposed techniques to add security labels (tags) at the hardware level to help with the enforcement of system security properties [5, 10]. Unfortunately, these techniques required more computing resources and memory than was feasible at the time. As hardware speeds have improved, the idea of security tagging has resurfaced as a promising mechanism for enhancing security.

The research in this dissertation grew out of an Air Force Research Laboratory project focused on the development of a new hardware-based security tagging processor architecture and a supporting operating system. It is part of a larger University of Idaho tagging project called UITags. There are many types of data tags in use, and being investigated, from hardware-based tags up through attribute tags on data in a cloud server. Data tags are used during both runtime execution and security decision making, as well as during system implementation and analysis. The purpose of the UITags project is to investigate the use of data tags by security mechanisms in support of system security policies. The specific hardware based tagging described in this dissertation is referred to as the UI tagging scheme.

There are several operational scenarios where tagging could be used. For the purposes of this work, consider a simple run-time executive that supports execution of tasks for one or more users, or one or more applications in different security domains. This could be part of an embedded control system, or a network appliance. As the system receives data from the network, or from user applications, the hardware-based security tagging

scheme ensures that the core software of the run-time executive is protected from user software and from malicious network inputs. In addition, the hardware provides separation and information-flow controls that facilitate access control mechanisms of the run-time executive, as well as protecting the system from buffer overflows. Such support would simplify the design of a runtime executive, giving it the security features of a full operating system without the additional software overhead.

Security tagging schemes attach labels to memory and/or registers to carry information about data during program execution. These labels are also called tags. They have been used by researchers to ensure that the semantics of computations are correctly implemented; to isolate code and data, users and system; and to enforce security policies at the hardware level. The implementation of tagging in hardware provides developers with enhanced security mechanisms without a penalty on performance, as compared to traditional microprocessors. Therefore, tagging schemes are seen as promising mechanisms to help processors and OSs implement security properly.

Modern security tagging schemes were first designed and implemented to protect against a few low-level attacks, such as buffer overflow and format string attacks [16, 18, 27, 25]. Recently, security tagging schemes have been developed that claim to prevent high-level attacks, which include SQL (Structured Query Language) injection and cross-site scripting [6, 13]. Tags are also implemented in some architectures to support memory access control [26, 29, 20, 19]. To support and speed-up the tagging schemes, several hardware-based security tagged architectures have been proposed [26, 25, 13, 20].

The goal of this dissertation is to design, implement, and evaluate a new security tagging mechanism for a "zero-kernel operating system" (ZKOS [20]) (See Section 2.1) running on the new security tagged architecture (STA). RTEMS [15], which is a run-time executive system, was chosen to evaluate the new security tagging scheme, since it has been designed with almost no security protections. The tagging scheme secures RTEMS by providing access control for code and data by using an efficient hardware-

based tagging scheme. Unfortunately, as of the writing of this dissertation, this type of specially designed hardware is not fully functional. Therefore the implementation of our tagging scheme was performed in a hardware simulator.

In this chapter, Sections 1.1 and 1.2 introduce basic concepts related to security tagging schemes and security tagged architecture. Section 1.3 indicates the problems that the proposed security tagging scheme was designed to solve. The motivation and objectives of this research are presented in Section 1.4. Section 1.5 concludes with an overview of the remaining parts of the dissertation.

1.1 Security Tagging Schemes

The idea behind security tagging schemes is to attach labels to memory and/or registers which will carry information about the tagged data. These tags are checked and propagated throughout the system during runtime. This section introduces several security tagging schemes that have been presented in the literature. The purpose of using these approaches can be divided into two categories: using security tags for attack prevention and using security tags for access control. The review presented here first appeared in detail in an MS thesis by this dissertation's author [22] and was summarized in a conference paper [21].

1.1.1 Implementing security tags for attack prevention

This section provides information about tagging schemes for attack prevention. These schemes include Dynamic information flow tracking (DIFT) [25], Raksha [6], LIFT [16], Low overhead architecture for security tag [18], Invalid pointer dereference tool [27] and Decoupling dynamic information flow tracking [13]. All of these tagging schemes can prevent low-level attacks, such as buffer overflow and/or format string attacks. In addition to protection against low-level attacks, Raksha [6] and Decoupling dynamic information

flow tracking [13] claim to prevent high-level attacks, such as SQL injection and OS command injection.

In DIFT, designed by Suh et al. [25], registers and memory are tagged with a 1-bit tag to differentiate between spurious data and authentic data. All untrusted I/O inputs are tagged as spurious. Then, by using information flow tracking, all information flows from this untrusted data are tracked by the processor. If a dangerous use of a spurious value is found, the processor generates a trap to a software handler for further checks. If the spurious data is not permitted under the enforced security policy, the handler will terminate the application. This approach is focused on securing the system against buffer overflow and format string types of attacks.

LIFT [16] is an information flow tracking system that uses dynamic binary instrumentation and optimizations to detect attacks. LIFT is a software only system, requiring no hardware extensions. It uses a 1-bit tag to tag the data in memory and general data registers. LIFT provides tag checks of the data in the execution region before performing the execution and merges multiple tag checks of consecutive memory locations into one check to reduce the check overhead.

Shioya et al. [18] presented a new security tagged architecture (STA) that uses a tag table, a multilevel table, and tag caches to reduce overhead. This architecture exploits two features of tags, non-uniformity and locality of reference. Non-uniformity indicates that some of the memory cells have tags associated with them and others do not. Locality of reference means that tags are used locally in code similar to data. Therefore, tags are cached more effectively. Compared to a traditional STA, the authors show that the new architecture significantly reduces the memory overhead.

Yong and Horwitz [27], proposed a security-enforcement tool for C programs that protects against attacks via unchecked pointer dereference. This tool can identify unsafe pointers and uses a 1-bit tag to indicate appropriate and inappropriate pointer usage. It tags each byte of memory to show whether it is an appropriate location that an unsafe

pointer can point to. If the memory location being written to or freed is not tagged as appropriate, an error message is raised and the program is halted.

According to Dalton, Kannan, and Kozyrakis [6], Raksha is a flexible architecture based on DIFT. It uses information flow tracking to enforce software security. Since DIFT only uses one security policy, which focuses on preventing memory corruption attacks, it cannot detect other attacks such as SQL injection, cross-site scripting, and command injections. In order to detect these types of attacks, Raksha adds a 4-bit tag to registers, cache and memory locations. Within the 4 bits, 2 bits are used for detecting high-level attacks, 1 bit is for detecting memory corruption and the other 1 bit is for low-overhead security exceptions. Beyond buffer overflow, Dalton et al. [6] claim that Raksha is able to detect directory traversal, command injection, SQL injection, and cross-site scripting attacks. We argue that this approach just moves the problem from application-level software to the OS without any true improvement [2, 1] (see Section 1.1.3).

Decoupling dynamic information flow tracking (Decoupling DIFT), presented by Kannan, Dalton, and Kozyrakis [13], checks tags associated with the instruction's memory location. This scheme is focused on making DIFT [25] more efficient by changing the hardware. Decoupling DIFT proposes to use a small coprocessor that implements DIFT, that is, it stores register tags and tag caches, and also performs the tag propagation and tag checks. The system attaches an asynchronous coprocessor to the main processor core and synchronizes between these two cores when there is a system call.

1.1.2 Implementing security tags for access control

The previous section surveyed tagging schemes for attack prevention and information flow control. In addition to use of tags for attack prevention, researchers have also used tags as security labels for access control and protection of memory and objects. Major tagging schemes for access control include Mondrian memory protection (MMP) [26], Sentry [19], Loki [29], and TIARA [20].

According to Witchel, Cates, and Asanovic [26], Mondrian memory protection (MMP) is a fine-grained memory protection scheme that extends the traditional memory protection scheme. Compared with traditional memory protection, which uses access permissions at the page level, MMP allows arbitrary access control for individual memory words. MMP uses two permission bits to support different access control for individual words, representing four meanings: no permission, read-only, read/write, and execute/read. In addition, MMP uses a compressed permissions table and two levels of permissions cache to lower the space overheads and performance overheads.

Sentry, designed by Shriraman and Dwarkadas [19], is a virtual memory tagging scheme that provides lightweight auxiliary memory access control that can be controlled at the user level. Sentry implements a permissions metadata cache (M-cache) to intercept L1 misses. The metadata cache also controls whether data is allowed to be stored into the L1 cache. Since data stored in L1 has been checked, access to L1 cache does not need any additional checks. Each entry in the metadata cache has 2-bit permission metadata to represent access permission to a specific virtual page, representing four different access permissions: read-only, read/write, no access, and execute.

Loki [29], is a hardware architecture that supports a word-level memory tagging approach. Loki has been ported to Histar, an operating system that has a small trusted kernel. Loki tags every 32-bits of physical memory with a 32-bit tag. To avoid the high overhead incurred by using large tags, it allows a page of memory to be tagged with only one 32-bit tag. By using a permission cache (P-cache), Loki enforces fine-grained permission checks. This cache stores the security tags and a 3-bit permission, which represents read, write, and execute.

According to Shrobe, DeHon and Dwarkadas [20], Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA) is a co-design of hardware, operating system, and applications. TIARA implements access control at different levels. At the hardware level, the tag management unit enforces fine-grained access con-

trol for individual words in memory and registers. At the system level, there is an access control layer which supports a variety of role-based access control models. TIARA is an example of a ZKOS in that it has separate OS modules that run in user mode, but are protected by hardware tagging.

1.1.3 Hardware Tagging for Semantic Attack Prevention

There have been claims in the literature that some hardware tagging schemes can assist in the prevention of semantic attacks, such as SQL-injection and cross-site scripting (XSS) [6, 13]. The concept is understandable; this is user supplied data that is being processed in an unexpected manner, resulting in a violation of the security policy.

As mentioned in the previous section, prevention of a semantic attack requires proper handling of user input. This is similar to the problem of a buffer overflow, which can be handled by hardware tagging. However, there are key differences that are important to understand.

A buffer overflow occurs when there is a violation of the run-time semantics of the programming language. The semantics of the language are usually well understood and standardized. The concept of a buffer is also standard and exists across many languages. The semantics of the run-time stack and the storage of the return address on the stack are intrinsic to the microprocessor. For each microprocessor, it is then possible to develop a mechanism that detects if user data overwrote a return address, or even just overflowed a buffer.

Similar arguments can be made about uninitialized memory, integer overflows, or other microprocessor intrinsic run-time semantic violations.

Security Tagging and SQL Injection Attacks

As discussed in [2], there are limitations to hardware based tagging. Detecting an application-level injection attack is different from detecting a buffer overflow attack.

Take SQL injection attacks as an example. If we use a hardware-based security tagging approach we need a process that receives user input and tags it - similar to the buffer overflow use case. That user input is then combined with server code to create the SQL query which is then sent to an interpreter. This means that the tag must propagate with the generated query to the interpreter. At some point, probably in the interpretation of the query, security tagging hardware must evaluate the use of the tagged user data and determine if a security exception should be thrown.

What information does the hardware have to make that decision? Instructions are being executed with user data as operands. A jump or branch to an address that was specified by a user would be a violation, but normal processing of SQL queries would not run into that case. Is there something else? For example, is a comparison operation a security violation? Is addition? The answer is that the hardware cannot know the semantic intent of the operation, and there is no fundamental concepts that we can extract from query processing to configure the hardware to generate an error. Therefore, the approach taken in the literature is to cause a security exception every time user data is encountered in an SQL query, and then send the request to a software handler to manage it. The impact of this is:

- There needs to be a software handler that tags user input data. This requirement already exists in most tagging systems, and fits into the model of low-level common functionality.

- There has to be a handler that can differentiate between the processing of an SQL query and processing of the raw input data for filtering purposes. In other words, there needs to be software controls to tell the tagging hardware to monitor this processing; which requires the application programmer to be involved in configuring and appropriately using the mechanism.

- Since hardware will not be able to differentiate what part of the query processing is

occurring, all queries with user data will cause a security exception. This exception requires a handler to know that the code was processing an SQL query (and not html code for a cross-site scripting attack) and appropriately process the SQL query to remove errors. This is actually more complex than requiring the programmer to use a standard SQL query library with security features.

Given the preceding impact it is clear that we have to trust the developer to understand the underlying tagging mechanism and how to tailor it for each type of semantic attack that is a threat to their application. This is just a new flavor on the current problem and does not improve the situation for the developers.

There is a need for further research to determine common elements of high level semantic attacks and common security mechanisms for them. At this point, it appears that the high-level semantic nature of the attacks - they are attacking the functional behavior of the code - is beyond the capability of inherent security mechanisms.

1.1.4 Summary of tagging schemes

Table 1.1 summarizes the reviewed tagging schemes, indicating the types of attacks that can be prevented[1], if the scheme is used for access control and if it has hardware support. The Y's in the access control column indicate that those five tagging schemes support access control. Some of the schemes are implemented only at the software level and some need support from hardware level, which requires design of tagging engines and changes to the hardware. The UI tagging scheme is shown in the last row, details of this are provided in Chapter 3.

[1]The attack list is based on claims by the cited authors, the claims for "semantic level" attacks are unable to be verified.

Table 1.1: Summary of Tagging Schemes

	Buffer overflow	Format string	SQL injection[1]	Cross-site scripting[1]	Access control	HW support
DIFT [25]	Y	Y				Y
Raksha [6]	Y	Y	Y	Y		Y
LIFT [16]	Y					
Dereference pointers [27]	Y					
Decoupling DIFT [13]	Y	Y	Y			Y
Low-overhead tagging [18]	Y	Y				Y
MMP [26]					Y	Y
Sentry [19]					Y	Y
Loki [29]					Y	Y
TIARA [20]					Y	Y
UI tagging [21]	Y	Y			Y	Y

1.2 Security Tagged Architecture (STA)

The concept of hardware-based memory tagging can be traced back to the Burroughs B5000, which was introduced in 1961. It used a 1-bit code to differentiate between data and instructions, partially to protect from corruption of the stack. Later the 1-bit code grew to a 3-bit tag in the B6500 [1]. In 1973, Feustel proposed expanding this to 5-bit tag [11]. However, this approach has not received significant attention until recent years. As computer hardware has been getting faster and more functionality is put on chips, it is becoming easier to use hardware to provide functionality currently implemented in software.

An STA is an architecture that supports tagging capabilities such as tag storage, tag propagation, and tag checking in hardware. Therefore, by using an STA, the maintenance of correctness and security of the system can be performed at the hardware level. This reduces the consumption of system resources, provide a common platform for secu-

rity, and reduces the possibility of successfully bypassing or tampering with the security mechanism.

The STA, defined by Shrobe, DeHon and Knight [20], is a hardware architecture that allows metadata to be used during runtime to enforce security policies and to maintain the semantics of the computation. The metadata can be represented with a tag. It is usually associated with a word of memory and carries information about the word, such as access rules, permissions, and data types. There have been a few STAs designed to implement tagging schemes. Four of them are discussed below.

1.2.1 Mondrian memory protection architecture

In the architecture of Mondrian memory protection (MMP) [26], every allocated memory region is owned by a protection domain. Each running thread has a protection domain ID and each domain has a corresponding permission table. The permission table is stored in memory, it specifies the permission of each address that the domain has. Every access to memory requires a check of the permission table. The processor checks the permissions in the sidecars, which are registers that hold information for one table segment. Each address register has one sidecar register associated with it that caches the last table segment accessed by this address register. If the sidecars do not have the permissions information, the processor will then reload the sidecar from a permission lookaside buffer (PLB). If the permission information needed is not in the PLB, the permission information will be retrieved from the permissions table resident in memory, by means of hardware or software, and refilled into both the PLB and sidecars.

1.2.2 Dynamic information flow tracking architecture

DIFT [25] implements the security tagging scheme in a processor. The on-chip architecture has new structures (ITag TLB, DTag TLB, T$L1, T$L2) that support the tagging

scheme. In DIFT, each register has an additional bit to store the tag. DIFT also adds two separate tag caches, T$L1 and T$L2. Because of the introduction of the new tag caches, a new translation table between L2 cache and memory is needed. DIFT adds two new registers: propagation control register (PCR) and trap control register (TCR). PCR has a bit vector that controls the propagation of tags based on the security policies. TCR has information about whether or not to generate a trap for a specific kind of values. Both registers are configured by the execution monitor, which is a software module used to enforce the security policies. The PCR and TCR in the STA provide DIFT with the ability to evaluate and propagate tags at the hardware level and raise an exception to the software level when an error occurs.

1.2.3 Dedicated coprocessor architecture

Kannan, Dalton, and Kozyrakis [13] proposed the decoupling dynamic information flow tracking architecture based on a dedicated small coprocessor. Compared with using a separate general core, the small coprocessor is a better choice because it is sufficient for doing tag propagation and tag checking, and does not impact normal CPU (central processing unit) operations. DIFT functionality is implemented in the small attached coprocessor. The main processor core will be synchronized with the small coprocessor when doing system calls. The small attached coprocessor includes a tag pipeline for tag propagation and a tag cache to store and maintain the tags.

1.2.4 TIARA architecture

In TIARA[20], the Tag Management Unit (TMU) runs parallel with the data path. It operates on the tag of the data and makes decisions on whether or not to send a trap signal based on the enforced security policies. The TMU takes certain fields of the operands, the program counter, and the instruction to compose the tags for the data. In

TIARA, every word in memory and all of the processor registers are tagged with their data type and security context information. In addition, the program counter is tagged, which helps handle conditional branch instructions. TIARA implements a policy table in main memory. When the main data path is executing an instruction, the TMU deals with the tags of the two operands, the program counter's tag, the instruction, and the principal register, which stores the privileges of the running process. If the execution of the instruction does not violate the access control policies, the resulting tag is derived from the tags of the input operands. However, if the execution violates the security policies, then the TMU forces the process to the trap handler.

1.3 Security Problems

Due to the increasing reliance on technology, a wide variety of software applications are installed on computers, smart phones, tablets and other computing devices. As attacks can exploit the vulnerabilities either in operating systems or in the installed software, security issues of computer systems have become one of the biggest concerns in today's systems. This section summarizes some common security attacks in today's computing environments. They can be addressed by the UI tagging scheme proposed in this dissertation (see Chapter 3), and some can also be addressed by other tagging schemes found in the literature.

- **Memory corruption attack.** A memory corruption attack causes the system to violate the standard programming language semantics by overwriting unexpected areas of memory. The buffer overflow attack is the most common type of memory corruption attack. If the data written to a buffer exceeds the capacity of the buffer, the extra data will overwrite the memory space adjacent to the allocated buffer. The attack stems from insufficient bounds checking. If the memory where the return address is stored is overwritten, then once the currently executing function

returns, execution will resume at the address specified by the attacker, which can result in the execution of malicious code. This kind of attack can be prevented by security tagging schemes such as DIFT [25], Raksha [6], LIFT [16], Low overhead architecture for security tag [18], Invalid pointer dereference tool [27], Decoupling dynamic information flow tracking [13] and the new UI tagging scheme proposed in this dissertation.

- **SQL injection.** SQL injection attack may be used to attack a website by sending malicious SQL statements to it, so as to get the defective website to release private contents of the database to the attacker. This attack is due to the use of badly designed query language interpreters. When a user input is incorrectly filtered or not strongly typed, malicious SQL commands sent to the website's software can be used to exploit security vulnerabilities of the website. According to their authors, SQL injection can be prevented by using tagging schemes like Raksha [6] and Decoupling dynamic information flow tracking [13]. However, this claim is disputed in Section 1.1.3 [2, 1]. Software extensions are being explored in the UI tagging project to address this problem.

- **Data isolation.** Attackers may forge pointer values to access the contents of unauthorized memory space, such as critical memory blocks owned by the operating system. The attackers can then use this access to read from or write into the memory space pointed to by the forged pointer. If the system does not check the validity of the pointer and allows the attacker to write the value of the memory space, then the attacker may gain complete control of the system. This attack can be prevented by the UI tagging scheme, Invalid pointer dereference tool [27], and memory access controls such as those found in TIARA [20].

- **Forged index or reference value.** Most operating systems manage a large number of resources for multiple users. System calls used to access these resources

are usually passed through a reference or index to indicate which resource the user wishes to access. It might be possible for an attacker to forge an index which refers to a resource not owned by the attacker. If the system does not validate that the reference belongs to the user, the attacker could force the system to corrupt, delete, or release contents of this resource. To prevent this kind of attack, access controls are need. This attack can be prevented by the UI tagging scheme (see Chapter 3).

1.4 Motivation and Contributions

This dissertation describes the implementation and verification of the new hardware based UI tagging scheme, a related security policy, and its implementation in a real-time operating system to support a new STA microprocessor. It is part of a larger project investigating the properties of security tagging schemes and operating system support for them. This dissertation is an extension of an earlier MS thesis by the dissertation author [22].

The initial objectives of the project team are listed below for context and completeness, and which objectives were achieved by this dissertation are pointed out.

- Objective 1: Evaluate existing tagging schemes. Many different security tagging schemes have been proposed and implemented. The understanding of the current scope of tagging schemes, their strengths and weaknesses, is important for the designing.
  - Task 1.1 Survey existing tagging schemes.
  - Task 1.2 Compare the different tagging schemes.
  - Deliverables: This objective is complete; results are documented in the MS Thesis [22] and in a conference paper [21].
- Objective 2: Evaluate the RTEMS source code to find places for security enhancements. RTEMS is a set of runtime executives which provide different functionalities.

Reviewing the source code of RTEMS helped in understanding its architecture and how the system works. This understanding was used to guide the modifications of RTEMS in support of the new tagging scheme.

  - Task 2.1 Review RTEMS security implications.
  - Task 2.2 Define changes that need to be made to RTEMS to enhance security.
  - Task 2.3 Document features the tagging scheme will need to support RTEMS security.
  - Deliverables: This objective is complete; results are documented in the MS Thesis [22] and in a conference paper [21].

- Objective 3: Develop a new security policy and corresponding tagging scheme for RTEMS. A new security tagging scheme has been designed to address the current security concerns in RTEMS while using as much hardware support as possible to keep performance overhead low. This includes designing the bit representation of tags, developing the security hierarchy of the system and specifying rules for it and the C language level tagging rules.

  - Task 3.1 Design the security tagging scheme.
  - Task 3.2 Design C language specific rules for tagging.
  - Task 3.3 Evaluate the security of the new tagging scheme.
  - Deliverables: The first two tasks of this objective were documented in the MS Thesis [22] and in a conference paper [21]. The third task is addressed in this dissertation.

- Objective 4: Apply the tagging rules to SPARC (Scalable Processor ARChitecture) instructions at the assembly language level. The new security tagging scheme will be eventually implemented in the SPARC architecture by the Cornell team partners. Therefore, whether or not the design is able to be implemented on the

hardware architecture needs to be considered. The review of the assembly code generated by GCC revealed that the previous C language level tagging definition is incomplete and a refinement on the definition to enable the hardware implementation is required.

  - Task 4.1 Enhance the tagging scheme with respect to SPARC.
  - Task 4.2 Design SPARC ISA specific rules for tagging.
  - Deliverables: This objective is complete; results are documented in the MS Thesis [22] and in a journal paper [23].

- Objective 5: Add a new tag manager in RTEMS to support the UI tagging scheme. To support tagging, a tag manager is needed in RTEMS to allow the OS to handle the tagging issues, for example checking if the copy bit is set or not in some specific functions and take or release the ownership of some data.

  - Task 5.1 Define the APIs of the tag manager.
  - Task 5.2 Implement the tag manager in RTEMS.
  - Task 5.3 Test and evaluate tag manager (completed as part of Objective 7).
  - Deliverables: This objective is complete; results are documented in the MS Thesis [22] and a project status report [2]. Source code is available.

- Objective 6: Expand the tagging scheme to support multiple users. RTEMS needs to be enhanced to support multiple users. Therefore, a user manager needs to be implemented in RTEMS to make tagging work properly for multiple users.

  - Task 6.1 Develop a model of user tags and support in RTEMS.
  - Task 6.2 Develop APIs of the user manager.
  - Task 6.3 Implement User manager in RTEMS.
  - Task 6.4 Test and evaluate user manager.

  - Deliverables: This objective is complete; results are documented in this dissertation, a project status report [2] and source code is available.

- Objective 7: Implement the UI tagging scheme in the SPARC simulator[2]. The SPARC instruction simulator was modified to make it support the modified RTEMS and the security tagging scheme. New instructions needed to be implemented to support tag checking and tag propagation. This objective is in partnership with other team members.

  - Task 7.1 Add hooks and additional instruction interpretations in SIS to support a tagging coprocessor.
  - Task 7.2 Implement tagging coprocessor in SIS.
  - Task 7.3 Test and evaluate tagging coprocessor simulation comparing with theoretical model of tagging scheme.
  - Deliverables: This objective is complete for both the initial tagging scheme and the multiuser tagging scheme. The test cases (over 2000 test cases) provide a coverage test of simulator functionality and tagging rules, which resulted in some changes to the tagging scheme and the simulator. Additional changes to the simulator and the tagging system have been finished. The final deliverable includes documentations in an MS thesis [28] and this dissertation, project status report [2] and source code.

- Objective 8: Implement test cases to evaluate application level use of the tagging scheme and RTEMS modifications. This objective is in partnership with other team members.

[2]The overarching AFRL project involved a partnership with Cornell University. The team at Cornell was tasked with implementing the tagging scheme in an FPGA model of the SPARC process. Changes in the scope of work and funding resulted in an incomplete implementation that halts the machine upon a single security exception. This made testing very difficult, and resulted in a move to the use of a software simulator for the tagging engine.

  - Task 8.1 Port sample test programs into RTEMS (including some of the SPEC CPU benchmarks), to assist in evaluation of new tagging scheme.
  - Task 8.2 Evaluate and implement changes needed in RTEMS libraries to support applications.
  - Task 8.3 Run and document tests.
  - Deliverables: The deliverable will be a process document on how to port benchmarks and enhance RTEMS libraries, a separate technical report and source code (when permitted - for example we cannot release SPEC benchmark source code).

- Objective 9: Develop Formal Models of the security policy enforced using the new tagging scheme, and prove security properties of the system architecture. The intent of these models is to allow people to understand and document the pros and cons of this and related tagging schemes.
  - Task 9.1 Develop formal model of tagging scheme, prove related lemmas.
  - Task 9.2 Develop formal model of abstract system architecture (a model of a multiuser RTEMS using tagging) and prove related lemmas.
  - Task 9.3 Develop formal model of the security policy(s) supported by this architecture and tagging scheme and prove related lemmas.
  - Task 9.4 Prove that the formal model supports the security policy(s) within the abstract architecture.
  - Deliverables: Source code is available, results are documented in this dissertation, a project status report [2], and academic paper(s) in preparation.

In summary, the UI tagging scheme presented in this dissertation provides a way to protect the system and user software from security attacks and invalid information access by using tags. The UI tagging scheme proposed is designed for the RTEMS

runtime operating system to provide security enhancements to RTEMS. It expands prior work from a MS thesis [22]. In the earlier work, RTEMS source code was examined to evaluate potential security problems and tags were designed to fix these problems for the system. In addition, the studies of the SPARC architecture refined the security rules that was proposed for the C programming language at software level to SPARC assembly language instructions. By being implemented in hardware, it will achieve a performance increase as well as a common security mechanism to support all of the code, alleviating the OS (Operating System) from the responsibility of ensuring that all OS code correctly implements security features.

During this dissertation work, RTEMS was expanded to a multiuser system. As a result, the tagging scheme required a modification to support multiple users. In order to handle multiple users, a new USER manager needs to be added to the system. To check that the new designs and implementations are correct, test cases are generated to test tag checking and propagation. Lastly, by using A Computational Logic for Applicative Common Lisp (ACL2), a formal model of the security policy enforced by the UI tagging scheme has been created and several related security properties and theorems have been proven.

1.5 Dissertation Overview

The remainder of this dissertation is organized as follows. Since the new tagging scheme is going to be implemented in RTEMS, Chapter 2 provides background on the zero kernel operating system concept, RTEMS and its architecture. Modifications are going to be made to the SPARC architecture to support the new tagging scheme and it has some special structures that are important to understand. Therefore, the SPARC architecture and the SPARC instruction simulator are also discussed in Chapter 2 as well.

Chapter 3 presents the 32-bit tag and the security tagging scheme. This chapter defines the format of the tag and illustrates how each field in the tag is used to help

secure RTEMS. This chapter also introduces a lattice which illustrates the security classes and the information flow in RTEMS. The information flow rules for the C programming language semantics are also discussed in detail. In addition, because some of the high level rules cannot be mapped to assembly level correctly, the tag checking and propagation rules for assembly instruction level are also explained.

Chapter 4 presents the work expanding RTEMS from a single user system to a multiple user system, adding code to the USER manager and TAG manager to support tagging for multiple users, and testing the correctness of tag checking and propagation rules and implementation.

A formal model of the security policy enforced by the UI tagging system is set up successfully in Chapter 5. Based on the formal model that has been set up, by using ACL2, security properties of the tagging system and related theorems were proved.

Chapter 6 concludes the security tagging system work that has been done. It is followed by the future work section, which proposes the possible future work.

In summary, this dissertation proposes an extension of the UI tagging scheme which supports fine-grain access control, and multiple users by using tags. The tags are associated with memory and registers, by giving specific tag checking and propagation rules, they could help prevent security attacks and invalid data access. This dissertation describes the process of implementing the tagging scheme, initially from a high level and then down to the assembly instruction level. The UI tagging scheme provides a way to isolate the OS modules from users and also further isolate different OS modules and different users in the system. It can be implemented in a general OS to enforce the information isolation and access control. In addition, a formal model of the security policy enforced by the new tagging scheme is given. It helps with proving related properties and theorems.

Chapter 2: Background

This chapter provides background on the concepts of zero kernel operating systems (ZKOS), RTEMS (Real-Time Executive for Multiprocessor Systems) and the SPARC (Scalable Processor ARChitecture) architecture. The basic idea of a ZKOS is to decompose the whole system into many smaller components. Each component is isolated from other components and has its own privileges, therefore avoiding having ultimate privileges on all of the system code. The operating system components run as part of user code, like library routines, but with special protections provided by a tagging mechanism. This structure avoids the need for an OS kernel and associated overhead, such as context switches.

The goal of the UI tagging project is to design and evaluate a security tagging mechanism for implementation in an operating system running on a security tagged architecture. RTEMS [15] is a real-time executive that has 18 managers, each of which can be viewed as an individual module. Since the architecture of RTEMS is very similar to the idea of a ZKOS, RTEMS was chosen to implement the new security tagging scheme. Because RTEMS implements almost no protection, the purpose of the new tagging scheme is to secure RTEMS without requiring a major rewrite and to reduce the overhead incurred by implementing the tags in hardware.

The Cornell University team on this project was implementing the tagging scheme on a SPARC processor. Therefore, this chapter highlights important features of the SPARC processor that must be noted in the design and evaluation of the UI tagging scheme. For example, SPARC uses a sliding register window to help pass parameters to subroutines and return values to the caller. The PC (Program Counter), parameters, return value, frame pointer, and stack pointer are stored in specific registers within the window. For that reason, this style of parameter passing must be taken into account.

In this chapter, the basic concepts of a ZKOS are introduced in Section 2.1. Section 2.2 gives a brief introduction to the RTEMS system and its important data structures. In the

last section, Section 2.3, the SPARC's architecture, register window and memory stack are briefly introduced. The SPARC instruction simulator is discussed in Section 2.4. And Section 2.5 summarizes this chapter.

2.1 Zero Kernel Operating System

A good operating system protects system code from modification by user or software. Conventionally, as shown in Figure 2.1, to satisfy this requirement, systems separate memory into user space and kernel space. To further control separation, systems also support the concept of virtual memory, which allows user code to run unmodified as the operating system reallocates memory regions or even swaps memory out of RAM to disk. Programs running in kernel space typically have ultimate privileges, while programs running in user space have only limited privileges. To be able to support the separation of user space from kernel space, separate kernel and user execution modes are traditionally required. However, the transition between kernel mode and user mode, or between different users' spaces, namely the *context switch*, incurs a huge performance overhead [20], as registers need to be saved and restored and memory maps need to be reloaded. In addition, the current implementations of virtual memory cause multiple memory accesses upon page misses.

Shrobe, DeHon, and Knight [20], designed a new operating system architecture called a zero-kernel operating system (ZKOS) to address performance and security issues of current operating systems. Along with the support of a new security tagged architecture, a ZKOS provides fine-grained memory protection by using tags. As shown in Figure 2.2, a ZKOS decomposes the whole operating system into many smaller components based on an object-oriented design philosophy. In their ZKOS implementation, TIARA, each component has its own compartment and principal. A compartment defines the resources and memory used by the compartment to store data and the principal specifies the authority of operations. A component's principal can access the data of its own compartment, but

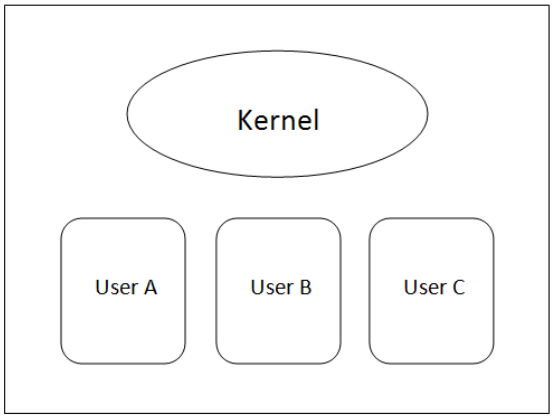

Figure 2.1: Traditional operating system model

cannot access other components' data and user's data. If data needs to be shared among two or more components, then shared compartments are used to support it.

In traditional memory protection, protected and supervisor modes give the programs running in supervisor mode ultimate privileges and the programs running in protected mode limited privileges. Different from the protected mode and supervisor mode, a ZKOS decomposes the whole operating system into many smaller components, each having its own limited privileges. This ensures that each component has the least privileges needed, preventing any of the components from having complete authority. Therefore, even if a component is compromised, the whole system will not be compromised. By using smaller components, the system has better isolation in code and data and also a better separation of the system from users. According to Shrobe, DeHon, and Knight [20], because the components have their own privileges and compartments, a ZKOS supported by a security tagged architecture avoids the expensive context switches between kernel mode and user mode that occur in traditional operating systems. What is more, the decomposed components help ensure fine-grained memory protection of the system.

Shrobe, DeHon, and Knight [20] require that each component consist of not only compartments and principals, but a set of access rules as well. Access rules are used to restrict access from other components that do not have the appropriate privileges.

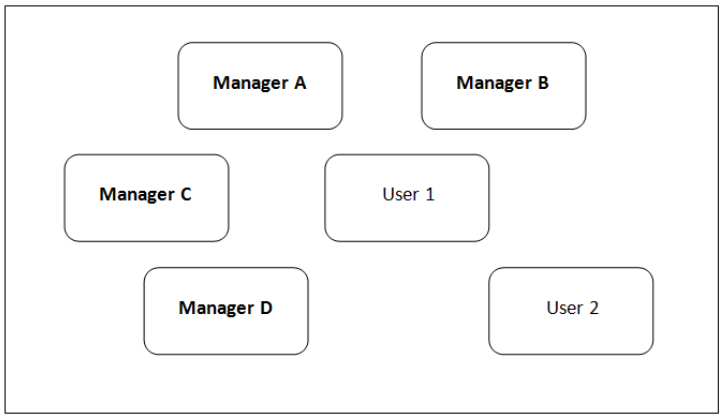

Figure 2.2: Zero kernel operating system model

Therefore operating system components are unable to access data in other operating system compartments and even the user's compartments. The interaction and sharing among different ZKOS components are also well structured. In simple terms, each component establishes a set of gates that include a procedure, a compartment, and a principal to indicate the entry points of the service. With the help of the TMU (discussed in Section 1.2.4), the invocation of the gate is only allowed by the components that are authorized.

In summary, compared with a traditional OS, a ZKOS allows better separation of user data and code from the system data and code, and a better separation between system modules. It also avoids the expense of performing context switches between kernel mode and user mode. The components turn the system into a multilevel system, which helps provide better control of information flow. With the support of an enhanced security tagged architecture, a tagged ZKOS would be more flexible and powerful [20].

2.2 RTEMS

RTEMS [15] is a real-time executive that provides a runtime environment to allow various types of services and applications to be embedded. It can perform a set of services that

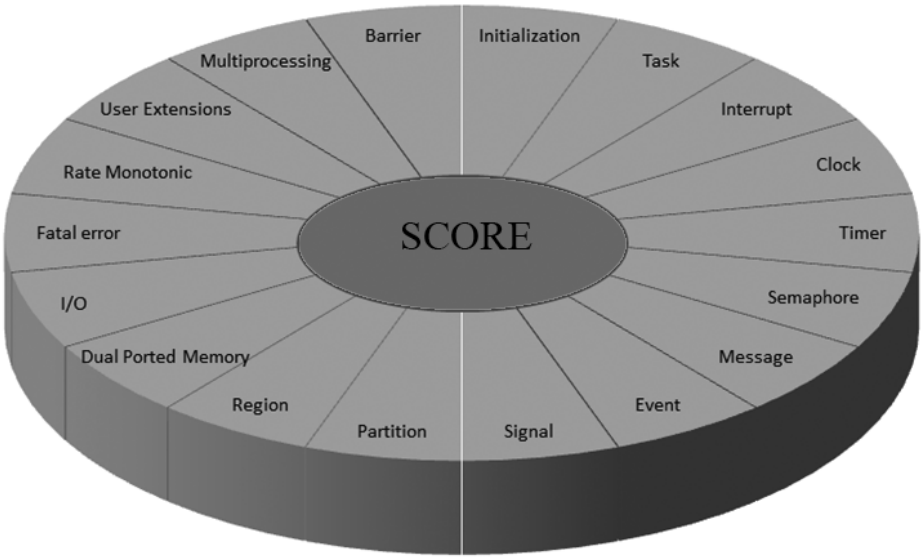

Figure 2.3: RTEMS Architecture

includes multitasking, inter-task communication, and dynamic memory allocation. There are two versions of RTEMS, one version is written in the ADA programming language and the other one is written in the C programming language; the C language version has been modified and ported to other processor families, such as ARM, MIPS, PowerPC, SPARC, Atmel AVR etc. In this project, the C language implementation was used for testing and evaluation.

2.2.1 RTEMS architecture

As shown in Figure 2.3, RTEMS consists of a super core (SCORE) and 18 managers. The 18 managers provide different services and the SCORE provides kernel functions that support the managers. In this dissertation, two new managers were added, Tag manager and User manager, to support the UI tagging scheme.

The original 18 managers of RTEMS provide services to user code in support of

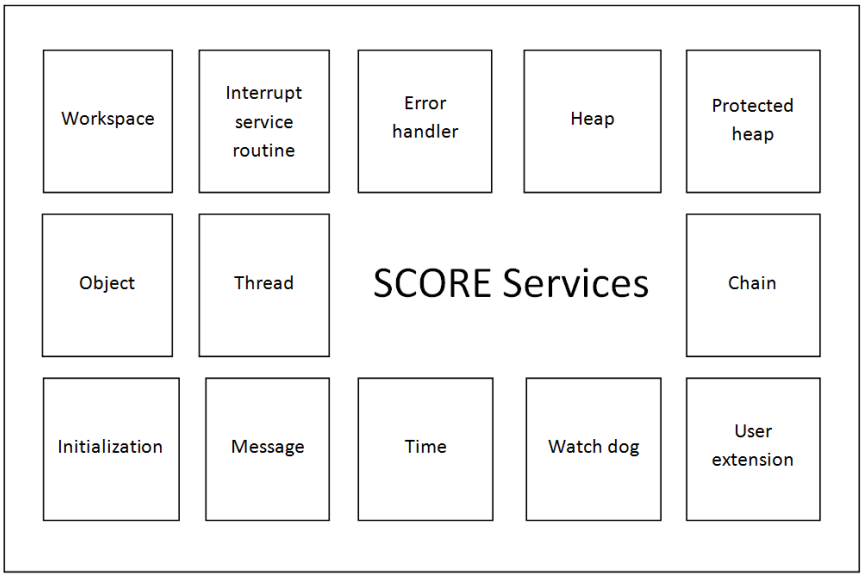

Figure 2.4: SCORE services

concepts such as tasks, memory, timers, and communications. Each manager provides a well-defined set of services through *directives* that take the place of traditional system calls. Each manager also has internal functions that it uses to support implementation, some of these are private to the specific manager, and some are intended to be used by other managers. In either case, these manager internal functions are not intended to be used by user code.

In addition to each manager, RTEMS has a set of modules that make up the SCORE subsystem. The SCORE provides services for all managers, and all managers interact with the SCORE via directives. As shown in Figure 2.4, some major SCORE modules are: object, thread, chain, interrupt service routine (ISR), error handler, heap and protected heap, workspace, initialization, message, time, watch dog, and user extension. These modules are key to the internal working of RTEMS, but are not intended for use by user code.

Conceptually the APIs are meant to be externally accessible, internally restricted to

other RTEMS modules or internal to specific modules. However, RTEMS currently does not restrict access to any of these functions or their private data structures. The following sections outline the major security concerns that need to be addressed to secure RTEMS.

2.2.2 Security Concerns in RTEMS

The examination of RTEMS's source code reveals several major security concerns.

No separation between user and system. In RTEMS, there is no separation between user resources and system resources. In traditional operating systems, the system has ultimate privileges while the user has limited privileges. In RTEMS, both the user and system have ultimate privileges. Currently user code can use all of the SCORE code, internal functions, and directives. This means that the user can do everything that the RTEMS system can. For example, the user has privileges to change the system configuration, delete system tasks, etc. This is a design decision since RTEMS is intended for embedded systems and is built for high performance. Unfortunately, this limits the use of RTEMS in secure environments or from safely running untrusted code.

As critical system data must not be changed by users, user code should be separate from system code and user data should be separate from system data. Therefore, a secure RTEMS needs a mechanism to distinguish user data and system data first, and then to restrict users to the minimal privileges.

System has no protection from users. RTEMS has no protection from users and no concept of multiple users. Take the directive, `rtems_task_delete`, as an example. There is no check of the identity of the caller of this directive. This means that other system managers can use this directive, other system code can use it, and even users can delete any task by calling this directive. There is no restriction on who can delete which task, which means that users could even delete system threads.

As RTEMS has other directives and internal functions that can delete, create, or change user data and system data, it is very dangerous to let users have permission to use these important system functions. Therefore, in order to protect system data from users, it is necessary to implement a mechanism to limit the capabilities of users.

No separation of RTEMS code. There is no isolation among RTEMS managers, directives and SCORE. In RTEMS, SCORE works as the micro kernel of the system. If the SCORE code is misused by a user or attacked by a malicious user, critical security problems may occur. Therefore it is important that SCORE code be separated from other system code and user code.

RTEMS has 18 managers and each manager has its own functionality. Each manager has specific directives that support the manager. RTEMS has internal functions that help RTEMS managers function correctly, inline routines that speed up the running of RTEMS, and library functions supporting some of the functionalities. In the current version of RTEMS, every directive of a RTEMS module can call any function, including internal functions of other RTEMS modules. This may cause security problems if any application in user code is modified by an attacker, because the attacked code can use any other RTEMS code that the attacker wants. With a STA, mechanisms can be implemented in RTEMS to prevent unintended or malicious usage of manager and SCORE code.

No separation of different users. Although RTEMS has a single user multi-threaded model of execution, it can be expanded to be a multiuser system. To be extended to a multiuser system, the system must have a method to identify the owner of user data and keep it separated from other users' data. However, RTEMS cannot tell which user is the owner of specific user data. For a thread, it might be a thread generated by user 1, a thread generated by user 2, or a system thread. Since the owner of the thread is unknown, who has the permission to change or delete it can not be specified. Currently

RTEMS is a single user system that contains no rules to specify who can do what. This becomes a problem when RTEMS is extended to allow more than one user.

2.3 SPARC Architecture

The Cornell team chose to use SPARC as their target architecture [7], therefore it is used in this project. SPARC has an instruction set architecture that is derived from a reduced instruction set computer (RISC) lineage [24]. RTEMS is designed to be portable to multiple processor architectures, and SPARC is one of them. SPARC provides flexible register window management by using separate instructions for window management and function call and return. The main features of SPARC are:

- The address space is 32-bit, linear.
- All instructions are 32 bits, and the number of instructions is reduced.
- A separate floating-point register file.
- A program can see a register window which contains 24 registers and 8 global registers at any time.
- The processor fetches the next instruction after a delayed control-transfer instruction.
- The architecture defines a coprocessor instruction set.
- It has instructions for a synchronizing multiprocessor.
- A trap causes the allocation of a new register window.

2.3.1 SPARC register set and register windows

At a given time, an instruction can see 8 global registers and a register window which contains 24 registers: 8 `in` registers, 8 local registers and 8 `out` registers. `In` and `out`

registers are used for passing parameters and getting return values, because the caller's `out` registers become the callee's `in` registers when there is a normal function call[1]. Registers `%i0-%i7` are `in` registers. The first six `in` registers (%i0-%i5) are used to store incoming parameters, `%i6` is for storing the frame pointer (`%fp`), and `%i7` is for storing the return address. In addition, `%i0` is also used to store the return value to the caller. Registers `%l0-%l7` are local registers that are mostly used for temporary values. The current PC and the next PC (nPC) are copied to `%l1` and `%l2` when a trap occurs. Registers `%o0-%o7` are the `out` registers. Among these registers, `%o0-%o5` are used to store the parameters being passed to a called subroutine, `%o6` stores the stack pointer, and `%o7` stores a temporary value or the return address. The `CALL` instruction writes its own address into `%o7` and the address is then used as the basis for the return address for the callee as `%i7` when the register window shifts. The `global` registers are `%g1-%g7` and have global scope. The `%g0` register has the hardwired value of zero; any write to it has no effect.

In the SPARC architecture, partially overlapping windowed integer registers are provided. The current window pointer (CWP) indicates the current window and walks a circular buffer of windows[2]. The execution of instruction `SAVE` or trap decreases the CWP by 1 and instruction `RESTORE` or `RETT` increases the window by 1. The overlap of windows allows each window to share its `in` and `out` registers with adjacent windows. As shown in Figure 2.5 (adapted from SPARC International Inc. [24]), when the CWP decreases by one, the `out` registers of the previous register window become the `in` registers of the new register window and the caller's stack pointer (`%sp`) becomes the current procedure's frame pointer (`%fp`). When executing `RESTORE` or `RETT`, the CWP increases by one and the window moves back to the previous window. The `local` registers are unique to each window. A return value is stored in `%i0`, therefore, after moving the register window

[1]The `CALL` instruction does not cause this register mapping, rather the `SAVE` instruction slides the register window – allowing more flexibility for the compiler, but decoupling parameter passing from the actual function call. The `RESTORE` instruction is used to slide the window back before a return.

[2]The Leon3 processor, the SPARC processor used in this project, supports 32 register windows.

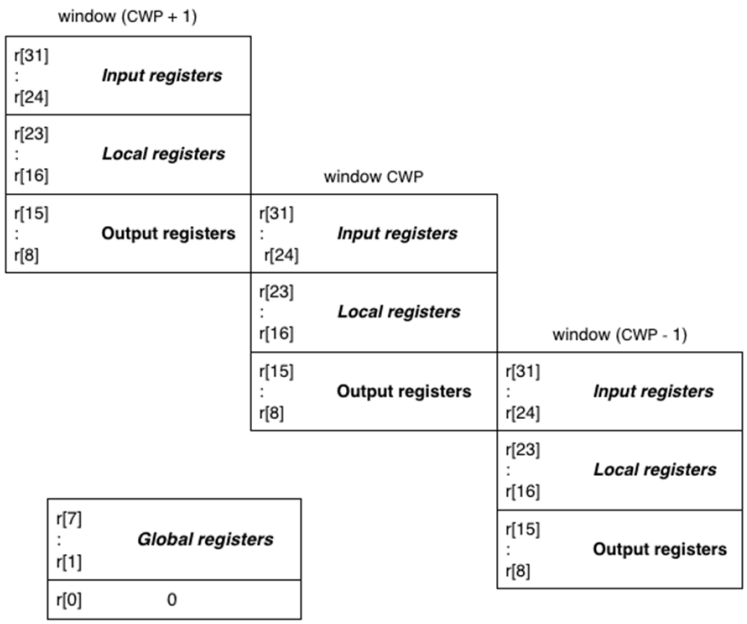

Figure 2.5: Change of the register window

with a RESTORE, the caller can access the return value that is stored in its `out` register `%o0`, which was callee's `in` register `%i0`. Window overflows and underflows cause a trap to the operating system.

2.4 SPARC Instruction Simulator (SIS)

The SPARC Instruction Simulator (SIS) [9] was developed by European Space Research and Technology Center (ESTEC). It has now been incorporated into the GNU Debugger (GDB) [12]. SIS emulates an ERC32 system (a variant of the OpenSparc processor), containing the IU, FPU, MEC, up to 16 MByte ROM and up to 32 MByte RAM. The source for the simulator is included as part of the standard GDB source distribution.

In a companion project to this dissertation, a modified SIS is used to allow for simulation of a variety of tag engines. A tagging coprocessor has been added to the simulator [28]. Hooks were placed inside SIS that call a set of tagging engine functions with instruction information, and implemented a set of coprocessor instructions that assist in

programming the tagging coprocessor. The tagging functions perform the relevant propagation and checking operations, as defined by the specific tagging rules. If an error is detected, the tag coprocessor throws an exception, which is then handled by the regular simulator. The implementation of the tagging rules specified in this dissertation was part of the work for the corresponding MS Thesis [22] and were further modified as part of this dissertation to support multiple users, and in response to tests conducted as part of this dissertation. The following assumptions were made with respect to the simulator:

- All exceptions are precise. When a tagging error occurs, the exception will be thrown during the instruction that generated the exception.
- Exceptions prevent instruction completion. When an exception is thrown, the instruction is not completed, and no changes are made to the registers or memory.
- The exception handler is part of the RTEMS code.

2.5 Summary

The chapter provides background on the UI hardware-based security tagging project. It first introduces the concept of ZKOS. The idea of a ZKOS is to decompose the whole system into smaller components. Supported by a security tagged architecture, a component has its own privilege and is isolated from other components. RTEMS was chosen to implement the new security tagging scheme because its architecture is very similar to the ZKOS's. RTEMS has 18 managers and 13 SCORE services, each of the manager or SCORE service can be seen as a component of the system. By using tags, each manager or SCORE service can be isolated from other managers and SCORE services. In addition, different privileges can be given to a manager or SCORE service based on its role in the system.

A SPARC processor was used by the Cornell University team to implement the tagging scheme on it. Therefore, the important features of the SPARC processor are discussed

in this chapter. Understanding these features help with the design and implementation of the UI tagging scheme. Because the hardware tagged architecture was not available, a SPARC instruction simulator is used in this project. Modifications have been made on the SIS to make it support the UI tagging scheme at the instruction level.

Chapter 3: UI Tagging Scheme Design

As discussed in the evaluation of the source code of RTEMS, RTEMS has many security concerns that need to be resolved if it is to be used in a multiuser, secure environment. This chapter proposes a three-field tag consisting of the Owner field, the Code-space field, and the Control-bits field to be used in a hardware based tagging scheme in support of security for RTEMS. Each of these three fields helps maintain the correctness and security of RTEMS. This tagging scheme will allow the use of the principle of least privilege [17] in protecting resources in the system with a finer-granularity than traditional hardware-based supervisor and user modes.

In addition to using tags to control data access, the idea of access control is implemented in the tagging scheme to control the execution of functions. To prevent malicious use of critical functions, specific rules are provided to specify who can call which functions. A security hierarchy in a lattice is defined to support the access control mechanism and information flows within RTEMS.

The security tagging scheme was first designed to be implemented in the C language. Tagging rules for different statements and operations in the C language are given to check tags and propagate tags. On an execution of a statement or an operation, the corresponding tagging rules are checked first to ensure the statement or operation is allowed to be executed. If it is allowed, then the tagging rules help propagate the tags.

Since one goal of this project is to implement tagging on the SPARC processor, the important features of SPARC, such as register windows and special purpose registers, have to be considered. What is more, at the assembly language level there are more things that need to be considered, such as the runtime stack, use of registers, and the program counter (PC). As a result, the remaining part of this chapter provides an assembly language level implementation of the tagging scheme. Rules are provided for assembly language instructions that have direct C language counterparts.

The remainder of this chapter is organized as follows: The design of the tag format

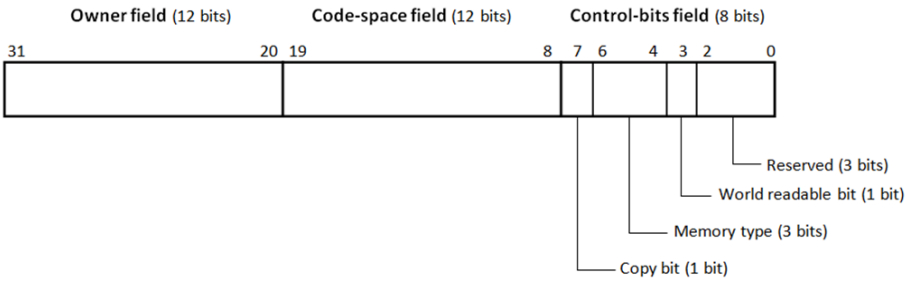

Figure 3.1: Tag format

is illustrated in Section 3.1. The security lattice is presented in Section 3.2. Section 3.3 discusses the low level bit representation of the tags, and the tagging rules for C language are in Section 3.4. Section 3.5 presents the SPARC instructions and divides the instructions into four groups, with associated tagging rules. Some concerns about the security of the runtime stack are discussed in Section 3.6. Lastly, Section 3.7 summarizes this chapter.

3.1 Tag Format and its Composition

As shown in Figure 3.1, the tag consists of three fields: Owner, Code-space, and Control-bits. The tag can be written as (⟨`Owner`⟩, ⟨`Code-space`⟩, ⟨`Control-bits`⟩).

The second field of the tag specifies which code space can manipulate the data. For example, the task manager can create tasks and task data structures which will be tagged with ownership from the calling user, and tagged with the task manager code space. A subsequent call by another user cannot have the task manager access data structures tagged for a different user even if manipulated by the same manager (i.e., same code space). This provides good least privilege protections.

The Owner field and Code-space field are used to provide a measure of modularity to the operating system. Secure systems require a design that uses the concept of least privilege, where data and code are designed in modular units controlling only limited

resources of the operating system. The Owner field is used to indicate the entity that owns the resource managed by the code module. The Code-space field indicates the code modules that are currently executing and/or the code modules that are authorized to access specific operating system resources (e.g., data structures and OS functions), in support of least privilege.

For example, the task manager code space is authorized to manage tasks of the system, but tasks may be created on behalf of different users or managers. Therefore the Owner field is used to further refine privileges so that the task manager will have hardware supported access control that ensures that the task manager manages tasks only for users who own those tasks (e.g., a user can not request the deletion of another user's task.)

The Control-bits field is used to further support least privilege by providing some typing and access control information to the system resources. This field may be extended with some of the attack prevention technologies of Suh [25]. The bit representation in Figure 3.1 is a current recommendation.

3.1.1 Owner field

In the implemented tagging scheme, the first field, the Owner field, helps separate system code/data and user code/data. It indicates the identity of the owner of the data or code. As shown in Figure 3.2, the values of the Owner field can be classified into six major classes: `SCORE internal`, `SCORE`, `Manager internal`, `Manager`, `Startup` and `User`. The `SCORE internal` class can be further divided into three groups: `SCORE internal init` group, `SCORE internal private` group and `SCORE internal` group. The `Manager internal` class is also broken into three groups: `Manager internal init`, `Manager internal private` and `Manager internal`. The possible values of each class are listed in Table 3.2.

By using the Owner field, the owner of data or code can be easily identified. For

example, the first field in the tag (object, ⟨Code-space⟩, ⟨Control-bits⟩) indicates that the data or code associated with this tag is object code or object data; where object is one of the SCORE modules. A tag (task manager, ⟨Code-space⟩, ⟨Control-bits⟩) shows that the data or code is owned by the task manager. In the remaining sections, ⟨SCORE⟩ in the tag means this field could be one of the possible values in the SCORE class, and the same holds for the Startup class, SCORE internal class, Manager class, Manager internal class and User class. Although RTEMS currently only supports a single user, it is expanded to a multiuser system as part of this dissertation (see Section 4.1). Therefore, multiple users are listed in the possible values of the User class. Having different users for the Owner field ensures that a user can only access its own data and code, but not other users' resources. The bit representation for the Owner field can be found in Section 3.3.1.

3.1.2 Code-space field

The second field of the tag is the Code-space field. This field shows which code space should manage the data or code. In addition to this, the Code-space field is also critical for information flow control and memory access control. The possible values of the Code-space field are the same as the Owner field (see Table 3.2). The Code-space can be ⟨User⟩, ⟨Manager⟩, ⟨Manager internal⟩, ⟨SCORE⟩, ⟨SCORE internal⟩ and ⟨Startup⟩. For example, the tag (User1, Manager1, ⟨Control-bits⟩) means the data is created in manager1's code for user1. Manager1 is used as a place holder for a specific manager name, such as task manager, partition manager, etc.

Code-space is used to show the class of the code or data and helps control function calls. Users in the system should only use some of the system functions, and not directly use SCORE, SCORE internal, and Manager internal functions. Therefore, firstly, Code-space is used to indicate which class the code belongs to, and then rules can be provided to control which classes of code can use which other classes of code. The bit representation for the Code-space field can be found in Section 3.3.1.

Table 3.1: Possible values of the memory type

Memory type	Read or read/write
Data memory (x00)	000 read-only 100 - read/write
Stack memory (x01)	001 read-only 101 - read/write
Code memory (executable) not entry point (x10)	010 read-only 110 - read/write
Code memory (executable) entry point (x11)	011 read-only 111 - read/write

3.1.3 Control-bits field

The Control-bits field is used for further control. It starts with a single copy bit which indicates whether a return value has been modified. The copy bit allows user code to have a copy of a trusted data value (i.e., a task ID) as long as it is not changed. The notation $\overline{cp}$ means the copy bit is not set while *cp* indicates the copy bit is set. The return value to a user will be tagged with the security class of the directive and will have the copy bit set. If the copy bit remains set, this means that user has not made any change to the value. If the copy bit is not set, the data is treated as modified data and will not be accepted when used as a parameter to a directive. This allows user programs to store copies of system IDs and handles while maintaining security of the values. For example, if user1 uses directive code from the task manager to get a tag identifier, the directive returns an identifier tagged (`user1, task manager,` *cp*). If the user modifies this returned ID, the tag will be changed to (`user1, user1,` $\overline{cp}$) to indicates this ID has been changed; and the OS will no longer trust the ID. This allows the system to trust IDs that come from users without having to keep an internal table of indirect identifiers or separate tables for each user, and thereby improve performance and simplify system code.

Three of the control bits are allocated for memory type. To further protect data in

memory, memory is divided into three classes: stack memory, code memory, and data memory. These three bits specify the memory type, such as readable, writable, read-only, executable, entry point, data memory, and stack memory.

- **Stack Memory.** Stack memory is readable and writable, and is treated using register expression rules for stores, instead of assignment rules.
- **Code Memory.** Code memory stores the executable and readable code. The "entry point" of a function has a special tag with it to indicate the correct starting address for executing a function.
- **Data Memory.** All other memory is data memory which is readable and writable, and it is treated using assignment rules.

The possible values of the memory type are shown in Table 3.1.

A world-readable bit is used to indicate that the tagged data can be read by all entities; used when the system (higher level) wants to give the user (lower level) permission to access some data, such as configuration data.

The other 3 bits are reserved for future use. The complete bit representation for the Control-bits field can be found in Section 3.3.2.

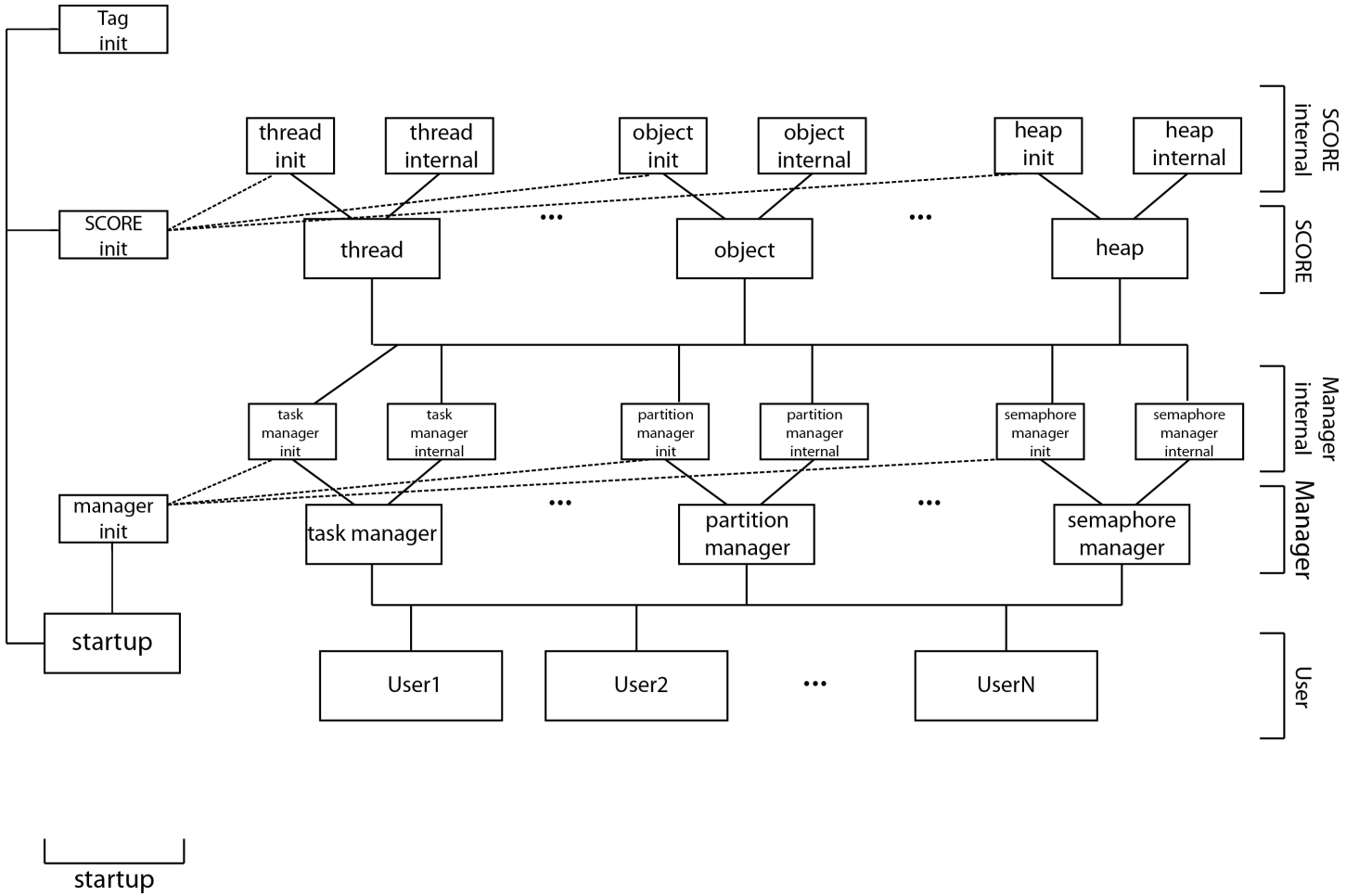

Figure 3.2: Classification of RTEMS code and users

Table 3.2: Possible values of the Owner field and Code-space field

Class	Group	Count	Possible values
SCORE internal	SCORE private internal	13	thread private internal, object private internal, chain private internal, initialization private internal, heap private internal, protected heap private internal, message private internal, error private internal, time private internal, ISR private internal, workspace private internal, watch dog private internal, user extension private internal
	SCORE internal	13	thread internal, object internal, chain internal, initialization internal, heap internal, protected heap internal, message internal, error internal, time internal, ISR internal, workspace internal, watch dog internal, user extension internal
	SCORE internal init	13	thread init, object init, chain init, initialization init, heap init, protected heap init, message init, error init, time init, ISR init, workspace init, watch dog init, user extension init
SCORE external		13	thread external, object external, chain external, initialization external, heap external, protected heap external, message external, error external, time external, ISR external, workspace external, watch dog external, user extension external

Continued on next page...

Class	Group	Count	Possible values
Manager internal	Manager private internal	20	initialization private internal, task private internal, interrupt private internal, clock private internal, timer private internal, semaphore private internal, message private internal, event private internal, signal private internal, partition private internal, region private internal, dual ported memory private internal, I/O private internal, fatal error private internal, rate monotonic private internal, user extensions private internal, multiprocessing private internal, barrier private internal, tag private internal, user private internal
	Manager internal	20	initialization internal, task internal, interrupt internal, clock internal, timer internal, semaphore internal, message internal, event internal, signal internal, partition internal, region internal, dual ported memory internal, I/O internal, fatal error internal, rate monotonic internal, user extensions internal, multiprocessing internal, barrier internal, tag internal, user internal
	Manager internal init	20	initialization init, task init, interrupt init, clock init, timer init, semaphore init, message init, event init, signal init, partition init, region init, dual ported memory init, I/O init, fatal error init, rate monotonic init, user extensions init, multiprocessing init, barrier init, tag init, user init
Manager external		20	initialization external, task external, interrupt external, clock external, timer external, semaphore external, message external, event external, signal external, partition external, region external, dual ported memory external, I/O external, fatal error external, rate monotonic external, user extensions external, multiprocessing external, barrier external, tag external, user external
Startup		4	startup, manager INIT, SCORE INIT, Tag INIT
User		N	user1, user2, ..., userN (Where N=118 for the implementation proposed in Chapter 4)

3.2 The Lattice for the UI Tagging Scheme

Since the Owner field and Code-space field are used to define the security class of the data, the ideas similar to those of the Data Mark Machine (DMM) [10] can be implemented to control the information flows within RTEMS. The solution is a bit different from DMM since a hierarchy is defined for confidentiality and integrity controls, and it is not for traditional multi-level security. However, accesses still can be controlled to prevent "lower-level" entities from unauthorized access to higher level entities.

The lattice model of information flow deals with channels of information flow and policies. For the lattice model of information flow, there exists a lattice with a finite set of security classes and a partial ordering between the classes. The nodes of the lattice represent the individual security classes. The notation $\leq$ denotes the partial ordering relation between two classes of information. For example, $A \leq B$ means class A is at a lower or equal level than class B. The notation $\oplus$ denotes the least upper bound (LUB) operation. In a lattice, there exists a unique class $C = A \oplus B$ such that:

$A \leq C$ and $B \leq C$, and

$A \leq D$ and $B \leq D$ implies $C \leq D$ for all D in the lattice.

Two parallel lattices are defined for information flow control within RTEMS, for each of the Owner and Code-space fields of the tags (Figure 3.3). These are extensions of the tag hierarchy of Figure 3.2. To simplify the diagram, assume that boxes higher on the figure have higher integrity and security classes than those below them, and that the $\leq$, $=$, $\oplus$ (least upper bound) and other operators support that hierarchy. In Figure 3.3, the Startup modules (STARTUP, MANAGER INIT, SCORE INIT, TAG INIT, and init functions of each RTEMS modules) connected by dashed lines are only used during the system initialization. They can be removed from the lattice and will not affect the information flow after the system starts running. This dual-use (security and integrity) is

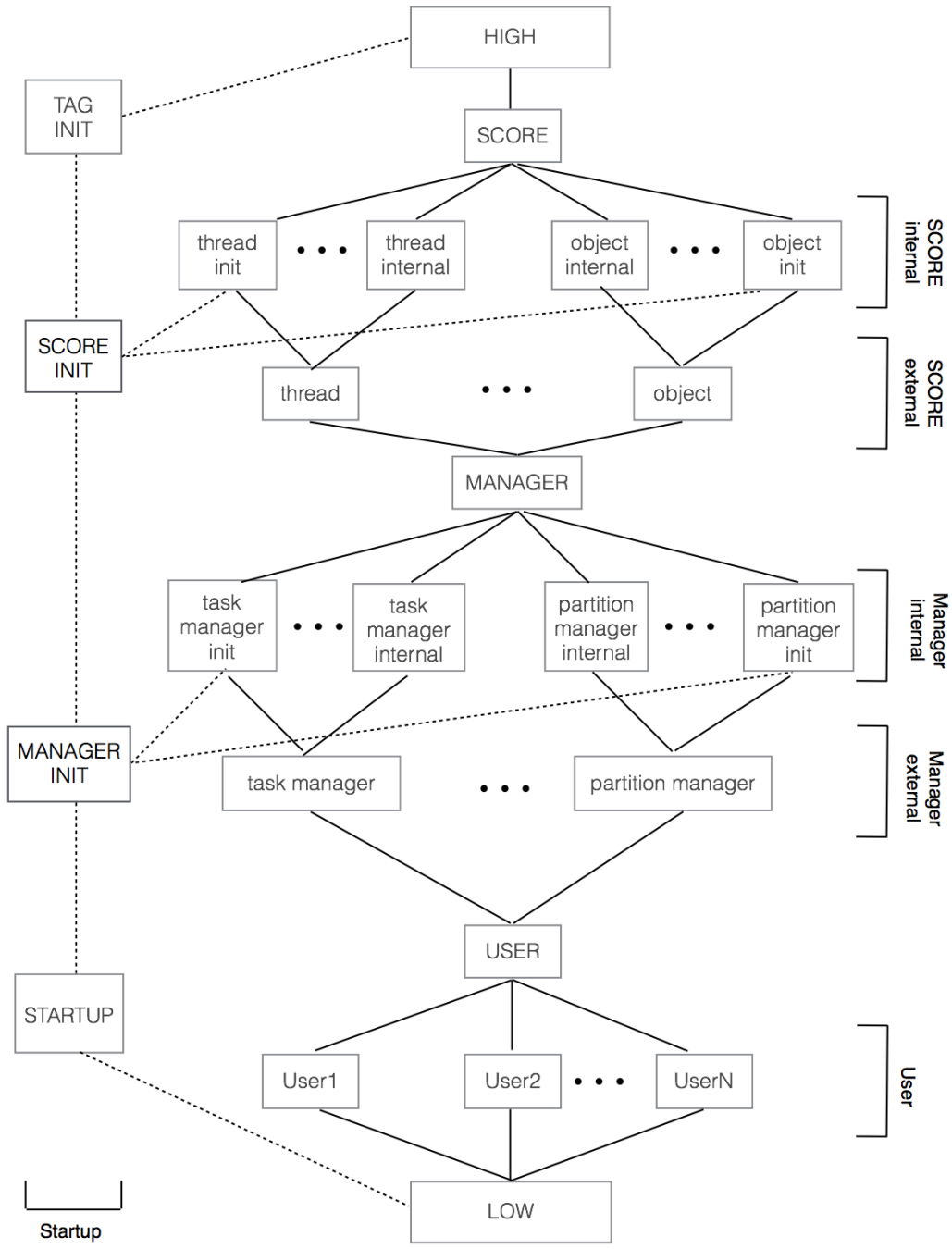

Figure 3.3: The lattice designed for Owner field and Code-space field

different from traditional security classes, but maps well to the standard use of privilege rings or supervisor/user execution modes in hardware.

In addition, the private internal and other internal functions are merged into a single box. An expanded version of these boxes is shown in Figure 3.4. The connections between managers and the internal functions of other managers will be restricted to the non-private internal functions. Connections between users and managers are described in Section 3.4.7.

The security class of an entity in the model, `a`, will be denoted as $\underline{a}$, and it can be written as $\underline{a}$ `= (Owner(a), Code-space(a))`. The Control-bits are ignored in this discussion since they are used separately from the lattice-based controls and formulas. `Owner(a)` refers to just the Owner field of `a`'s tag, `Code-space(a)` refers to the Code-space field of the tag of `a`.

The definition of the least upper bound, $\oplus$, of the security classes of two tags is:

- If $\underline{a}$ = `(Owner(a), Code-space(a))`, and $\underline{b}$ = `(Owner(b), Code-space(b))`, then $\underline{a} \oplus \underline{b}$ = `(Owner(a)` $\oplus$ `Owner(b), Code-space(a)` $\oplus$ `Code-space(b))`.

For example, if $\underline{x}$ = `(user1, semaphore manager)` and $\underline{y}$ = `(user2, semaphore manager)` then according to the lattice, $\underline{x} \oplus \underline{y}$ = `(user, semaphore manager)`.

The definition of the equality of the security classes of two tags is:

- If $\underline{a}$ = `(Owner(a), Code-space(a))`, and $\underline{b}$ = `(Owner(b), Code-space(b))`, then if `Owner(a)` = `Owner(b)` and `Code-space(a)` = `Code-space(b)`, then the security class of a and b are the same.

The similar rules for $\leq$, $<$, $\geq$ and $>$ operations can be defined similarly and are not shown here.

3.3 A Proposed Representation of Tags

Section 3.1 introduces a possible high-level representation of the tags which is used in the rest of this dissertation. A proposed low-level representation of the tags is described here to show how to map the system into a 32-bit tag (Figure 3.1).

3.3.1 Bit representation for Owner field and Code-space field

In a tag, each of the Owner and Code-space fields is represented with 12 bits, with the remaining 8 bits for the Control-bits. For Owner field and Code-space field, the higher 4 of the 12 bits are used to indicate if the tagged data or code is a system resource or not. If the higher 4 bits are 1111, this means that the tagged data or code is from the system, otherwise it is from users. For system code (tags that start with 1111), the lower eight bits are used to divide the code into smaller classes. Among the 8 lower bits, the higher 3 bits indicate the level of the class, and the lower 5 bits represent which component. Table 3.3 gives a general overview of the tag bit representation by only showing the levels of RTEMS system code.

The detailed component ID (lower 5 bits) can be found in Table 3.4. By doing this, the security class of a tag can be easily identified. For example, if the Owner field is `1111 100 00010` and the Code-space field is `1111 100 00010`, then the security class of the code is {`object, object`}. If two Owner fields are given, such as `0010 000 10010` (User) and `1111 001 01101` (user extensions manager), then the least upper bound of these two classes is `1111 001 01101` (user extensions manager). The least upper bound of `1111 100 00110` (heap) and `1111 100 01001` (message) is `1111 111 11111` (SCORE). For RTEMS, this configuration is sufficient for all the security classes that are needed.

Table 3.3: Overview of tag bit representation for Owner field and Code-space field

Bits (12 bits)			Representation
RTEMS Level			
RTEMS/ USER (4 bits)	RTEMS Level (3 bits)	Component ID (5 bits)	RTEMS Representation
1111 (RTEMS)	111	11111	Tag INIT, SCORE, HIGH
		00001-11110	SCORE private internal functions
		00000	
	110	11111	
		00001-11110	SCORE internal functions
		00000	
	101	11111	
		00001-11110	SCORE internal init functions
		00000	SCORE INIT
	100	11111	
		00001-11110	SCORE external functions
		00000	
	011	11111	MANAGER
		00001-11110	Manager private internal functions
		00000	
	010	11111	
		00001-11110	Manager internal functions
		00000	
	001	11111	
		00001-11110	Manager internal init functions
		00000	Manager INIT
	000	11111	
		00001-11110	Manager external functions
		00000	Startup
USER LEVEL			
1110 111 11111			USER
...			User1, User2, UserN
0000 000 00000			LOW

Table 3.4: Using tags to represent security classes

Class	**Bit representation**			
HIGH / SCORE / Tag INIT	1111 111 11111			
SCORE private internal	1111 111 00001	thread private internal	1111 111 00010	object private internal
	1111 111 00011	initialization private internal	1111 111 00100	chain private internal
	1111 111 00101	protected heap private internal	1111 111 00110	heap private internal
	1111 111 00111	workspace private internal	1111 111 01000	error private internal
	1111 111 01001	message private internal	1111 111 01010	ISR private internal
	1111 111 01011	watch dog private internal	1111 111 01100	time private internal
	1111 111 01101	user extension private internal		
SCORE internal	1111 110 00001	thread internal	1111 110 00010	object internal
	1111 110 00011	initialization internal	1111 110 00100	chain internal
	1111 110 00101	protected heap internal	1111 110 00110	heap internal
	1111 110 00111	workspace internal	1111 110 01000	error internal
	1111 110 01001	message internal	1111 110 01010	ISR internal
	1111 110 01011	watch dog internal	1111 110 01100	time internal
	1111 110 01101	user extension internal		
SCORE internal init	1111 101 00001	thread init	1111 101 00010	object init
	1111 101 00011	initialization init	1111 101 00100	chain init
	1111 101 00101	protected heap init	1111 101 00110	heap init
	1111 101 00111	workspace init	1111 101 01000	error init
	1111 101 01001	message init	1111 101 01010	ISR init
	1111 101 01011	watch dog init	1111 101 01100	time init
	1111 101 01101	user extension init		
SCORE INIT	1111 101 00000			

Continued on next page...

Class	Bit representation			
SCORE external functions	1111 100 00001	thread external	1111 100 00010	object external
	1111 100 00011	initialization external	1111 100 00100	chain external
	1111 100 00101	protected heap external	1111 100 00110	heap external
	1111 100 00111	workspace external	1111 100 01000	error external
	1111 100 01001	message external	1111 100 01010	ISR external
	1111 100 01011	watch dog external	1111 100 01100	time external
	1111 100 01101	user extension external		
MANAGER	1111 011 11111			
Manager private internal	1111 011 00001	initialization private internal	1111 011 00010	task private internal
	1111 011 00011	semaphore private internal	1111 011 00100	message private internal
	1111 011 00101	interrupt private internal	1111 011 00110	barrier private internal
	1111 011 00111	rate monotonic private internal	1111 011 01000	clock private internal
	1111 011 01001	dual ported memory private internal	1111 011 01010	timer private internal
	1111 011 01011	multiprocessing private internal	1111 011 01100	I/O private internal
	1111 011 01101	user extensions private internal	1111 011 01110	event private internal
	1111 011 01111	fatal error private internal	1111 011 10000	signal private internal
	1111 011 10001	partition private internal	1111 011 10010	region private internal
	1111 011 10011	tag private internal	1111 011 10100	user private internal
Manager internal	1111 010 00001	initialization internal	1111 010 00010	task internal
	1111 010 00011	semaphore internal	1111 010 00100	message internal
	1111 010 00101	interrupt internal	1111 010 00110	barrier internal
	1111 010 00111	rate monotonic internal	1111 010 01000	clock internal
	1111 010 01001	dual ported memory internal	1111 010 01010	timer internal
	1111 010 01011	multiprocessing internal	1111 010 01100	I/O internal
	1111 010 01101	user extensions internal	1111 010 01110	event internal
	1111 010 01111	fatal error internal	1111 010 10000	signal internal
	1111 010 10001	partition internal	1111 010 10010	region internal
	1111 010 10011	tag internal	1111 010 10100	user internal

Continued on next page...

Class	Bit representation			
Manager internal init	1111 001 00001	initialization init	1111 001 00010	task init
	1111 001 00011	semaphore init	1111 001 00100	message init
	1111 001 00101	interrupt init	1111 001 00110	barrier init
	1111 001 00111	rate monotonic init	1111 001 01000	clock init
	1111 001 01001	dual ported memory init	1111 001 01010	timer init
	1111 001 01011	multiprocessing init	1111 001 01100	I/O init
	1111 001 01101	user extensions init	1111 001 01110	event init
	1111 001 01111	fatal error init	1111 001 10000	signal init
	1111 001 10001	partition init	1111 001 10010	region init
	1111 001 10011	tag init	1111 001 10100	user init
Manager INIT	1111 001 00000			
Manager external functions	1111 000 00001	initialization external	1111 000 00010	task external
	1111 000 00011	semaphore external	1111 000 00100	message external
	1111 000 00101	interrupt external	1111 000 00110	barrier external
	1111 000 00111	rate monotonic external	1111 000 01000	clock external
	1111 000 01001	dual ported memory external	1111 000 01010	timer external
	1111 000 01011	multiprocessing external	1111 000 01100	I/O external
	1111 000 01101	user extensions external	1111 000 01110	event external
	1111 000 01111	fatal error external	1111 000 10000	signal external
	1111 000 10001	partition external	1111 000 10010	region external
	1111 000 10011	tag external	1111 000 10100	user external
Startup	1111 000 00000			
User	1110 111 11111			
Users	see Section 4.1 for details			
LOW	0000 000 00000			

3.3.2 Bit representation for Control-bits field

In addition to 12 bits for each of the User and Code-space fields, the 8-bit Control-bits field is used for further control. A copy bit (bit 7) is used to indicate whether a return value has been modified. The copy bit allows user code to have a copy of a trusted data value (i.e., a task ID) and be able to propagate the trust as long as the value is not changed.

Bit 6 to 4 are allocated for memory type, divided into three classes: stack memory, code memory, and data memory. All memory can be marked as read-only. Normally only stack memory and data memory can be writable. Within the 3 bits, bit 6 indicates it is read-only or readable and writable memory (1 indicates readable and writable and 0 indicates read-only). Code memory stores executable and readable code. The "entry point" of a function has a special tag to indicate the correct starting address for executing a function. Therefore for the lower 2 bits, 11 indicates an entry point to an executable function, 10 means the memory is executable but not an entry point, 01 indicates stack memory while 00 indicates data memory. For example, read-only data memory will be presented by 000.

A world-readable bit (bit 3) represents whether the tagged data can be read by all entities or not; it is used when the system (higher level) wants to give the user (lower level) permission to access some data, such as configuration data. For this bit, 1 indicates the world-readable bit is set while 0 means the bit is not set.

The remaining 3 bits (bit 2 to 0) are reserved for future use (e.g., DIFT-style protection [25]).

3.4 C Language Tagging Rules

At the start of the system, and as each subroutine is called, data and code tags are initialized with the correct security classification and memory types. This section discusses

how those classifications can be used, in the context of the C programming language, to define tagging rules needed to satisfy the security concerns based on the partial ordering and lattice concepts. Full details can be found in the corresponding MS Thesis [22].

3.4.1 Tagging rules for the basic values in C

This section introduces the basic concepts of using tags at the C language level.

- During execution of a program, the tag of the current running thread is the same as the security class of the program counter, denoted as `PC`.
- The tag of the variable `a` is the tag of the memory location of `a` and its security class is denoted `a`.
- The tag of a literal, or constant, `n`, is the same as the tag of the PC. This is because the use of the literal is controlled by the current thread.
- The security class of the tag of an array item, `a[i]` is the least upper bound of the security class of the index to the array and the security class of the memory location referenced by the array: `a[i]` = `i` $\oplus$ `[[a+i]]` where `[[a+i]]` denotes the memory address referenced by `a[i]`. In cases where the copy bit of the memory location is set, the tag of that memory location is used instead of the LUB.
- The tag of a value referenced by a pointer, `*p` or structure `p->fld` or `p.fld` is the tag of the memory location referenced. For example, `*p` = `[[p]]` where `[[p]]` denotes the memory address referenced by the pointer `p`.
- All code will be tagged as read-only, executable memory, and all entry points to functions will be tagged as function entry points (to avoid problems with the misuse of function pointers).
- All data memory will be tagged as read-write data memory.

3.4.2 Rules for the arithmetic and logic operations

For the arithmetic and logic operations, the first thing that needs to be checked is that all the variables can be accessed by the current running thread. Take mathematical expression `a + b` as an instance. To calculate the value of the expression, the value of `a` and `b` must be read. Therefore the security levels of a and b have to be lower than the security level of PC's tag.

- To be able to access the values stored in a and b, the conditions $\underline{\texttt{a}} \leq \underline{\texttt{PC}}$ and $\underline{\texttt{b}} \leq \underline{\texttt{PC}}$ have to be satisfied.

The rules for arithmetic and logic operations are used to specify the tag of the resulting value of the operation. For example, for the expression `a + b`, `a` and `b` are two operands, and the security class of the result depends on the copy bits and the security classes of the two operands. A set copy bit indicates that the tag of a value consists of the tag of the directive that created the value. Any change to the value ignores the directive's tag. Therefore the copy bit in the tag of `a` is `cp` and the copy bit of `b` is $\overline{cp}$, then only the security class of `b` needs to be considered. If the copy bit of both operands are $\overline{cp}$, then the security classes of both `a` and `b` need to be used and the security class of the result will be $\underline{\texttt{a}} \oplus \underline{\texttt{b}}$. The notation $\oplus$ denotes the least upper bound (LUB) operation. If `a` or `b` are constants, they have the same tag as the PC. In the tag of the result of arithmetic and logic operations, the copy bit is always reset, since the value has been modified. The particular rules for arithmetic and logic operations are shown in Table 3.5.

3.4.3 Rules for the comparison

In the C programming language, a comparison expression gets an integer value as the result. For the comparison `a > b`, to make sure that the proper tag is maintained, the result of the comparison should get the appropriate security class. Similar to the rules

Table 3.5: Rules for arithmetic, logic and comparison operations

Copy bit of a	Copy bit of b	Security class of the result	Copy bit of the result
cp	cp	$\underline{PC}$	$\overline{cp}$
cp	$\overline{cp}$	$\underline{b}$	$\overline{cp}$
$\overline{cp}$	cp	$\underline{a}$	$\overline{cp}$
$\overline{cp}$	$\overline{cp}$	$\underline{a} \oplus \underline{b}$	$\overline{cp}$

for arithmetic and logic operations, the security class of the result depends on the copy bits and the security classes of `a` and `b`. The copy bit of the result of the comparison is always reset since it is a new value. The explicit rules for comparing `a` and `b` are the same as the rules for arithmetic and logic operations shown in Table 3.5.

3.4.4 Information flow rules for the assignment statements

For assignment statements, such as `y = x;`, whether the information is allowed to flow from `x` to `y` needs to be checked. In the lattice model of information flow, the assignment statement of `y = x` requires the relation $\underline{\texttt{x}} \leq \underline{\texttt{y}}$ to ensure confidentiality. In addition, the security class of the current thread is also important to ensure integrity. To avoid unauthorized copying or modification of data, $\underline{\texttt{y}} \leq \underline{\texttt{PC}}$ is required for integrity. The value `x` can be a variable, a constant, a complex expression, etc. The rules for the assignment statement (`y = x;`) are:

- If the copy bit in the tag of `y` is $\overline{cp}$:
  - When the copy bit in the tag of `x` is cp and if $\underline{y} \leq \underline{PC}$ and $Owner(x) = Owner(y)$, the assignment statement is allowed and the tag of `x` is copied to `y`'s tag, otherwise it is not allowed.
  - When the copy bit in the tag of `x` is $\overline{cp}$ and $\underline{x} \leq \underline{PC}$ and $\underline{y} \leq \underline{PC}$, the information flow is allowed to flow from `x` to `y` and the tag of `y` is unchanged,

otherwise it is not allowed.

- If the copy bit in the tag of `y` is *cp*:
  - When the copy bit in the tag of `x` is *cp* and if $Owner(PC) > Owner(y)$ and $Owner(x) = Owner(y)$, the assignment statement is allowed and the tag of `x` is copied to `y`'s tag, otherwise it is not allowed.
  - When the copy bit in the tag of `x` is $\overline{cp}$ and $\underline{x} \leq \underline{PC}$ and $Owner(PC) > Owner(y)$, the information flow is allowed to flow from `x` to `y` and the Code-space field of the tag of `y` is reset, otherwise it is not allowed.

If an assignment is not allowed, the hardware will generate a security exception and execute code in a specified exception handler.

When performing an assignment statement (`y = x;`), `x` might be the result of an arithmetic operation or a result of several arithmetic operations. Therefore, the rules for arithmetic and logic operations (Section. 3.4.2) are used to define the tag of the expression that `x` represents. The information flow check can be done by checking the relation between the tags of `y` and the result of the operation, `x`.

Note that the check of $\underline{\texttt{y}} \leq \underline{\texttt{PC}}$ seems to violate the traditional security "no write down" model of Bell-LaPadula [3], however, it satisfies the integrity controls of Biba [4].

3.4.5 Information flow rules for the if statements

For the information flow rules for the `if` statements, such as `if(expression) commands;` and `if(expression) commands1; else commands2;`, whether the expression can be read by code executing this `if` statement needs to be checked. To ensure that the expression can be read by code the relation between $\underline{\texttt{PC}}$ and $\underline{\texttt{expression}}$ needs to be checked before running the `if` statement. Therefore, the rule for `if` statement is:

- If $\underline{\texttt{expression}} \leq \underline{\texttt{PC}}$, then the statement is allowed.

As with assignment, the `expression` can be a variable, a constant, a result of comparisons etc., and the tag of the `expression` is calculated using the rules for comparison operations (Section. 3.4.3). The commands in the `then` and `else` statements are controlled by their appropriate rules.

3.4.6 Information flow rules for the while statements

The `while` statement, such as `while(expression) commands;`, requires a check on whether the expression can be read by code executing the `while` statement, similar to the rule for the `if` statement. The rule for `while` statement is:

- If $\underline{\texttt{expression}} \leq \underline{\texttt{PC}}$, then the statement is allowed.

Again, the `expression` can be a variable, a constant, or a result of comparisons; and its tag is set using the appropriate rules. The commands in the `while` statement are controlled under other specific rules. Different from the `if` statement, the `while` statement body could be a loop of commands if the `expression` is true. Therefore, the information flow rules for the `while` statement will be used at every loop iteration.

3.4.7 Information flow rules for the function calls

The idea of functional access control is implemented in the UI tagging scheme. When executing a function call, the tag manager will first check the access control rules for function calls to see if the code has permission to call the function. This is called "function execution control".

By performing access control for the function calls, the control of who can execute which function can be ensured.

Figure 3.4 shows a detailed view of a single kernel module. In the module, all of the APIs are classified into four groups: external APIs, init APIs, internal APIs and

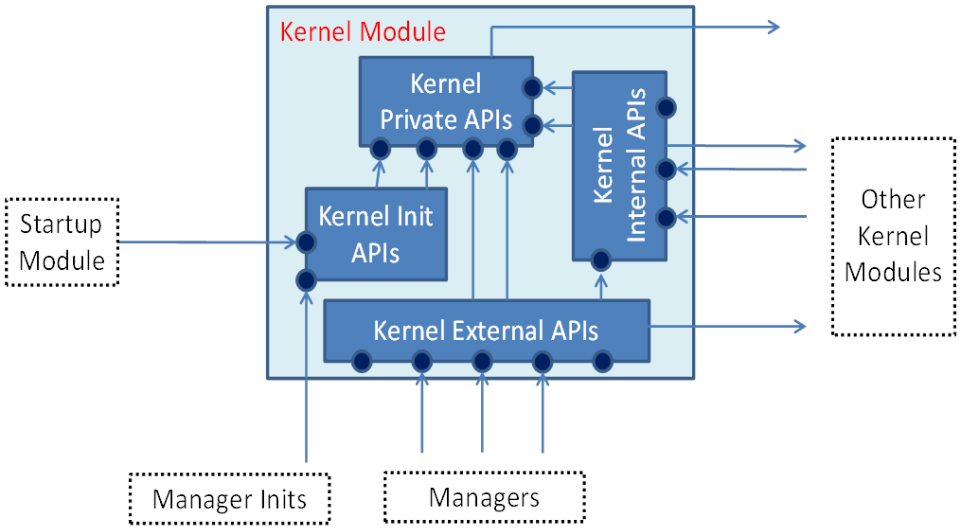

Figure 3.4: A detailed kernel module

private APIs. This classification can be applied to different modules on either Manager level or SCORE level. Among these APIs, only the external APIs can be used by lower Manager level APIs, and the other three groups of kernel APIs are not allowed to be used by lower levels. The init APIs are used to initialize the kernel module and it can be used by the startup module and manager level init APIs. The difference between the private APIs and internal APIs is that the internal APIs are allowed to be used by other kernel modules, while the private APIs can be used only by the APIs of the same kernel module. With the help of tags, these access control rules can be enforced by using function execution control rules.

In addition to the rules for different groups of kernel APIs, there exist rules to deal with access of entities from different levels. For example, user1 code, which is created in `user1`, is only allowed to call its own functions and manager external APIs (directives) but not SCORE APIs and other APIs. Both Manager APIs, Manager internal APIs and Manager private internal APIs are allowed to use SCORE external APIs. All the system functions (Startup, SCORE, SCORE internal, Manager, Manager internal) are not allowed to call user functions. Note that Manager internal functions are divided into

Manager internal functions and Manager private internal functions.

When performing a function call, function execution control rules are checked first. If the function is allowed to be called by the caller, then whether the parameters passed to the function are allowed to be accessed by the code will be checked. Therefore, the security classes of the parameters and the security class of the caller, which is the security class of the current PC, are used. The information flow rule for function calls is very similar to the rule for testing the expression in the while statement. The information flow rule for a function call like `foo(parameter1,parameter2);` is:

- If the function execution control rules allow the call and if $\underline{\texttt{parameter1}} \leq \underline{\texttt{PC}}$ and $\underline{\texttt{parameter2}} \leq \underline{\texttt{PC}}$, then the function call is allowed to execute.

The tags of the parameter variables in the function will follow the expression and assignment rules.

3.4.8 Other C constructs

The tagging rules for other C language constructs are not defined, such as the `for` loop or `switch` statement since they can be mapped to other constructs which we have defined. In addition, rules for `break` and `continue` are not defined, since their only function is to change location of execution within the same code space and therefore do not have information flow concerns for the tagging model.

3.5 Implementation of the UI Tagging Scheme at the Assembly Language Level

Section 3.4 introduced the implementation of the UI tagging scheme at the C programming language level. However, the higher-level abstraction of the C language ignores important implementation details that will affect the security of the tagging scheme. As

a result this section explains the implementation of the UI tagging scheme at the assembly language level; more complete details can be found in corresponding MS Thesis [22]. From the study of the SPARC instruction set, instructions are divided into five major groups:

(1) Branch instructions

(2) Call related instructions

(3) Arithmetic, logic and shifting instructions

(4) Load and store instructions

(5) Save and restore instructions

Tagging rules have been specified for each of these groups.

3.5.1 Rules for branch instructions

Branch instructions are either unconditional branches or conditional branches. Unconditional branch instructions are `BA` (branch always), which means always execute the branch, and `BN` (branch never), which means never execute the branch (effectively a `NOP`). Conditional branches are based on the condition codes, which are updated by compare instructions, to decide whether or not take the branch. Therefore, whether or not the current running thread is permitted to access the condition code has to be checked.

For the execution of the target code, a check is needed to make sure the current thread is allowed to execute (jump to) the target code. Consequently, for branch instructions such as `Bicc address`, the tags of the address and PC must be checked to ensure the jump is appropriate. Since this is a jump and a jump should be made in the same Code-space, the Code-space should not change. Therefore, the `Code-space(address)` (`Code-space(address)` refers to the Code-space field of the address's tag) and the `Code-space(PC)` must be checked to make sure the current thread is allowed to execute the target code. In addition, the target address must be an executable code memory, but not data memory or stack. The rule for branch instructions is:

- if [condition code] $\leq$ PC, and Code-space(address) = Code-space(PC), and the target address is an executable code memory then the branch is allowed; otherwise, throw an exception.

3.5.2 Rules for call related instructions

The four call related instructions are CALL, JMPL, RETT and Ticc. When making a function call (CALL address), the tag system must check whether or not the function is allowed to be called by the current program, using the function execution control rules. The function execution control rules specify that if the current program is not allowed to call the function, then the call will not be executed. If it is allowed under the rules and the target code is the entry point of an executable function, then the call is allowed. The call will save the value of PC into %o7, PC into %o7, set the copy bit of %o7, update the PC and tag of the PC, and then start execution of the function. The tag of the updated PC will follow the function execution control rules. The rule for CALL instruction is:

- If the function call is allowed under the function execution control rule, and if the target code is tagged as entry point to an executable function, then the CALL instruction is allowed.

The JMPL instruction is used to save the current PC in a specified register and then jump to any specified address. There are two usages of JMPL instruction. The first is JMPL address, %g0 which copies the current PC to %g0 before making the jump. However, because %g0 always has value 0 in it and cannot be overwritten, the write to it has no effect. As a result this is a true jump with no expected return. As with branch instructions, the current running programs must have the permissions to jump and execute the target code. For that reason the security class of PC and the target address must be compared. The rule for the first usage of the JMPL instruction is:

- If `Code-space(address) = Code-space(PC)`, and the target address is an executable code memory then the `JMPL` is allowed.

The second usage, `JMPL address, %o7` will write the current PC to `%o7`. Since SPARC uses `JMPL %o7+8, %g0` to jump back (i.e., the return statement) according to the return address stored in `%o7`, a check is needed to ensure the return address has not been modified. However, `JMPL` has many other usages, so a restrictive rule can not be applied to `JMPL`. Consequently a new instruction is required to be implemented, `RET`, which is used for return from subroutines. The format of `RET` instruction pseudo-op could be `RET %o7+8, %g0` (SPARC specifies a `RET` instruction that is just a shorthand notation for `JMPL %o7+8, %g0`, this will be replaced with a separate instruction). For return from a subroutine, if the copy bit of the tag of `%o7` is not set, then the return is not allowed. If the copy bit is set, which ensures that the return address was not modified, then the checks of the PC and the return address stored in `%o7` are required. The rule for `RET` is:

- when the copy bit of `%o7` is not set, the `RET` instruction is not allowed.
- If the copy bit of `%o7` is set, and `Code-space([%o7+8]) = Code-space(%o7)` and the target address (`[%o7+8]`) is executable code, then the `RET` is allowed, with the new `PC = %o7+8`, and the new security class of the tag of PC is security class of `%o7`.

`RETT` is used to return from a trap handler. `Ticc` is used to generate a trap to the trap handler based on an integer condition code. For this initial tagging system, same rules are applied to `CALL` and `RET` for these instructions; understanding that they will have to be modeled and possibly changed for a fully secure system.

3.5.3 Rules for arithmetic, logic and shifting instructions

For arithmetic instructions, logic instructions and shifting instructions, tags are propagated during execution. For example, the instruction `ADD %g2, %g1, %g3` adds the value

Table 3.6: Rules for arithmetic, logic and comparison instructions

Copy bit of %g2	Copy bit of %g1	Security class of the %g3	Copy bit of the %g3
cp	cp	$\underline{\text{PC}}$	$\overline{cp}$
cp	$\overline{cp}$	$\underline{\%\text{g1}}$	$\overline{cp}$
$\overline{cp}$	cp	$\underline{\%\text{g2}}$	$\overline{cp}$
$\overline{cp}$	$\overline{cp}$	$\underline{\%\text{g2}} \oplus \underline{\%\text{g1}}$	$\overline{cp}$

in `%g2` to the value in `%g1`, then stores the result in `%g3`. To propagate tags, the copy bit in the tags of the value in `%g1` and `%g2` need to be checked, and the result value stored in `%g3` has a new tag associated with it. Whether or not the result can be stored in `%g3` is not checked, instead the registers are used as temporary storage. Taking `ADD %g2, %g1, %g3` as an example, the new tag of `%g3` is calculated based on the tags of `%g2` and `%g1`, with the copy bit unset. The particular rules for arithmetic, logic, and shifting instructions are shown in Table 3.6. Instructions that modify the condition code, such as `ANDcc, ORcc, SUBcc`, and `UMULcc`, set the tag of the condition code to the tag of the result. For example, after executing the instruction `ADDcc %g2, %g1, %g3`, the condition code's tag will be the tag of `%g3`. If `%g1` is an immediate value, then `%g1`'s tag is PC's tag with the copy bit not set.

3.5.4 Rules for load and store instructions

Load instructions are used to load a value from a memory space and store it to a register. For example, the instruction `LD [%fp], %o1` loads the content of the memory address `[%fp]` to the `%o1` register. A check of whether the current program can read and use the value is needed. What is more, whether the current running thread can access the value of `%fp` needs to be checked. Therefore the security classes of PC and the tag of the value and the tag of the `%fp` need to be checked. The load instruction is allowed when `[%fp]` $\leq$ `PC`, where `[%fp]` denotes the security class of the tag of the value stored in `[%fp]`. The

rules for load instructions enforce these checks of expressions and parameters. The rules for load instructions (`LD [%fp], %o1`) are:

- If the world readable bit of the memory address's tag is set, and the memory type is data memory or stack memory, then the `LD` is allowed. The tag of the memory location will be copied to the register's tag.
- If the copy bit in the tag of `[%fp]` is *cp*, and `%fp` $\leq$ `PC`, and `Owner([%fp])` $\leq$ `Owner(PC)`, and `[%fp]` is readable memory, then the `LD` is allowed and the tag of the data will be copied to the register's tag.
- If the copy bit in the tag of `[%fp]` is $\overline{cp}$, and `if [%fp]` $\leq$ `PC` and `%fp` $\leq$ `PC`, and `[%fp]` is readable memory, then the `LD` instruction is allowed and the tag of the data will be copied to the register's tag.

Store instructions are used to store the value from a register to memory. The rules for store instructions check whether or not the value of the register is allowed to be stored in that location. For example, for the instruction `ST %g1, [%fp]`, which stores the content of `%g1` to memory space `[%fp]`, a check is needed to make sure the current running thread can access the value of `%fp` and write to the memory space. Rules for store instructions (`ST %g1, [%fp]`) are:

- When the copy bit in the tag of `[%fp]` is $\overline{cp}$ and the copy bit in the tag of `%g1` is $\overline{cp}$, if `[%fp]` $\leq$ `PC` and `%fp` $\leq$ `PC` and `[%fp]` is writable data memory, the store instruction is allowed. The tag of the `[%fp]` is unchanged.
- When the copy bit in the tag of `[%fp]` is $\overline{cp}$ and the copy bit in the tag of `%g1` is *cp*, if `[%fp]` $\leq$ `PC` and `%fp` $\leq$ `PC` and `Owner(%g1) = Owner([%fp])` and `[%fp]` is writable data memory, the store instruction is allowed. The tag of the `[%fp]` is copied from the tag of `%g1`.

- When the copy bit in the tag of `[%fp]` is *cp* and the copy bit in the tag of `%g1` is $\overline{cp}$, if `Owner([%fp])` $\leq$ `Owner(PC)` and `%fp` $\leq$ `PC` and `[%fp]` is writable data memory, the store instruction is allowed. The Code-space of the tag of the `[%fp]` will be reset to the Owner field of the tag and the copy bit will be unset.

- When the copy bit in the tag of `[%fp]` is *cp* and the copy bit in the tag of `%g1` is *cp*, if `Owner([%fp])` $\leq$ `Owner(PC)` and `%fp` $\leq$ `PC` and `Owner(%g1) = Owner([%fp])` and `[%fp]` is writable data memory, the store instruction is allowed. The tag of the `[%fp]` is changed to the tag of `%g1`. If `[%fp]` is writable stack memory, then write is allowed and tag of `[%fp]` is tag of `%g1`.

3.5.5 Rules for SAVE and RESTORE instructions

The `SAVE` and `RESTORE` instructions are used to slide the register window between the caller and callee windows. It is possible but rare, to use these instructions without a corresponding subroutine call. The execution of the `SAVE` instruction causes the system to subtract one from the CWP. Therefore, the instruction automatically allocates a new window of registers and a new stack frame in main memory. The `out` registers of the caller become the `in` registers of the callee.

In addition to sliding the register window, the `SAVE` instruction acts like an `ADD` instruction, which adds a number to `%sp`. The number indicates the size of the stack for the new function. The caller's stack pointer `%sp` automatically becomes the frame pointer `%fp` of the callee. The instruction `RESTORE` is used to restore the caller's window. The instruction adds one to CWP, therefore the callee's `in` registers become the caller's `out` registers.

The user can use `SAVE` and `RESTORE` instructions, which brings about issues concerning the security of the windows. The user may use the `RESTORE` instruction to change the register window to get data used in the caller's subroutine. Therefore, every register

window will be associated with a tag to indicate who created the register window. On every `SAVE` instruction, the tag of the PC is copied to the tag of the new window. On every `RESTORE` instruction, a check on the tag of the caller's window is required. If the tag indicates that the creator of the window is not the one who wants to restore (pop) the register window, then the restore instruction is not allowed to execute. During traps and exceptions, SCORE code is given the ability to manage the register window. As a result SCORE code has permission to restore any register window without any check of the window's tag. If the tag of the register window is identical to the tag of the PC, then the restore is allowed.

The rules for `SAVE` and `RESTORE` instructions are:

- On every `SAVE` instruction, the tag of the PC will be copied to the tag of the register window.
- For the `RESTORE` instruction, if the tag of the register window and the PC's tag are the identical or if `Code-space(PC)` $>$ `MANAGER`, then the `RESTORE` is allowed. Otherwise the `RESTORE` is not allowed.

3.6 Concerns about the Memory and Stack in the SPARC Architecture

Assembly language implementation of the C programming language requires memory to store local variables, functions, parameters, and linkage information in support of procedure calls. This memory region for each procedure is called a frame. Normally, these frames are allocated on a runtime stack, and pushed and popped with procedure calls and returns. For the SPARC architecture, the frame contains space for registers, parameters, PC, and so on. To access the frame, the frame pointer, `%fp`, is normally used.

Stack memory is used to store frames, and the frames on the same stack in a ZKOS can belong to directive code or users. Therefore, frames can not be treated the same as regular memory. For example, after return from a directive, the memory that has been used for that directive frame will be tagged with a tag containing Manager Code-space. If the user tries to call a new user function, the same memory can not be used for the user function frame without violating the assignment tagging rules unless the stack is treated differently from regular memory. Stack memory is treated as if the copy bit is set for all write access to the stack, and must therefore be limited to specific pages of memory. The specification of the memory pages for stacks can be implemented as secure API's of the tag manager and as functionality in the tag coprocessor.

3.7 Summary

As discussed in the previous chapter, RTEMS has many security concerns that need to be addressed. Therefore the goal of this part of the UITags project is to use a hardware based security tagging scheme to secure RTEMS. Although the new tagging scheme is focused on the security concerns of RTEMS, the concerns are common to many types of operating systems and thus the tagging scheme should be usable for other operating systems as well. This chapter introduces the UI tagging scheme by first illustrating the tag format. Then the tagging rules for the C language have been presented, including tagging rules for assignment statement, if statements, while statements, and arithmetic operations.

A study of SPARC instructions revealed that the information flow control rules designed for the C programming language needed to be refined to fit into the SPARC architecture. Some of the rules for SPARC instructions are similar to the rules for the C language, but some of the security features require refinements of the tagging rules for instructions at the assembly level. Therefore, this chapter presents the tagging rules for different groups of assembly instructions. These rules have been implemented in the

simulator, hence when executing an instruction, the specific tagging rules will be used to check the tags of the operands and propagate the tags.

Chapter 4: Multiuser system and test cases

One of the goals of this project is to expand RTEMS to support multiple users. To enable this, a new "user" construct has been added to RTEMS, similar to the task construct. All references to users in this chapter refer to this construct and not any real individuals. To manage the users, a "superuser" has been defined to have the authority to create, delete and control non-privileged users. Non-privileged users are isolated from each other and do not have authority to control other users. All control is managed through a new user manager.

With no hardware implementation available, modifications of the SPARC instruction simulator (SIS) was required to support tag checking and tag propagation. In addition, some tagging functions must be added to RTEMS to provide software support for tag handling. Hooks must be added in SIS, and additional modified instruction interpretations are required. For example, the normal `LD` instruction loads data from memory to registers. However, the modified `LD` instruction checks tags first to make sure the current running thread can access the data, and then loads data from memory if the access is allowed according to the tagging rules. New instructions simulations were also needed for coprocessor control and communication and for the new `RET` instruction.

Test cases have been generated to ensure that the tagging rules and the modified interpretations of instructions are correct. In total, 76 test cases were generated (consisting of more than 2000 smaller tests) which tested the modified instructions.

The remainder of this chapter is organized as follows: Section 4.1 introduces the design of a multiuser system and the tagging issues related to changing RTEMS. A test case for tagging multiple users is illustrated in this section. The test cases developed to test the simulator and the tagging rules are explained in Section 4.2. Section 4.3 summarizes the whole chapter.

4.1 Implementation of the Multiple User System

Before our work, RTEMS implemented a single-user multi-threaded model of execution. To expand it to a multiuser system, the concepts of "non-privileged user" and "superuser" have to be supported. A superuser is a user who has authority to create, delete, and control non-privileged users. Non-privileged users are isolated from each other and can only control themselves, but have no abilities to create, delete or control other users.

The new RTEMS user model allows for 117 non-privileged user IDs and one superuser. For an embedded system, 118 users each with 32 possible subtasks should be sufficient. Each user can have up to 32 separated task IDs. To have more control over users and their tasks, a set of task IDs is provided to each user. This allows additional secure isolation between the tasks within a single user. To implement this, the user levels' Owner field of the tag is divided into two parts – users and tasks. The higher 7 bits among the 12 bits are used to represent separate user IDs and the lower 5 bits are used to indicate task IDs for those users. For example, the 12 bits for the superuser are `1110 110 XXXXX`, where the `1110 110` indicates superuser ID and the `XXXXX` represents task IDs. Since only 7 bits are used to represent different users, this limits the number of users in the system to be user1 to user117 (`0000 001 XXXXX` to `1110 101 XXXXX`), recalling that `1111 XXX XXXXX` is reserved for RTEMS code. Including the superuser, there are 118 possible users in the system. This does not influence the levels above the general user (USER) level, because separate controls are made for users and their tasks. The new bit representation for Owner field of tags are shown in Table 4.1.

4.1.1 User manager

To change RTEMS to a multiuser system, system code needs to be modified to support multiple users. This includes adding a user manager to RTEMS to handle the superuser and its abilities to create, delete and control non-privileged users and corresponding

Table 4.1: New bit representation for Owner field of USER and LOW levels

USER	1110 111 11111
Superuser	1110 110 XXXXX
user117	1110 101 XXXXX
...	...
user1	0000 001 XXXXX
LOW	0000 000 00000

changes to the tag manager. The code displayed in Figure 4.1 creates a user in RTEMS. In addition, the simulator needed modification of the tag engine code to support the user hierarchy rules.

The `rtems_user_create` function first checks that the calling user is the superuser, because only the superuser is authorized to create, control and delete the non-privileged users. If this is the case, `rtems_user_allocate` gets the next available user id, starting at 1, and updates its internal data structures. After getting the user number, the `get_user_tag` is called to get a specific tag associated with the `task_id`s and `task_name`s for that user. For example, if user number is 1, then a tag `(USER1, USER1, true | READWRITE | DATA_MEMORY | WORLD_NOT_READABLE)` is returned. If user number is 2, the function `get_user_tag` will return a tag, `(USER2, USER2, true | READWRITE | DATA_MEMORY | WORLD_NOT_READABLE)`. Further in the function, `_rtems_tag_word` is used to tag the `Task_id`s and `Task_name`s. (Although tags are associated with pointers, they are actually being attached to the memory location referenced by the pointer.)

In multiuser RTEMS, a user program is assigned a primary application with task id 0. This is the first application to execute for the user. The primary application for a user first declares an array of `task_id` and another array of `task_name`. When creating a task, the user application passes two pointers to the `rtems_task_create` directive which point to the `task_id` and `task_name`. Then the directive creates the task for the user application and returns a system-wide unique `task_id` back. By using the system `task_id`, the user application is able to start, suspend, resume, or delete the task.

```
1  rtems_status_code rtems_user_create(rtems_id *user_id_ptr, rtems_name *
       user_name_ptr, int user_memsize)
2  {
3      int i, user_num = 0;
4      char c;
5      tag_t user_tag;
6      if (!rtems_superuser_check())
7         {return RTEMS_INVALID_USER;}
8      user_num = rtems_user_allocate();
9      user_tag = get_user_tag(user_num);
10     c = (char)(user_num + 48);
11     for (i=user_memsize; i>0; i--)
12         {*user_id_ptr = user_num;
13          *user_name_ptr = rtems_build_name('U','S','R',c);
14          _rtems_tag_word((addr_t)user_id_ptr, user_tag);
15          _rtems_tag_word((addr_t)user_name_ptr, user_tag);
16          user_id_ptr++;
17          user_name_ptr++;
18         }
19     return RTEMS_SUCCESSFUL;
20 }
```

Figure 4.1: RTEMS user create function

4.1.2 Test for multiuser tagging system

Figure 4.2 shows an `Init` function that uses the `rtems_user_create` function twice to create two non-privileged users. In this example the PC's tag is manually set to the SUPERUSER. Normally this will happen in the kernel prior to calling `Init`. The result of calling the `rtems_user_create` function is shown in Figure 4.3. The `Task_id[1]` to `Task_id[4]` are tagged with user1's tag, and `Task_id[5]` to `Task_id[7]` are tagged with user2's tag.

As shown in Figure 4.4, the user application sends `Task_id[1]` to `Task_id[4]` to the `rtems_task_create` directive to create four new tasks separately. The directive then creates four tasks and stores the RTEMS task ids into the `Task_id[1]` to `Task_id[4]` respectively. The copy bit of the tags associated with `Task_id[1]` to `Task_id[4]` will be set. Similarly, another three tasks will be created by user2 for `Task_id[5]` to `Task_id[7]`.

```
1  rtems_task Init(rtems_task_argument argument)
2  { ...
3    tag_t pc_tag = make_tag(SUPERUSER, SUPERUSER, false, READWRITE |
       DATA_MEMORY | WORLD_NOT_READABLE);
4    _rtems_tag_pc(pc_tag);
5
6    /*CREATE user1 and user2*/
7    status = rtems_user_create(id_ptr, name_ptr, NODE1_NUM);
8    if (status)
9    {printf("\n user1 created!!\n id_ptr tag: %s\n", tag_to_string(str,
       _rtems_get_memory_tag((addr_t)id_ptr)));
10     for(i = SUPERUSER_TASK_NUM; i <= NODE1_NUM; i++ )
11     {printf("Task_id[ %d ] tag: %s\n", i, tag_to_string(str1,
         _rtems_get_memory_tag(&Task_id[i]))); }
12   }
13   else {printf("\n Failed to create user1!!\n");}
14
15   id_ptr2 = id_ptr + NODE1_NUM;
16   name_ptr2 = name_ptr + NODE1_NUM;
17
18   status = rtems_user_create(id_ptr2, name_ptr2, NODE2_NUM);
19   if (status)
20   {printf("\nuser2 created!!\n id_ptr2 tag: %s\n", tag_to_string(str,
       _rtems_get_memory_tag((addr_t)id_ptr2)));
21     for(i = SUPERUSER_TASK_NUM + NODE1_NUM; i <= NODE1_NUM + NODE2_NUM;
         i++ )
22     {printf("Task_id[ %d ] tag: %s\n", i, tag_to_string(str1,
         _rtems_get_memory_tag(&Task_id[i]))); }
23   }
24   else {printf("\n Failed to create user2!!\n");}
25
26   status = rtems_task_ident( RTEMS_SELF, RTEMS_SEARCH_ALL_NODES, &tid )
       ;
27   init_index = task_number( tid );
28   printf("\nThe current running task is %d\n", init_index);
29 }
```

Figure 4.2: A test of the user create function

```
**********This is INIT (SUPERUSER) function!!**********
*** SAMPLE SINGLE PROCESSOR MULTIUSER APPLICATION ***
Creating and starting an application task

user1 created!!
id_ptr tag: <User 1,User 1,CP=true,WorldReadable=false,Read-Write Data Memory>
Task_id[ 1 ] tag: <User 1,User 1,CP=true,WorldReadable=false,Read-Write Data Memory>
Task_id[ 2 ] tag: <User 1,User 1,CP=true,WorldReadable=false,Read-Write Data Memory>
Task_id[ 3 ] tag: <User 1,User 1,CP=true,WorldReadable=false,Read-Write Data Memory>
Task_id[ 4 ] tag: <User 1,User 1,CP=true,WorldReadable=false,Read-Write Data Memory>

user2 created!!
id_ptr2 tag: <User 2,User 2,CP=true,WorldReadable=false,Read-Write Data Memory>
Task_id[ 5 ] tag: <User 2,User 2,CP=true,WorldReadable=false,Read-Write Data Memory>
Task_id[ 6 ] tag: <User 2,User 2,CP=true,WorldReadable=false,Read-Write Data Memory>
Task_id[ 7 ] tag: <User 2,User 2,CP=true,WorldReadable=false,Read-Write Data Memory>

The current running task is 0
```

Figure 4.3: Result of a successful call of user create function

The four tasks created for user1 are different from the three tasks for user2. User1's tasks keep printing out the time every five seconds. After starting user2's tasks, the tasks are suspended for 15 seconds and then wake up to attempt to delete user1's tasks.

The seven tasks run concurrently. Figure 4.5 shows output of the tasks running without the tag engine turned on. At the first 15 seconds, tasks 1 to 4 are running and printing time information every five seconds. After 15 seconds, task 5 starts to delete task 1, task 6 deletes task 3 and task 7 deletes task 4. After successful deletion of these tasks, only task 2 is still alive and running.

With tag engine turned on, the output of running the seven tasks is displayed in Figure 4.6. The first part of this figure is almost identical to what was shown in Figure 4.5. This shows that user1's four tasks are running concurrently. The difference occurs when user2's task (task 5) wants to delete user1's task (task 1); the tag engine generates an exception, prints an error message and stops the system. From the information given by the tag engine, the call is successfully made and the new PC's tag is generated correctly (`(USER2, TASK EXTERNAL, false | READWRITE | DATA_MEMORY | WORLD_NOT_READABLE)`). However the exception is raised on a `LD` instruction that the

```
1  /*create and start tasks 1 to 4 for user1*/
2  void App_task()
3  {
4    Task_name[1] = rtems_build_name( 'T', 'A', '1', ' ' );
5    status = rtems_task_create( Task_name[1], 1, RTEMS_MINIMUM_STACK_SIZE
         ,
6                RTEMS_INTERRUPT_LEVEL(0), RTEMS_DEFAULT_ATTRIBUTES, &
                     Task_id[1]);
7    if(!status) {printf("\nTask 1 created");}
8    else {printf("\nFailed to create task 1");}
9    ...
10   status = rtems_task_start( Task_id[ 5 ], Test_task, 5 );
11   if(!status) {printf("\nTask 5 starting");}
12   else {printf("\nFailed to start task 5");}
13   ...
14 }
15
16 /*create and start tasks 5 to 7 for user2*/
17 void tick_test()
18 {
19   status = rtems_task_create(
20     Task_name[ 5 ], 1, RTEMS_MINIMUM_STACK_SIZE * 2,
          RTEMS_DEFAULT_MODES,
21     RTEMS_DEFAULT_ATTRIBUTES, &Task_id[ 5 ]
22   );
23   if(!status) {printf("\nTask 5 created");}
24   else {printf("\nFailed to create task 5");}
25   ...
26   status = rtems_task_start( Task_id[ 5 ], Test_task, 5 );
27   if(!status) {printf("\nTask 5 starting");}
28   else {printf("\nFailed to start task 5");}
29 }
```

Figure 4.4: Create and start tasks for user1 and user2

```
^^^^^^IN app task, the task_index2 = 1
 - rtems_clock_get_tod - 05:00:00   12/31/1988
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:00   12/31/1988
^^^^^^IN app task, the task_index2 = 3
 - rtems_clock_get_tod - 05:00:00   12/31/1988
^^^^^^IN app task, the task_index2 = 4
 - rtems_clock_get_tod - 05:00:00   12/31/1988
^^^^^^IN app task, the task_index2 = 1
 - rtems_clock_get_tod - 05:00:05   12/31/1988
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:05   12/31/1988
^^^^^^IN app task, the task_index2 = 3
 - rtems_clock_get_tod - 05:00:05   12/31/1988
^^^^^^IN app task, the task_index2 = 4
 - rtems_clock_get_tod - 05:00:05   12/31/1988
^^^^^^IN app task, the task_index2 = 1
 - rtems_clock_get_tod - 05:00:10   12/31/1988
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:10   12/31/1988
^^^^^^IN app task, the task_index2 = 3
 - rtems_clock_get_tod - 05:00:10   12/31/1988
^^^^^^IN app task, the task_index2 = 4
 - rtems_clock_get_tod - 05:00:10   12/31/1988
^^^^^^IN app task, the task_index2 = 1
 - rtems_clock_get_tod - 05:00:15   12/31/1988

******IN tick task, the task_index2 = 5
 - ***rtems_clock_get_tod - 05:00:00   12/31/1988
 task-5 delete task 1!
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:15   12/31/1988

******IN tick task, the task_index2 = 6
 - ***rtems_clock_get_tod - 05:00:00   12/31/1988
 task-6 delete task 3!

******IN tick task, the task_index2 = 7
 - ***rtems_clock_get_tod - 05:00:00   12/31/1988
 task-7 delete task 4!
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:20   12/31/1988
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:25   12/31/1988
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:30   12/31/1988
```

Figure 4.5: User2 delete user1's task without tag engine turned on

```
^^^^^^IN app task, the task_index2 = 1
 - rtems_clock_get_tod - 05:00:00   12/31/1988
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:00   12/31/1988
^^^^^^IN app task, the task_index2 = 3
 - rtems_clock_get_tod - 05:00:00   12/31/1988
^^^^^^IN app task, the task_index2 = 4
 - rtems_clock_get_tod - 05:00:00   12/31/1988
^^^^^^IN app task, the task_index2 = 1
 - rtems_clock_get_tod - 05:00:05   12/31/1988
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:05   12/31/1988
^^^^^^IN app task, the task_index2 = 3
 - rtems_clock_get_tod - 05:00:05   12/31/1988
^^^^^^IN app task, the task_index2 = 4
 - rtems_clock_get_tod - 05:00:05   12/31/1988
^^^^^^IN app task, the task_index2 = 1
 - rtems_clock_get_tod - 05:00:10   12/31/1988
^^^^^^IN app task, the task_index2 = 2
 - rtems_clock_get_tod - 05:00:10   12/31/1988
^^^^^^IN app task, the task_index2 = 3
 - rtems_clock_get_tod - 05:00:10   12/31/1988
^^^^^^IN app task, the task_index2 = 4
 - rtems_clock_get_tod - 05:00:10   12/31/1988
^^^^^^IN app task, the task_index2 = 1
 - rtems_clock_get_tod - 05:00:15   12/31/1988

******IN tick task, the task_index2 = 5
 - ***rtems_clock_get_tod - 05:00:00   12/31/19
 CALL instruction tag propagation
 tag of addr is = <Task External,Task External,CP=false,WorldReadable=false,Read-Write Entry-point Code Memory>
 tag of original pc is = <User 2,User 2,CP=false,WorldReadable=false,Read-Write Data Memory>
 new calculated pc tag is = <User 2,Task External,CP=false,WorldReadable=false,Read-Write Data Memory>
 new set pc tag is = <User 2,Task External,CP=false,WorldReadable=false,Read-Write Data Memory>

LD_IMM CHECK ERROR when copy bit of source address is set
 own_pc = User 2 does NOT DOMINATE
 own_addr = User 1
88
Unexpected trap (40) at address 0x020019AC
```

Figure 4.6: User2 delete user1's task with tag engine turned on

Owner field of the PC's tag (`USER2`) does not dominate the Owner field of the address's tag (`USER1`). This is because when task 5 tries to delete task 1, it uses the `rtems_task_delete` directive and passes `Task_id[1]` to it. Then, `rtems_task_delete` directive tries to get the specific thread by using `Task_id[1]`, which is tagged with user1's tag, and therefore generates a LD error message.

Although RTEMS is not fully tagged and the tag engine has to be turned on manually in the code, the test case shows that the security tagging scheme for multiple users is working and can be used to help isolate tasks of different users.

4.2 Tagging Test Cases

To support the UI tagging scheme presented in this dissertation, modifications to SIS were required for tag checking and tag propagation when executing instructions. In order to test that all tag checking and propagations worked as intended, 76 test cases (around 68000 lines of C code) were manually generated, each containing up to 64 smaller tests.

In these test cases, the memory locations of the variables and the PC were manually tagged when the tag engine is off, then the test turns on the tag engine and executes the statement. After that, the test turns off the tag engine and prints the results of the tag propagation, or security exception.

Table 4.2 shows the basic utility functions developed for dealing with tags and special bits. For example, tags are specified by using `make_tag()`, and a word of memory is tagged with a specific tag using `_rtems_tag_word()`. The `_rtems_get_memory_tag()` function can be used to return the tag of a specific memory. The PC is tagged and the tag of the PC can be checked with `_rtems_tag_pc()` and `_rtems_get_pc_tag()` respectively. In addition, the domination relationship of two labels (label represents a 12 bits Owner field or Code-space field of a tag) or security classes of two tags can be checked. Function `tag_lub()` is used to calculate the least upper bound of two tags. In addition, there are functions that set or unset the copy bit and world readable bit, and change the memory types for testing purposes only.

4.2.1 Tagging test cases for assignment in C language

To help explain the test cases, sample tests are listed in Figure 4.7. The figure shows part of a test case for assignment in the C language. Different tags are associated with variables and the PC to test the tag engine rules.

- On lines 3 to 6, four tags are created. The variable `tag4_ucp` stores the tag `(USER1, WATCHDOG_EXT, false, READWRITE | DATAMEMORY)`. The Owner field of the tag

Table 4.2: Summary of test case utility functions

make_tag()	Make a 32-bit tag of three fields. (Owner, Codespace, Control)
set_copy_bit()	Set the copy bit of a tag.
reset_copy_bit()	Unset the copy bit of a tag.
_rtems_tag_pc()	Tag the PC (Program Counter) with a specific tag.
_rtems_get_pc_tag()	Get the tag of the PC.
_rtems_tag_word()	Tag the word with a specific tag.
_rtems_get_memory_tag()	Get the tag of a memory.
label_lub()	Calculate the LUB of two labels.
label_glb()	Calculate the GLB of two labels.
label_dominates_or_eq()	Give the domination relationship of two labels.
tag_lub()	Calculate the LUB of two tags.
tag_glb()	Calculate the GLB of two tags.
tag_dominates_or_eq()	Give the domination relationship of two tags.
set_world_readable()	Set the world readable bit of a tag.
reset_world_readable()	Unset the world readable bit of a tag.
set_read_write()	Set the memory type to read/write.
set_read_only()	Set the memory type to read-only.
set_code_mem_entry()	Set the memory type to a entry point of a code memory.
set_code_mem_not_entry()	Set the memory type to a non-entry point of a code memory.
set_stack_memory()	Set the memory type to stack memory.
set_data_memory()	Set the memory type to data memory.

```
1  void test4()
2  {    ...
3       tag4_ucp = make_tag(USER1,WATCHDOG_EXT, false ,READWRITE|DATA_MEMORY)
           ;
4       tag2_ucp = make_tag(USER1,REGION_EXT, false ,READWRITE|DATA_MEMORY);
5       tag4 = make_tag(USER1,WATCHDOG_EXT, true ,READWRITE|DATA_MEMORY);
6       tag1_ucp = make_tag(USER1,USER1, false ,READWRITE|DATA_MEMORY);
7       asm(CPOP_DEBUG_ON);
8       printf("****** 4-1. value2 < value1 < PC ******\n");
9       value1 = 1; value2 = 10;
10      printf("value1 = %d, value2 = %d\n",value1, value2);
11      _rtems_tag_word((addr_t)&value1, tag2_ucp);
12      _rtems_tag_word((addr_t)&value2, tag1_ucp);
13      _rtems_tag_pc(tag4_ucp);
14      printf ...
15      start_tagging();
16      value1 = value2;
17      asm(CPOP_TURN_OFF_TAGGING);
18      new_tag = _rtems_get_memory_tag((addr_t)&value1);
19      printf("OPERATION: value1 = value2, new tag of value1(%d) is:\n %s\
           n", value1, tag_to_string(str, new_tag));
20      ...
21      printf("****** 4-32. value1 < pc < value2 ******\n");
22      value1 = 32; value2 = 320;
23      printf("value1 = %d, value2 = %d\n",value1, value2);
24      _rtems_tag_word((addr_t)&value1, tag1_ucp);
25      _rtems_tag_word((addr_t)&value2, tag4);
26      _rtems_tag_pc(tag2_ucp);
27      printf ...
28      start_tagging();
29      value1 = value2;
30      asm(CPOP_TURN_OFF_TAGGING);
31      new_tag = _rtems_get_memory_tag((addr_t)&value1);
32      printf("OPERATION: value1 = value2, new tag of value1 (%d) is:\n %s
           \n", value1, tag_to_string(str, new_tag));
33      ...
34      asm(CPOP_DEBUG_OFF);
35 }
```

Figure 4.7: Sample test case

is USER1, Code-space field is WATCHDOG_EXT (watchdog external function). The Control-bits field shows the tagged data is read/write data memory with the copy bit not set. The variable tag4 stores a tag that similar to tag4_ucp, but has the copy bit set. The variable tag2_ucp stores a tag with Code-space field is REGION_EXT (region external function). Variable tag1_ucp has a tag with both Owner field and Code-space field set to USER1, and copy bit not set.

- On line 7, a CPOP_DEBUG_ON instruction is used to turn on the debugging. After turning on the debugging, error information will be printed when a tagging exception is generated.

- Lines 8 to 20 show one of the 42 subcases (test 4-1) in test 4, and lines 21 to 33 are for another subcase (test 4-32) from test 4. The outputs of these tests are shown in Figure 4.8.

- On line 9, the two variables: value1 and value2, are initialized to 1 and 10 respectively.

- On lines 11 to 12, the _rtems_tag_word() function is used to give tags to value1 and value2. After executing lines 11 and 12, value1 will be tagged with (USER1, REGION_EXT, false, READWRITE | DATAMEMORY), and value2 will be associated with tag (USER1, USER1, false, READWRITE | DATAMEMORY).

- On line 13, _rtems_tag_pc() function gives the PC a tag (USER1, WATCHDOG_EXT, false, READWRITE | DATAMEMORY).

- Before turning on the tagging scheme, line 14 prints out the tags of value1, value2 and PC.

- On line 15, the tagging scheme is turned on. The function start_tagging() directs GCC to not use values previously stored in registers[1], and then executes

[1]There may be unexpected interruptions if GCC uses the values stored in the registers.

the CPOP_TURN_ON_TAGGING instruction (asm (CPOP_TURN_ON_TAGGING);) to turn on tag engine in the simulator.

- Line 16 performs the assignment value1 = value2;.
- On line 17, asm(CPOP_TURN_OFF_TAGGING); is used to turn off the tag engine.
- After executing the assignment with the tag engine turned on, the tag should be propagated correctly as specified by the tagging rules.
- On line 19, the function _rtems_get_memory_tag((addr_t) &value1) is used to get the new tag associated with value1. According to the tagging rules for assignment (See Section. 3.4.4), if both value1 and value2 have copy bit not set, to allow the information flow from value2 to value1, the security level of value1 and value2 have to be lower than the level of the PC's security level. On line 13, the PC is tagged with tag (USER1, WATCHDOG_EXT, false, READWRITE | DATAMEMORY). Since value1 has tag (USER1, REGION_EXT, false, READWRITE | DATAMEMORY) and value2 is tagged with (USER1, USER1, false, READWRITE | DATAMEMORY), both security levels of value1 and value2's tags are lower than the security level of the PC's tag. Therefore the assignment is allowed and no exception will be generated. The value of value2 will be stored in value1, but the tag of value1 should not be changed based on the tagging rules. This is confirmed by the output in Figure 4.8.
- Figure 4.8 shows the output of the sample test in lines 1 to 9. For test 4-1, when both copy bits of value1 and value2's tags are not set, after executing the assignment (value1 = value2;), value1 gets value2's value but its tag doesn't change (see line 8 in Figure 4.8).
- In the other small test (test 4-32), the values stored in value1 and value2 are changed to 32 and 320, and different tags are given to value1, value2 and the PC.

- Lines 24 to 26 show that in this test (test 4-32), the variable `value1` is tagged `(USER1, USER1, false, READWRITE | DATAMEMORY)`, the variable `value2` has tag `(USER1, WATCHDOG_EXT, true, READWRITE | DATAMEMORY)` associated with it and the PC has tag `(USER1, REGION_EXT, false, READWRITE | DATAMEMORY)`.

- On lines 29 to 30, the tagging scheme is turned on and the assignment is executed. According to the tagging rules, if the copy bit of `value1`'s tag is $\overline{\texttt{cp}}$ and the copy bit of `value2`'s tag is `cp`, the conditions $\underline{\texttt{value1}} \leq \underline{\texttt{PC}}$ and `Owner(value1) = Owner(value2)` have to be met. The owner in both tags is `USER1`, and the security level of the tag of `value1` (`USER1, USER1`) is lower than the security level of the PC's tag (`USER1, REGION_EXT`), therefore the assignment is allowed. The tag of `value2` will be copied to the tag of `value1`.

The output for test 4-32 is shown in Figure 4.8 starting at line 11. For test 4-32, the copy bit of `value2`'s tag is `cp` while $\overline{\texttt{cp}}$ for `value1`'s tag. Since the copy bit of `value2`'s tag is set, `value2`'s tag will be copied to `value1`'s tag (see line 19 in Figure 4.8).

In another test, test 4-33 (source is not shown, but output is in Figure 4.9), the PC has tag `(USER1, USER1, false, READWRITE | DATAMEMORY)`. The variable `value1` has tag `(USER1, REGION_EXT, false, READWRITE | DATAMEMORY)` and `value2` has tag `(USER1, WATCHDOG_EXT, true, READWRITE | DATAMEMORY)`. Similar to tests 4-32, the copy bit of `value2`'s tag is `cp` while $\overline{\texttt{cp}}$ for `value1`'s tag. However, the assignment will not be allowed, because the condition $\underline{\texttt{value1}} \leq \underline{\texttt{PC}}$ is not met. Therefore, a security exception will be raised. A tagging exception handler will handle this and print out information about the error. As shown in Figure 4.9, lines 7 to 14 are information about the violation. It indicates that the violation is caused by the `ST` instruction and the reason is the relationship of the PC's tag and `value1`'s tag is not satisfied.

Because of the violation, the assignment will not be executed. Therefore, on lines 15 and 16 (Figure 4.9), the new tag of `value1` is its original tag and the value of `value2` (330) is not copied to `value1`.

```
1 -value1 cp not set, value2 cp not set-
2 ****** 4-1. value2 < value1 < PC ******
3 value1 = 1, value2 = 10
4 PC tag: <User 1,Watchdog External,CP=false,WorldReadable=false,Read-
      Write Data Memory>
5 value1 tag: <User 1,Region External,CP=false,WorldReadable=false,Read-
      Write Data Memory>
6 value2 tag: <User 1,User 1,CP=false,WorldReadable=false,Read-Write Data
       Memory>
7
8 OPERATION: value1 = value2, new tag of value1(10) is:
9  <User 1,Region External,CP=false,WorldReadable=false,Read-Write Data
       Memory>
10
11 -value1 cp set, value2 cp not set-
12 ****** 4-32. value1 < pc < value2 ******
13 value1 = 32, value2 = 320
14 pc tag is: <User 1,Region External,CP=false,WorldReadable=false,Read-
      Write Data Memory>
15 value1 tag: <User 1,User 1,CP=false,WorldReadable=false,Read-Write Data
       Memory>
16 value2 tag: <User 1,Watchdog External,CP=true,WorldReadable=false,Read-
      Write Data Memory>
17
18 OPERATION: value1 = value2, new tag of value1 (320) is:
19  <User 1,Watchdog External,CP=true,WorldReadable=false,Read-Write Data
       Memory>
```

Figure 4.8: Sample test case output without exception

```
1  ****** 4-33. pc < value1 < value2 ******
2  value1 = 33, value2 = 330
3  pc tag is: <User 1,User 1,CP=false,WorldReadable=false,Read-Write Data
       Memory>
4  value1 tag: <User 1,Region External,CP=false,WorldReadable=false,Read-
       Write Data Memory>
5  value2 tag: <User 1,Watchdog External,CP=true,WorldReadable=false,Read-
       Write Data Memory>
6
7  ST_IMM2 CHECK ERROR when copy bit of source is set and addr is not set,
       non-stack memory
8   tag_pc = <User 1,User 1,CP=false,WorldReadable=false,Read-Write Data
       Memory> does NOT DOMINATE
9   tag_addr = <User 1,Region External,CP=false,WorldReadable=false,Read-
       Write Data Memory>
10
11 ...
12  Hit vector 0x28!! @ PC = 0x 20EE484 with NPC = 0x20EE488
13  RETT at 0x20fb900 with addr =0x20ee48c
14  Violating Instruction: 0xC227BFFC :: ST
15 OPERATION: value1 = value2, new tag of value1 (33) is:
16  <User 1,Region External,CP=false,WorldReadable=false,Read-Write Data
       Memory>
```

Figure 4.9: Sample test case output with exception

```
1  void test63()
2  { ...
3    tag2_ucp = make_tag(USER1,TASK_EXT,false,READWRITE | DATA_MEMORY);
4    tag2_ucp_user2 = make_tag(USER2,TASK_EXT,false,READWRITE |
         DATA_MEMORY);
5    asm(CPOP_DEBUG_ON);
6    ...
7    printf("test 63-4: value1 cp not set, value1 =/= imm = PC (USERS)\n")
         ;
8    _rtems_tag_word((addr_t)&value1, tag2_ucp_user2);
9    printf("value1's tag is:\n %s\n", tag_to_string(str,
         _rtems_get_memory_tag((addr_t)&value1)));
10   _rtems_tag_pc(tag2_ucp);
11   printf("PC tag is:\n %s\n\n", tag_to_string(str, _rtems_get_pc_tag())
         );
12
13   start_tagging();
14   if(value1) {;}
15   asm(CPOP_TURN_OFF_TAGGING);
16   printf("ASSIGNMENT: if(value1) {;}, tag of value1 (%d) is :\n %s\n\n\
         n", value1, tag_to_string(str, _rtems_get_memory_tag((addr_t)&
         value1)));
```

Figure 4.10: Sample test case for if statement

4.2.2 Tagging test cases for different users

Because the tagging scheme has been redesigned from the original version [22] to support multiple users, additional test cases are generated to test multiuser tags. Figure 4.10 shows one of the multiuser test cases.

In this test case, the `if` statement is tested using different user tags (`USER1` and `USER2`). The variable `value1` is tagged with a tag that belongs to user2 (line 8). The PC is tagged with a tag with the Owner field set to `USER1` (line 10). Then the tag engine is started on line 13, and the if statement, `if(value1) ;` is tested on line 14. The result is shown in Figure 4.11; the `if` statement is not allowed. Because the PC's tag does not dominate the address's tag, `value1` can not be read by the current running thread.

```
---------- test 63-4: value1 cp not set, value1 ≠ imm = PC (USERS)----------
value1's tag is:
 <User 2,Task External,CP=false,WorldReadable=false,Read-Write Data Memory>
PC tag is:
 <User 1,Task External,CP=false,WorldReadable=false,Read-Write Data Memory>


LD_IMM CHECK ERROR when copy bit of source address is not set and not world readable
 own_pc = <User 1,Task External,CP=false,WorldReadable=false,Read-Write Data Memory> does NOT DOMINATE
 tag_addr = <User 2,Task External,CP=false,WorldReadable=false,Read-Write Data Memory>
...
 Hit vector 0x28!! @ PC = 0x2052CE4 with NPC = 0x2052CE8

RETT at 0x20fbe7c with addr =0x2052cec
 other
  Violating Instruction: 0xC207BFFC :: LD
EXCEPT: Test 63-4 PASSES
ASSIGNMENT: if(value1) {;}, tag of value1 (100) is :
 <User 2,Task External,CP=false,WorldReadable=false,Read-Write Data Memory>
```

Figure 4.11: Output for If statement test for different users

4.3 Summary

This chapter first discusses the enhanced tagging scheme which supports a multiuser system. It applies the idea of superuser and non-privileged user to RTEMS. By using different user tags, users and their resources can be isolated from other users and resources. A user manager has been added to RTEMS and a sample test of multiuser support is illustrated in the first section of this chapter.

The test cases shown in the second section are a small sample of the whole suite of test cases. There are test cases for different groups of operations which include arithmetic, logic, shifting operations, assignment, if/while statements, function call and return, pointers, and so on. These test cases cover many possibilities of tag checking and propagation, including many combinations of relationships between the PC's tag and the variable tags, memory types and control bits. These test cases were used to help refine the tagging rules and debug the implementation of the tag engine.

The test cases for arithmetic, logic and shifting operations ensure that the new tag is correctly propagated from the two operands' tags. The assignment tests verify that the tag checking and propagation rules are completely implemented in the tag engine. The test cases for function call and return make sure the function execution control rules are

correctly implemented and the PC's tag is propagated successfully. In addition, knowing how important the pointers are, test cases for pointers are generated to ensure correctness of pointers' tags.

As the hardware which could support the UI tagging scheme is not available, the tag engine had to be implemented in the simulator of the SPARC architecture. Therefore, these test cases test the correctness of the tag engine and help verify the tag checking and propagation rules are properly designed. These test cases were designed to easily port to the hardware once it is available.

Chapter 5: ACL2 Models and Proofs

One of the objectives of the UI tagging project is to develop a formal model of the security policy enforced using the UI tagging scheme and prove security properties of the tagging system. This chapter describes the process of developing a formal model for the UI tagging scheme in ACL2. ACL2 [14] stands for "A Computational Logic for Applicative Common LISP". The language is based on Common LISP. In addition to providing an execution engine, ACL2 provides a reasoning engine to help prove theorems about the models. The theorem prover is based on rewriting terms. The process includes specifying the tag format, propagation and checking rules in ACL2. Then these tag checking and propagation functions have been applied to corresponding assembly instructions. There is a state list in ACL2 which keeps track of the system states and resources. When executing an instruction, the system first checks the tagging rules to determine whether the instruction is allowed. If it is allowed, the corresponding register or memory location and its tag will be updated in the state list. Otherwise, the model generates a security exception and stops the evaluation.

Since, in the lattice, security classes are arranged in a partial ordering, lemmas are proven that the reflexive, transitive and antisymmetric properties hold for all the classes. In addition, the model for the UI tagging scheme has been proven to be a lattice. Other security related lemmas and theorems have also been proven and are described.

The remainder of this chapter is organized as follows: Section 5.1 introduces the ACL2 model of the tag, and shows some tagging related utility functions. Then in Section 5.2, the ACL2 model of tag checking functions for various instructions are described. The formal model of the security policy enforced by the UI tagging system is discussed in Section 5.3. Section 5.4 describes the instruction execution and tag propagation functions for different groups of instructions in ACL2. Some security properties of the system architecture are proven in Section 5.5 and Section 5.6. Lastly, Section 5.7 summarizes this chapter.

5.1 Tag Representation in ACL2

This chapter discusses the ACL2 model of the tagging engine. A security tag is represented as the list `(LABEL1 LABEL2 (CPSET RANDW DATAMEM WORLDSET))` in ACL2. The first and second elements in the list, `LABEL1` and `LABEL2`, define the Owner field and the Code-space field. The third part models the Control-bits field. In this example, `LABEL1` and `LABEL2` are place holders and the Control-bits field shows possible values. There are more possible values for this field, such as: `CPNOTSE`, `RONLY`, `STACKMEM`, `EXEENTRY`, `WORLDNOTSET`, etc, described in Section 5.1.2.

5.1.1 Owner field and Code-space field in ACL2

For the Owner field and Code-space field, ACL2 constants are used to represent different entities in the tagging system. For example, `USER1`, `USER2`, `SUPERUSER`, `USER` are used to represent different users and the general user level. On the manager level, `TASK_EXT`, `TASK_INIT`, `TASK_INT` and `TASK_PINT`, represent task external class, task initialization class, task internal class and task private internal class accordingly. Similar labels, such as `PARTITION_EXT`, `CLOCK_INT`, `MESSAGE_INIT` exist for all other managers. Above the manager level, the SCORE services are divided into the same internal, external, initialization, and private internal classifications, such as `OBJECT_EXT`, `THREAD_INIT` and `WORKSPACE_INT`. The complete list of services and managers is shown in Table 3.4.

Since ACL2 provides both a theorem prover and an execution engine, the tagging scheme is able to be modeled as well as user code can be interpreted. To facilitate this, user assembly language is converted into a lisp data structure, whose representation includes the actual data bits for tags. The ACL2 model therefore includes functions to map these bit representations to the ACL2 labels. The mapping is defined in a "code-table" and facilitates the dominates and other utility functions.

As shown in Section 3.3.1, each of the Owner and Code-space fields is represented with

12 bits, which can be divided into RTEMS/USER bits (4 bits), RTEMS level bits (3 bits) and Component ID bits (5 bits). This is mapped into lists of the form `(TASK_EXT 1 0 2)` for task manager external class, which has bit representation `1111 000 00010`. The bit representation for task private internal class is `1111 011 00010`, in ACL2, `(TASK_PINT 1 3 2)` is assigned to it. Compare the representations for both task manager external class `(1 0 2)` and task manager private internal class `(1 3 2)` in ACL2. The 1s in both representations indicate they are RTEMS data/code instead of user data/code. The third numbers, which represent RTEMS component IDs are the same, since both of them are for task manager. The difference between the two representations is the second number, which indicates the RTEMS levels. Manager external functions are at level 0, while manager private internal functions are at level 3. Similarly, `(THREAD_INT 1 6 1)` represents thread initialization class at the SCORE level, `(WORKSPACE_EXT 1 4 7)` shows the workspace external class and `LOW` is represented by `(0 0 0)`. The code table used for labels in ACL2 is shown in Table 5.1.

Comparing two labels can be done by simply comparing the numbers in the code table. The `label-dominate` function (See Section 5.1.3) checks the dominate relationship between two labels based on the code table and the lattice. The `security-class-dominate` function uses the `label-dominate` function to check both Owner field labels and Codespace field labels.

5.1.2 Control-bits field in ACL2

As discussed in Section 3.1.3, the control-bit field takes 8 bits. The highest bit (bit 7) is used as a copy bit, is represented with labels `CPSET` and `CPNOTSET`. The copy bit is followed by 3 bits which indicate the memory type and are represented with two separate label fields. Label `RONLY` means the memory is read only, while label `RANDW` allows read and write on the tagged data/code. After the read/write control labels, another four labels `(DATAMEM, STACKMEM, EXEENTRY, EXENONEENTRY)` are used to show the memory

Table 5.1: Code table for labels of Owner field and Code-space field in ACL2

Code representation			Functions
RTEMS Level			
RTEMS/ USER (4 bits)	RTEMS Level (3 bits)	Component ID (5 bits)	RTEMS Representation
1 (RTEMS)	7	31	Tag INIT, SCORE, HIGH
		1-13	SCORE private internal functions
		0	
	6	31	
		1-13	SCORE internal functions
		0	
	5	31	
		1-13	SCORE internal init functions
		0	SCORE INIT
	4	31	
		1-13	SCORE external functions
		0	
	3	31	MANAGER
		1-20	Manager private internal functions
		0	
	2	31	
		1-20	Manager internal functions
		0	
	1	31	
		1-20	Manager internal init functions
		0	Manager INIT
	0	1	
		1-20	Manager external functions
		0	Startup
USER LEVEL			
0	1	1	USER
		0	SUPERUSER
	0	n	USERn
		...	...
		2	USER2
		1	USER1
		0	LOW

types. `DATAMEM` and `STACKMEM` indicate data memory and stack memory. Both `EXEENTRY` and `EXENONEENTRY` represent code memory, however `EXEENTRY` differentiates entry point to a function from non-entry point code memory. The world readable bit (bit 3) is represented with `WORLDSET` or `WORLDNOTSET` indicating whether or not the tagged data is world readable. The list of Control-bits field is formed by choosing one label from each of the four groups respectively. For instance, a Control-bits field shown as `(CPNOTSET RONLY EXEENTRY WORLDNOTSET)` in ACL2 indicates a read only entry point of code memory with the copy bit not set and it is not world readable.

5.1.3 Utility functions for tagging

After developing the tag model in ACL2, several utility functions were written to help with the tag checking and propagation, such as checking the tag format, checking the dominates relation between two tags. These functions made the tag checking and modification much easier.

Figure 5.1 shows some functions which are used to check whether or not the format of a tag is correct. The parameter `spec` is used to represent the static specification of the system and is described in Section 5.3.

- The first function (`label-format-check`) is a format check function for the Owner field label and Code-space field label. It is a predicate that compares the given label with the code table to find out whether it is a valid label.

- The second function (`control-label-check`) is a predicate that checks the format of the Control-bits field label. The code checks the length of the Control-bits label, which should be four, and the copy bit label, the read/write permission label, memory type label and world readable label. These checks make sure that all the labels are in correct format.

- The `tag-format-check` function is a predicate that uses `label-format-check` and

```
1  (defun label-format-check (code-label spec)
2    (let ((code-table (second (assoc 'TABLE spec))))
3         (is-a-member code-label (auto-list code-table))))
4
5  (defun control-label-check (control-label)
6    (and (equal (length control-label) 4)
7         (control-copybit-check control-label)
8         (control-rw-check control-label)
9         (control-type-check control-label)
10        (control-world-check control-label)))
11
12 (defun tag-format-check (one-tag spec)
13   (and (equal (length one-tag) 3)
14        (label-format-check (get-owner-label one-tag) spec)
15        (label-format-check (get-code-label one-tag) spec)
16        (control-label-check (get-control-label one-tag))))
```

Figure 5.1: Format check functions in ACL2

`control-label-check` functions to verify that the format for the three fields are correct. It also checks that the length of the tag is three, since the tag in ACL2 consists of one Owner field label, one Code-space field label and one list which represents the Control-bits field.

In Figure 5.2, part of the `label-dominate` function is shown. The function is a predicate that checks whether `label1` dominates `label2` following the lattice discussed in Section 3.2. This function first checks if one label is `LOW` or `HIGH` or both labels are the same, for a simple response. If this is not the case, then the function checks the relationship between levels and then within levels.

In addition to the functions discussed in this section, there are many other functions that have been written in ACL2 to deal with the tag checking and modification. These include setting or unsetting the copy bit of a tag, making a tag based on the given labels, calculating the `LUB` of two labels and so on.

```
(defun label-dominate (label1 label2 spec)
  (let ((label1-entry (get-code-reps label1 spec))
        (label2-entry (get-code-reps label2 spec))
        (score-level (second (assoc 'SCORE-LEVELS spec)))
        (rtems-level (second (assoc 'RTEMS-LEVELS spec))))
   (let ((label1-name (first label1-entry))
         (label1-rtems-flag (second label1-entry))
         ...
         (label2-component (fourth label2-entry)))
     (cond ((or (equal label2-name 'LOW)
                (equal label1-name 'HIGH)
                (equal label1-name label2-name)) t)
             ...
           ((and (equal label1-name 'SCORE)
                 (not (equal label2-name 'TAG_STARTUP))
                 (not (equal label2-name 'HIGH))) t)
           ((and (equal label1-name 'USER)
                 (equal label2-rtems-flag 0)) t)
           ((and (equal label1-rtems-flag 1)
                 (equal label2-rtems-flag 0)) t)
           ((and (equal label1-rtems-flag 1)
                 (equal label2-rtems-flag 1))
       (cond ((and (is-a-member label1-level score-level)
                   (is-a-member label2-level rtems-level)) t)
             ((and (is-a-member label1-level score-level)
                   (is-a-member label2-level score-level))
           (cond ((and (not (equal label1-level 4))
                       (equal label2-name 'SCORE_STARTUP)) t)
                 ((and (> label1-level label2-level)
                       (equal label1-component label2-component)) t)
             (t nil)))
               ((and (is-a-member label1-level rtems-level)
                     (is-a-member label2-level rtems-level))
           (cond ...)
           (t nil)))))))
```

Figure 5.2: Label dominate function in ACL2

5.2 Tag Checking for Different Instructions in ACL2

As mentioned in Section 3.5, SPARC assembly instructions can be divided into five classes: (1) Branch related instructions, (2) Call related instructions, (3) Arithmetic, logic and shifting instructions, (4) Load and store instructions and (5) Save and restore instructions. This section introduces how tag checking is modeled in ACL2 for each class of instructions.

5.2.1 Tag checking for BRANCH instructions

Whether or not a branch instruction is executed depends on the value of the integer condition codes (icc). The integer condition codes can be modified by the arithmetic and logical instructions whose names end with "cc", such as `ANDcc`, `SUBcc`. According to the tagging rules for branch instructions (see Section 3.5.1), the branch must made to executable code memory within the same Code-space.

The branch instructions tag checking code in ACL2 is shown in Figure 5.3. The `branch-inst-tag` function first checks the Control-bits field of the destination address's tag to make sure the memory type label is either `EXEENTRY` or `EXENONEENTRY`. If this is the case, the code further checks that the branch is made in the same code space and the current running thread is able to read the `icc`'s value.

5.2.2 Tag checking for CALL related instructions

The code for tag checking for the `CALL` instruction is shown in Figure 5.4. Whether or not the `CALL` instruction is allowed to be executed or not, depends on the function execution control rule. The function execution control rules (see Section 3.4.7) specify which callers are allowed to call which specific functions. The ACL2 code for `CALL` instruction is shown in Figure 5.4. The `call-inst-tag` function gets the destination address's tag, PC's tag and the `spec` which has instruction list information. The code first checks the format of

```
(defun branch-inst-tag (mem-tag pc-tag icc-tag spec)
  (if (and (tag-format-check mem-tag spec)
           (tag-format-check pc-tag spec)
           (tag-format-check icc-tag spec))
    (let ((mem-tag-control (get-control-label mem-tag))
          (mem-tag-code (get-code-label mem-tag))
          (pc-tag-code (get-code-label pc-tag)))
      (if (or (equal 'EXEENTRY (get-control-type mem-tag-control))
              (equal 'EXENONEENTRY (get-control-type mem-tag-control)))
        (if (and (equal mem-tag-code pc-tag-code)
                 (tag-dominate pc-tag icc-tag spec))
          pc-tag nil)
        nil))
    nil))
```

Figure 5.3: Tag checking on BRANCH instruction in ACL2

```
(defun call-inst-tag (addr-tag pc-tag spec)
  (if (and (tag-format-check addr-tag spec)
           (tag-format-check pc-tag spec)
           (function-control-tag pc-tag addr-tag spec))
      (function-control-tag pc-tag addr-tag spec)
    nil))
```

Figure 5.4: Tag checking on CALL instruction in ACL2

the address's tag and PC's tag to make sure both of them are in correct tag format. Then the function uses the `function-control-tag` function to verify that the call is allowed. This function is based on the function execution control rules. If the call is allowed, the function combines the tags and returns the new PC's tag. If it is not allowed, it returns `NIL`. If all of the checks are passed, the `call-inst-tag` function returns the new PC's tag from the `function-control-tag` function, otherwise it returns `NIL` to indicate a fail on the tag checking of the function call.

5.2.3 Tag checking for arithmetic, logic and shifting instructions

Section 3.5.3 introduced the tag checking rules for arithmetic, logic and shifting instructions. The tag propagation rules for arithmetic, logic and shifting instructions can be classified into four groups based on the copy bits of the two operands: (1) both of the operands have the copy bit set, (2) only operand1 has the copy bit set (3) only operand2 has the copy bit set and (4) neither of the two operands have the copy set). Figure 5.5 shows the ACL2 code for tag checking for arithmetic, logic and shifting instructions.

The `arith-inst-tag` function checks the copy bits of the two operands and determines the proper tag propagation rule. For instance, lines 13 to 15 deal with the condition that both of the two operands have the copy bit set, and propagates a new tag by combining the PC tag's Owner field label, the PC tag's Code-space label and a default Control-bits field label with the copy bit unset. Lines 22 to 24 are good for propagating a new tag for two operands which have the copy bit not set. Based on the tag propagation rule shown in Table 3.6, the new tag is formed by calculating the `LUB` of the two tags. Therefore on line 24, the code uses `label-lub` function twice to get the `LUB` of the two Owner field labels and the `LUB` of the two Code-space field labels separately. The code then combines two labels with another Control-bits field label to form a new tag and return it.

5.2.4 Tag checking for LD and ST instructions

The code shown in Figure 5.6 indicates how to implement the tag checking rule for `LD` instruction in ACL2.

The function gets the tag of the memory and the tag of the PC, then checks each field of the tags based on the tagging rule described in Section 3.5.4.

The function `ld-inst-tag` first checks on the tag formats of both tags to ensure

```
1  (defun arith-inst-tag (reg1-tag reg2-tag pc-tag spec)
2   (if (and (tag-format-check reg1-tag spec)
3            (tag-format-check pc-tag spec)
4            (tag-format-check reg2-tag spec))
5       (let ((reg1-tag-owner (get-owner-label reg1-tag))
6             (reg1-tag-code (get-code-label reg1-tag))
7             (reg1-tag-control (get-control-label reg1-tag))
8             (pc-tag-owner (get-owner-label pc-tag))
9             (pc-tag-code (get-code-label pc-tag))
10            (reg2-tag-owner (get-owner-label reg2-tag))
11            (reg2-tag-code (get-code-label reg2-tag))
12            (reg2-tag-control (get-control-label reg2-tag)))
13    (cond ((and (equal 'CPSET (get-control-copybit reg1-tag-control))
14                (equal 'CPSET (get-control-copybit reg2-tag-control)))
15           (make-tag pc-tag-owner pc-tag-code (make-default-control-label
                 'CPNOTSET) spec))
16          ((and (equal 'CPNOTSET (get-control-copybit reg1-tag-control))
17                (equal 'CPSET (get-control-copybit reg2-tag-control)))
18           (make-tag reg1-tag-owner reg1-tag-code (
                make-default-control-label 'CPNOTSET) spec))
19          ((and (equal 'CPSET (get-control-copybit reg1-tag-control))
20                (equal 'CPNOTSET (get-control-copybit reg2-tag-control)))
21           (make-tag reg2-tag-owner reg2-tag-code (
                make-default-control-label 'CPNOTSET) spec))
22          ((and (equal 'CPNOTSET (get-control-copybit reg1-tag-control))
23                (equal 'CPNOTSET (get-control-copybit reg2-tag-control)))
24           (make-tag (label-lub reg1-tag-owner reg2-tag-owner spec) (
                label-lub reg1-tag-code reg2-tag-code spec) (
                make-default-control-label 'CPNOTSET) spec))))
25      nil))
```

Figure 5.5: Tag checking on Arithmetic instruction in ACL2

```
1  (defun ld-inst-tag (mem-tag pc-tag spec)
2    (if (and (tag-format-check mem-tag spec)
3             (tag-format-check pc-tag spec))
4      (let ((mem-tag-owner (get-owner-label mem-tag))
5            (mem-tag-control (get-control-label mem-tag))
6            (pc-tag-owner (get-owner-label pc-tag)))
7        (if (and (equal 'WORLDSET (get-control-world mem-tag-control))
8                 (data-stack-mem mem-tag-control))
9          mem-tag
10           (if (equal 'CPSET (get-control-copybit mem-tag-control))
11             (if (label-dominate pc-tag-owner mem-tag-owner spec)
12                 mem-tag
13               nil)
14             (if (tag-dominate pc-tag mem-tag spec)
15               mem-tag
16             nil))))
17     nil))
```

Figure 5.6: Tag checking on LD instruction in ACL2

that they are valid tags. Then the code checks if the world readable bit is set and the memory location is data memory or stack memory. If this is the case, it returns the memory tag as the new tag, otherwise the function starts checking the copy bit of the memory's tag. If the copy bit is set, then the function checks whether or not the Owner field of the PC's tag dominates the Owner field of the memory location's tag. If the PC's Owner tag dominates the memory location's Owner tag, the function returns the memory's tag. Otherwise, the function checks whether or not the PC's tag dominates the memory location's tag, and returns the memory's tag or `NIL` respectively.

If the function returns `NIL`, then the tags failed the tag checking, and the execution of the `LD` instruction will be disabled as discussed in Section 5.4. If the instruction is allowed to execute, then the tag returned from the `ld-inst-tag` function will be the new tag of the register where the data is loaded.

5.3 Formal Model of the UI Tagging System

In the formal model, a state list is kept to record the state of the tagging system which includes state of the registers, memory, special registers and a security exception number (see Figure 5.7). While executing the instructions, the state list is changed and passed around to keep track of the state of the system. The list of registers (lines 1 to 9) contains triples that include the register number, the value stored in the registers and the tag associated with the register. Similarly the memory list (lines 10 to 15) contains triples that include the memory location, the data stored in the memory location and the security tag. Special registers' names, values and tags are stored in the special register list (lines 16 to 20). The security exception number (line 21), gives information about the whole system. A `0` indicates the system is running correctly without a security exception. Any other positive numbers indicate that there is a security exception generated. When a security exception is generated, the state list is updated with a security exception number that corresponds to a unique cause of the security exception. Therefore the problem can be easily identified when a security exception happens.

The registers and the model of sliding the register window in ACL2 is different from the actual SPARC architecture. In SPARC, the `SAVE` instruction decreases the `CWP`, while `RESTORE` increases it. However in the ACL2 tagging model, this is done in the opposite way; the `SAVE` instruction adds 1 to the `CWP` and the `RESTORE` reduces the `CWP` by 1.

Shown in Figure 5.8, the register list starts from register 0 and grows up. The initial state has 32 registers (register 0 to register 31). Register 0 to register 7 are `global` registers, registers 8 to 15 are `in` registers, `local` registers are registers 16 to 23 and `out` registers are registers 24 to 31. When sliding forward the register window, the `SAVE` instruction zeros these 16 registers and associates the PC's tag with them. Sliding the register window back will not cause these registers to be deleted. The special register list keeps track of the stack pointer, frame pointer, current window pointer, integer condition code and `y` register.

```
1  ((...
2    ((REG 16) 0 (LOW LOW (CPNOTSET RANDW STACKMEM WORLDNOTSET)))
3    ((REG 15) 0 (USER1 USER1 (CPNOTSET RONLY STACKMEM WORLDNOTSET)))
4    ((REG 14) 3000 (USER1 USER1 (CPNOTSET RANDW DATAMEM WORLDNOTSET)))
5    ...
6    ((REG 3) 111 (USER1 USER1 (CPNOTSET RANDW DATAMEM WORLDNOTSET)))
7    ((REG 2) 0 (LOW LOW (CPNOTSET RANDW STACKMEM WORLDNOTSET)))
8    ((REG 1) 2345 (USER USER (CPNOTSET RANDW DATAMEM WORLDNOTSET)))
9    ((REG 0) 0 (LOW LOW (CPNOTSET RANDW STACKMEM WORLDSET))))
10   (((MEM 33681632) 4 (LOW LOW (CPNOTSET RANDW DATAMEM WORLDNOTSET)))
11   ((MEM 33680800) 0 (LOW LOW (CPNOTSET RANDW DATAMEM WORLDNOTSET)))
12   ...
13   ((MEM 33680784) 1413558560 (LOW LOW (CPNOTSET RANDW DATAMEM
         WORLDNOTSET)))
14   ((MEM 33680792) 1413559072 (LOW LOW (CPNOTSET RANDW DATAMEM
         WORLDNOTSET)))
15   ((MEM 33680788) 1413558816 (LOW LOW (CPNOTSET RANDW DATAMEM
         WORLDNOTSET))))
16   (((SPEC RW) 0 (HIGH HIGH (CPSET RANDW DATAMEM WORLDNOTSET)))
17   ((SPEC FP) 3000 (LOW LOW (CPNOTSET RANDW DATAMEM WORLDNOTSET)))
18   ((SPEC ICC) 0 (LOW LOW (CPNOTSET RANDW DATAMEM WORLDNOTSET)))
19   ((SPEC Y) 0 (LOW LOW (CPNOTSET RANDW DATAMEM WORLDNOTSET)))
20   ((SPEC SP) 3000 (LOW LOW (CPNOTSET RANDW DATAMEM WORLDSET))))
21   0)
```

Figure 5.7: Sample state list in ACL2

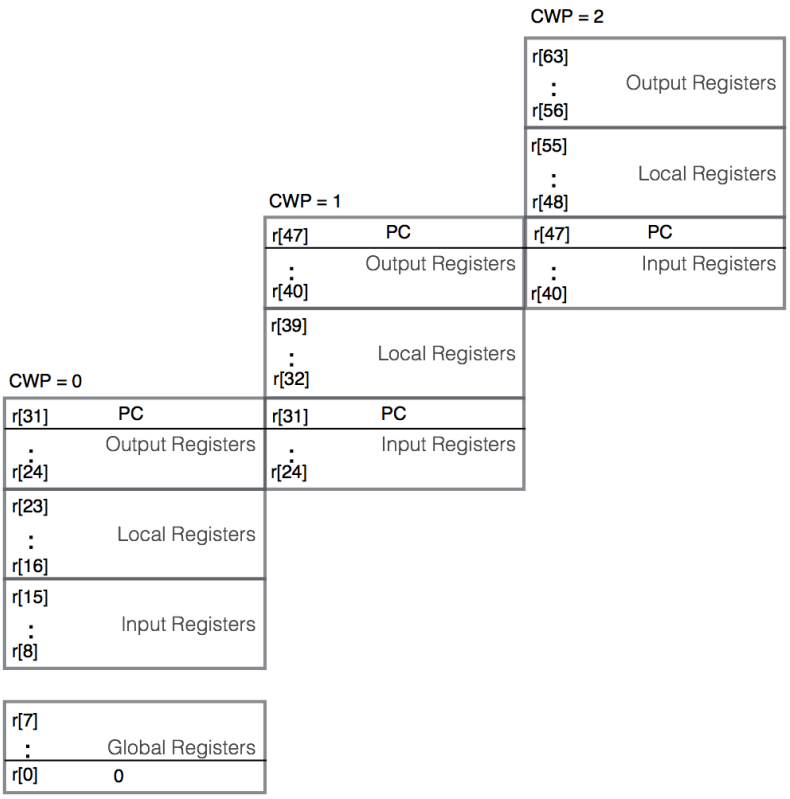

Figure 5.8: Register window and registers in ACL2

In the tagging model, all registers are represented by numbers starting from 0 up to 519. The following formula is used in the tagging model to set up the relationship between register names and register numbers.

- $RegisterNumber = (RW - 1) * 16 + BasicRegisterValue$

By using the value stored in `RW` (Register Window) register and the basic register value, the correct register number can be calculated easily. The basic register values are shown in Table 5.2. For example, if `RW` is 2, then the `%o7` register in that register window is register 47 (31+16). If `RW` is 3, then the `%l3` register is register 51 (19+32). Whichever register window the thread in, the global registers `%g0` to `%g7` are always register 0 to register 7, and the formula is not used for the global registers.

Table 5.2: Basic register value

register name	basic register value	register name	basic register value	register name	basic register value
%i0	8	%l0	16	%o0	24
%i1	9	%l1	17	%o1	25
%i2	10	%l2	18	%o2	26
%i3	11	%l3	19	%o3	27
%i4	12	%l4	20	%o4	28
%i5	13	%l5	21	%o5	29
%i6	14	%l6	22	%o6	30
%i7	15	%l7	23	%o7	31

Starting from the initial state, the system keeps track of the state list while executing instructions. For example when executing a `LD` instruction: `LD [%l1 + 8], %o0`, that passes the tag checking rules, the `%o0` register's value and tag will be changed to the memory location `[%l1 + 8]`'s value and tag. Similar to this, executing other instructions causes the system to check corresponding tagging rules and make specific changes on the state list as described in Section 5.4. If an instruction violates the tagging rules, a security exception will be generated by updating the security exception number in the state list with a positive number.

In the ACL2 model, a parameter `spec` is used to keep all the static values and tables. It consists of a list of instructions for testing purposes, the code table for comparing the relationship between labels, etc. Therefore when static information is needed, specific value or table can be retrieved from the `spec` easily.

The SPARC architecture has delayed control transfer instructions. This means that the processor fetches and executes the instruction that is located after the delayed control transfer instruction, before executing the transfer. The delayed control transfer instructions include the conditional `BRANCH` instructions, `CALL`, `RET/RETT`, `JMPL` (Jump&Link) and `Trap/Ticc`. Because of the existence of the delayed control transfer instructions, the system needs to identify these instructions and determine whether or not the next instruc-

```
(defun inst-class (inst-name)
  (cond ((equal inst-name 'LD) 1)
        ((equal inst-name 'ST) 2)
        ((equal inst-name 'ADD) 3)
        ((equal inst-name 'SUB) 4)
        ((equal inst-name 'MUL) 5)
        ((equal inst-name 'DIV) 6)
        ((equal inst-name 'JMPL) 7)
        ((equal inst-name 'SAVE) 8)
        ((equal inst-name 'CLR) 9)
        ((equal inst-name 'OR) 10)
          ...
        (t nil)))
```

Figure 5.9: Classification of instructions in ACL2

tion needs to be executed first or ignored. For the delayed control transfer instructions, such as `CALL`, `RET` and `JMPL`, the next instruction will always be fetched and executed before the current instruction. However, this rule is not applicable to the conditional branch instructions. For the conditional branch instructions, the next instruction will be fetched, but whether or not it is executed depends on the `BRANCH` instruction itself. Based on the integer condition code, if the `BRANCH` is taken, the next instruction will be executed, otherwise it will be skipped. A processor that provides speculative execution of branches will have to coordinate with the tag coprocessor to ensure precise exceptions.

As shown in Figure 5.9, the `inst-class` function goes through the instruction name list to find the correct classification number for instructions.

The `inst-execute` function is used to execute the correct instruction based on the instruction's classification number. The function uses another function `operand-one` to help find the data and tags from the correct locations. If a source operand is `%i4`, then the data is stored in register `%i4` and its tag should be gathered. The operand could be `[%fp-4]`, which means the data stored in `%fp` needs to be found first, and then add the offset (minus 4) to get the correct location of the operand. For example, `ld [%fp-4], %l2`, loads the value stored in `[%fp-4]` to register `%l2`. The function

```
1  (defun specification ()
2  (list (list 'TABLE (code-table))
3        (list 'MAX-MEM #x201ce00)
4        (list 'MIN-MEM #x2000000)
5        (list 'SCORE-LEVELS (score-level))
6        (list 'RTEMS-LEVELS (rtems-level))
7        (list 'INST-LIST (inst-lists4))
8        (list 'DEFAULT-TAG (default-tag))
9        (list 'DEFAULT-STACK-TAG (default-tag2))
10       (list 'DEFAULT-HIGH-TAG (default-high-tag))
11       (list 'DEFAULT-PC-CONTROL-LABEL (default-pc-control-label))))
```

Figure 5.10: Specification (spec) format

`operand-one` gets the value and tags of the 2 operands separately and passes them to the `inst-execute` function. The function extracts out the instruction opcode, LD, and calls the corresponding function for the `LD` instruction.

In Figure 5.11, part of the code of the `inst-execute` function is listed. The function gets one instruction with all the operands from the instruction list. Then it extracts the instruction name and at most three operands from the instruction. After getting the instruction classification number by using the `inst-class` function, the function starts comparing the classification number with various numbers to figure out which instruction it is. Then the code chooses the corresponding execution function to execute the instruction. Note that `inst-execute`, and many other functions include a final parameter `spec`. This parameter defines the static configuration (i.e., specification) of the system being modeled. This includes the `code-table`, SCORE levels, RTEMS levels (see Figure 5.10).

The `inst-execute` function controls the execution of all the instructions. If an instruction is not recognized, a security exception will be thrown by setting the security exception number of the state list to 44. The exception numbers are part of the model and are unique for each security exception to facilitate analysis.

```
1  (defun inst-execute (one-inst state-list spec)
2    (let ((inst-all (get-inst-list one-inst))
3          (inst-address (get-inst-addr one-inst)))
4      (let ((inst-name (get-inst-name inst-all))
5            (op1 (get-op1 inst-all))
6            (op2 (get-op2 inst-all))
7            (op3 (get-op3 inst-all)))
8        (let ((class-num (inst-class inst-name))
9              (reg-win (get-reg-value (search-special-register 'rw (third
                   state-list)))))
10         (let ((pc-list (search-register (get-register-num 'o7 reg-win)
                 (first state-list)))
11               (icc-list (search-special-register 'icc (third state-list))
                   ))
12       (cond
13         ((equal class-num 1)  ;ld
14          (ld-exe (operand-one op1 state-list spec) (operand-one op2
                 state-list spec) pc-list state-list spec))
15         ((equal class-num 2)  ;st
16          (st-exe (operand-one op1 state-list spec) (operand-one op2
                 state-list spec) pc-list state-list spec))
17         ((equal class-num 3)  ;add
18          (add-exe (operand-one op1 state-list spec) (operand-one op2
                 state-list spec)
19            (operand-one op3 state-list spec) pc-list state-list spec))
20            ...
21         ((equal class-num 7)  ;jmpl
22          (jmpl-exe inst-address (operand-one op1 state-list spec) (
                 operand-one op2 state-list spec)
23            pc-list state-list spec))
24         ((equal class-num 8)  ;save
25          (save-exe (operand-one op2 state-list spec) (operand-one op1
                 state-list spec)
26            pc-list state-list spec))
27           ...
28     (t (set-multi-exception state-list 44)))))))))
```

Figure 5.11: Function for instruction execution in ACL2

```
(defun st-exe (reg-op1-list mem-op2-list pc-list state-list spec)
    (let ((mem-tag (get-list-tag mem-op2-list))
          (mem-addr (get-list-addr(car mem-op2-list)))
          (pc-tag (get-list-tag pc-list))
          (reg-value (get-list-value reg-op1-list))
          (reg-tag (get-list-tag reg-op1-list)))
      (if (st-inst-tag mem-tag reg-tag pc-tag spec)
    (update-mem-state-list (update-memory-list mem-addr reg-value
      (st-inst-tag mem-tag reg-tag pc-tag spec)
              (second (check-exist mem-op2-list state-list spec)))
         (check-exist reg-op1-list state-list spec))
    (set-multi-exception state-list 23))))
```

Figure 5.12: Instruction execution of ST instruction

5.4 Execution and Tag Propagation in ACL2

There are 42 individual functions in ACL2 that have been developed for different instructions. These include the `ld`, `st`, `call`, `jump`, `ret`, `save`, `restore`, `branch` instructions, `shifting`, `logic` and `arithmetic` instructions and the "cc" version of these instructions (the "cc" version of these instructions modifies the integer condition code in addition to doing what they should do) and other instructions. In these functions, the tag checking and propagation functions (see Section 5.2) are used to check whether the instructions are allowed to be executed or not. The functions then either update the state list and corresponding tags, or generate a security exception. This section describes the model of each of these classes of instruction.

5.4.1 Execution for LD and ST instructions

The `ST` instruction stores data from a register to a memory location. If the tag checking for `ST` instruction passes, then the corresponding memory location in the state list needs to be updated with the value and tag of the register. Figure 5.12 lists the ACL2 code for executing the `ST` instruction.

```
1 (defun subcc-exe (op1-lst op2-lst op3-lst pc-lst state-list spec)
2   (let ((op1-tag (get-list-tag op1-lst))
3         (pc-tag (get-list-tag pc-lst))
4         (op2-tag (get-list-tag op2-lst))
5         (op1-value (get-list-value op1-lst))
6         (op2-value (get-list-value op2-lst))
7         (op3-num (get-list-addr (car op3-lst))))
8     (if (arith-inst-tag op1-tag op2-tag pc-tag spec)
9      (update-statelist-spec2 'icc (- op1-value op2-value)
10       (arith-inst-tag op1-tag op2-tag pc-tag spec)
11       (update-reg-state-list
12        (update-register-list op3-num
13                  (- op1-value op2-value)
14                  (arith-inst-tag op1-tag op2-tag pc-tag spec)
15                  (first state-list))
16         state-list))
17      (set-multi-exception state-list 2422))))
```

Figure 5.13: Instruction execution of SUBcc instruction

The `st-exe` function checks the tagging rules of `ST` instruction using the tags of memory, register and PC. If the `ST` instruction is allowed to be executed according to the tagging rule, the function updates the memory location in the state list with the value of register that is being stored. The tag of the memory location will be replaced by the tag returned from the `st-inst-tag` function. After doing this, the state list will be updated. However if the `ST` tag checking rules block the execution of the `ST` instruction, then a security exception will be generated.

The idea of the `LD` instruction is very similar to the `ST` instruction. The tag checking function for `LD` is used to make sure the LD is allowed to be executed. If the `LD` is allowed, then the destination register in the state list will be updated with the value and tag of the memory location where the data is loaded. If it is not allowed, a security exception will be generated.

5.4.2 Execution of arithmetic, logic and shifting instructions

When running an arithmetic, logic, or shifting instruction, the result's tag is calculated by the `arith-inst-tag` function. The instruction execution function updates the correct register in the state list. For "cc" versions, the instructions also modify the value and tag of the `icc` in the state list.

To illustrate the execution of this group of instructions, the code for executing the `SUBcc` instruction is shown in Figure 5.13.

The `SUBcc` performs the subtract function, however the difference between a `SUB` instruction and `SUBcc` instruction is that `SUBcc` also makes a change to the integer condition code based on the subtraction (see line 9).

In the `subcc-exe` function, all the information that is needed for the tag checking and subtraction function, such as tags and values, are extracted first. Then the tag checking function, `arith-inst-tag`, is used to check whether or not the `SUBcc` is allowed be executed. If it is allowed, the code updates the destination register's value and tag. Furthermore, the `icc` in the state list will be updated, and the whole updated state list will be returned. However if the `arith-inst-tag` function returns `NIL`, then the `subcc-exe` function will generate an exception, and leave the rest of the state unmodified.

For the other instructions in this class, the execution functions are very similar to the `SUBcc-exe` function. All of the instructions in this group use the same tag checking function (`arith-inst-tag`). Therefore only the substraction operation needs to be changed to other operations such as addition, multiply or logical operations.

5.4.3 Execution of CALL related instructions

In the SPARC architecture, the `CALL` instruction copies the contents of the PC, which is the address of the `CALL` instruction itself, into the `%o7` register. The `call-exe` function implemented in ACL2 does more than this, since the tags need to be checked and

```
1 (defun call-exe (address-list inst-addr pc-list state-list spec)
2    (let ((addr-tag (get-list-tag address-list))
3          (pc-tag (get-list-tag pc-list))
4          (pc-num (get-list-value (first pc-list))))
5      (if (call-inst-tag addr-tag pc-tag spec)
6        (update-reg-state-list (update-register-tag (+ pc-num 16) (
             call-inst-tag addr-tag pc-tag spec)
7        (update-register-list pc-num inst-addr (set-cpbit-tag pc-tag
             spec)
8          (first (update-16-registers (+ pc-num 16) state-list
9                (call-inst-tag addr-tag pc-tag spec) spec)))) state-list
                  )
10     (set-multi-exception state-list 255))))
```

Figure 5.14: Instruction execution of CALL instruction

propagated. The code of `call-exe` function is shown in Figure 5.14.

Similar to the execution functions introduced before, the `call-exe` gets parameters such as the PC's tag, state list, specification, the calling address and the current running instruction's address. The code first uses tag checking to determine whether or not the function call is allowed. If it is allowed, it continues by changing the state list, otherwise it raises a security exception. For changing the state list, the code zeros 16 registers and tags them with the correct PC's tag, which is calculated by the `call-inst-tag` function. In addition to these updates, code sets the copy bit of the PC's tag and copies the current address of the `CALL` instruction and PC's tag to the `%o7` register and its tag.

5.4.4 Execution of BRANCH instructions

The `BRANCH` instructions execute a branch based on the `icc` value. Different `BRANCH` instructions require different checks on the `icc` value. However the idea of the execution of the different `BRANCH` instructions is almost identical. Take `be-exe` as an example (Figure 5.15).

The `be-exe` executes the `BE` (branch on equal) instruction. The function first uses

```
1 (defun be-exe (cur-addr op1-addr-list pc-list icc-list state-list spec)
2   (let ((op1-tag (get-list-tag op1-addr-list))
3         (icc-tag (get-list-tag icc-list))
4         (pc-tag (get-list-tag pc-list)))
5     (if (or (branch-inst-tag op1-tag pc-tag icc-tag spec)
6             (check-same-function cur-addr (car (second op1-addr-list))
                 spec))
7         (if (equal 0 (get-list-value (search-special-register 'icc (
               third state-list))))
8             state-list
9           (set-multi-exception state-list -1))
10        (set-multi-exception state-list 301))))
```

Figure 5.15: Instruction execution of BE instruction

the `branch-inst-tag` function to verify that the branch is allowed based on the tag checking rules. If it is not allowed, it generates an exception. Because the `BE` instruction executes a branch on equal, the `icc` value is compared with 0 to determine whether or not to execute the branch. If the branch is going to be taken, it simply returns the `state-list`. However if the branch is not going to be taken, the security exception value will be changed to -1. In the system, if the security exception value is greater than 0, the instruction execution will be stopped and the state list will be returned with the security exception number in it to indicate the violation. The -1 is a security exception number that is special for branch instructions. The -1 is used to indicate that the `BRANCH` instruction will not be executed because of the branch condition. Therefore the system will not be terminated and the -1 will be reset to 0 later.

The execution functions for other `BRANCH` instructions, such as `BNE` (branch on not equal), `BG` (branch on greater than), `BLE` (branch on less than or equal to), are similar to the code shown in Figure 5.15. The difference will be the check on the branch conditions.

```
1  (defun save-exe (num-op2-list sp-op3-list pc-list state-list spec)
2    (let ((reg-win (get-list-value (search-special-register 'rw (third
       state-list)))))
3      (if (add-exe-spec sp-op3-list num-op2-list sp-op3-list pc-list
          state-list spec)
4          (update-statelist-regular2 (get-register-num 'o6 (+ reg-win 1))
5          (+ (get-list-value num-op2-list) (get-list-value sp-op3-list))
6          (get-list-tag pc-list)
7          (update-statelist-spec 'fp sp-op3-list
8          (increase-cwp reg-win
9          (get-list-tag pc-list)
10         (add-exe-spec sp-op3-list num-op2-list sp-op3-list pc-list
             state-list spec) spec)))
11           (set-multi-exception state-list 257))))
```

Figure 5.16: Instruction execution of SAVE instruction

5.4.5 Execution of SAVE and RESTORE instructions

In the SPARC architecture, the `SAVE` instruction decrements the `CWP` by 1 and `RESTORE` instruction increments it. Because the register numbers in the ACL2 model start from 0 and grow up, the modified `SAVE` instruction increase the `CWP` by 1 while the `RESTORE` instruction decreases the `CWP` by 1. The `SAVE` instruction is usually in a format such as `SAVE %sp, -128, %sp`. It takes the value in `%sp` for the current register window, adds it to the second operand, and then writes the result into the `%sp` in the `CWP+1` register window. The number indicates the size of the stack for the new function. Figure 5.16 lists the code for instruction execution of the `SAVE` instruction.

The `save-exe` function first ensures that the instruction is allowed to be executed based on the tag checking. If not, it sets the security exception number of the state list to 257. If the instruction is allowed, the function increases the `CWP` and returns a new state list by using the `increase-cwp` function. After that, the `save-exe` function writes the new `%sp` value (adds the old `%sp` with the number) to the `%sp` of the new register window. Because the `%sp` is always stored in `%o6` register, the new register number is calculated by the new register window number (the old register number plus 1) and the

```
1  (list
2    (list 'FUNCTION #x2001284 #x20013a8
3    (list 'USER1 'USER1 (list 'CPNOTSET 'RONLY 'EXEENTRY 'WORLDNOTSET))
4           ;"<Init>:"
5    (list
6     '(#x2001284 ( save (sp) (-128) (sp)) )
7     '(#x2001288 ( sethi (#x201e400) (g1)) )
8     '(#x200128c ( ld (IND g1 #x340) (g1))  )
9     '(#x2001290 ( sethi (#x201cc00) (o1)) )
10    '(#x2001294 ( ld (IND g1 #xc) (o0)) )
11    '(#x2001294 ( st (10) (IND o0 594)) )
12    '(#x2001294 ( ld (IND g0 594) (o3)) )
13    '(#x2001294 ( add (g1) (o3) (o1)) )
14    '(#x2001298 ( or (o1) (#x210) (o1)) )
15    '(#x200129c ( sethi (#x201cc00) (o2)) )
16    '(#x20012a0 ( call (#x20119b8) ()) )
17    '(#x20012a4 ( or (o2) (#x218) (o2))  )
18    '(#x20012a8 ( mov (#x7c4) (g1)) )
19    '(#x20012ac ( st (g1) (IND fp -28)) )
20        ...
21       )))
```

Figure 5.17: Sample SPARC instructions for RTEMS code

`%o6`. Then the new state list will be returned.

The `RESTORE` instruction decreases `CWP` to slide the register window back.

5.4.6 Test the tagging system in ACL2

After setting up the tagging model in ACL2, real SPARC assembly instructions are used to test the system. We wrote our test code in 'C', integrated it into RTEMS and ran the compiler. Then by using `sparc-rtems4.10-objdump` command, we generated the assembly for the object files. This assembly was then converted to a LISP notation. Figure 5.17 displays the instructions for a `Init` function in RTEMS. The instructions and other information such as tag, starting address and ending address, are put together in a list in ACL2 to make it usable by the tagging model.

The code in Figure 5.17 shows a list which includes (1) a `FUNCTION` label, (2) the

starting address of the `Init` function (#x2001284), (3) an ending address of the function (#x2001318), (4) the tag of the `Init` function, `(USER1 USER1 (CPNOTSET RONLY EXEENTRY WORLDNOTSET))` and (5) the instructions of the function with their address (from line 5 to 20).

Our tool generates a separate list for each function in RTEMS. In our testing, many functions have been put into lists like the list shown in Figure 5.17. The ACL2 model of the tagging system is able to start running from the first instruction in a function and keep running through several functions (by `CALL` and `RET`) without security exceptions being generated. For testing purposes, the tested functions cover functions from different RTEMS levels (MANAGER and SCORE), the `LOW` level C library functions, and user functions.

As proof of concept a small portion of RTEMS code has been tested. The tests were made to cover as many of the tagging possibilities as possible.

5.5 Verifying the Security Lattice

As mentioned in Section 3.2, the security classes in the lattice are arranged in a partial ordering (using the operator $\leq$). For the lattice model of information flow, there exists a lattice with a finite set of security classes and a partial ordering between the security classes. The symbol $\leq$ denotes the partial ordering relation between two security classes. For example, $A \leq B$ means class A is at a lower or equal level than class B. In a lattice, the $\leq$ operator is reflexive, transitive, and antisymmetric:

reflexive: $A \leq A$

transitive: if $A \leq B$ and $B \leq C$, then $A \leq C$

antisymmetric: if $A \leq B$ and $B \leq A$, then $A = B$

The ACL2 code for proving these three properties is shown in Figure 5.18, Figure 5.19 and Figure 5.20. The other basic functions which are used for proving these theorems,

such as `label-dominate` and `security-class-check`, are not shown. Also, additional lemmas needed by the proof engine are not shown.

- Figure 5.18 shows the code for proving the reflexivity theorem. Lines 3 checks the format of the label1 (label1 means the Owner field or the Code-space field of a tag) to make sure it is correct and then on line 4, proves that the label1 can dominate itself.

- Similar to the proof of label-reflexivity theorem, lines 6 to 10 prove the reflexivity of security class (security class is the combination of Owner field and Code-space field of a tag). Line 7 checks the format of the sclass1, and implies that sclass1 should dominate itself on line 8.

- In Figure 5.19, the proofs of transitive theorem for labels and security class are displayed. On line 5 to 7, it checks whether the labels are in valid format. The code checks that label1 dominates label2, label2 dominates label3 and then implies that label1 dominates label3 (line 10).

- On lines 13 to 24, the transitive theorem for security classes is proved. If sclass1 dominates sclass2 (line 20) and sclass2 dominates sclass3 (line 21), then implies sclass1 dominates sclass3 (line 22).

- In Figure 5.20, lines 2 to 10 are used to prove the anti-symmetry theorem for labels. If label1 dominates label2 (line 5), label2 dominates label (line 6), and if all these labels' format are valid, then implies that label1 and label2 are equal (line 7).

- Lines 12 to 21 prove that if sclass1 dominates sclass2 (line 15), sclass2 dominates sclass1 (line 16), then sclass1 and sclass2 are equal.

After proving these three basic theorems, whether the least upper bound (LUB) and greatest lower bound (GLB) operations can be implied from the lattice are proven. The definitions of LUB and GLB [8] are:

```
;------ REFLEXIVITY THEOREM ------
(defthm label-reflexivity-thm
  (implies (label-format-check label1 spec)
     (label-dominate label1 label1 spec)))

(defthm security-class-reflexivity-thm
  (implies (security-class-check sclass1 spec)
     (security-class-dominate sclass1 sclass1 spec))
  :hints  (("Goal"
      :in-theory (disable label-format-check label-dominate))))
```

Figure 5.18: Prove reflexivity theorem in ACL2

```
;------ TRANSITIVE THEOREM ------
(defthm label-transitive
  (let ((code-table (second (assoc 'TABLE spec))))
    (implies
      (and (member-equal label1 (auto-list code-table))
           (member-equal label2 (auto-list code-table))
           (member-equal label3 (auto-list code-table)))
      (implies (and (label-dominate label1 label2 spec)
                    (label-dominate label2 label3 spec))
     (label-dominate label1 label3 spec))))
  :hints (("Goal"  :in-theory (disable label-dominate))))

(defthm security-class-transitive-thm
 (implies (and (sclass-in-table sclass1 spec)
               (sclass-in-table sclass2 spec)
               (sclass-in-table sclass3 spec))
   (implies (and (security-class-check sclass1 spec)
                 (security-class-check sclass2 spec)
                 (security-class-check sclass3 spec))
   (implies (and (security-class-dominate sclass1 sclass2 spec)
                 (security-class-dominate sclass2 sclass3 spec))
         (security-class-dominate sclass1 sclass3 spec))))
  :hints (("Goal"
   :in-theory (disable label-format-check label-dominate dominates-list
       dominates-subsetp-thm))))
```

Figure 5.19: Prove transitive theorem in ACL2

```
1  ;------ ANTI-SYMMETRY THEOREM ------
2  (defthm label-anti-symmetry-thm
3    (implies (and (label-format-check label1 spec)
4                  (label-format-check label2 spec)
5                  (label-dominate label1 label2 spec)
6                  (label-dominate label2 label1 spec))
7             (equal label1 label2))
8    :rule-classes nil
9  :hints (("Goal" :in-theory (disable label-format-check label-dominate)
10         :use (LABEL-FORMAT-CHECK-AXIOM1 anti-symmetry-func-lab))))
11
12 (defthm security-class-anti-symmetry-thm
13   (implies (and (security-class-check sclass1 spec)
14                 (security-class-check sclass2 spec))
15    (implies (and (security-class-dominate sclass1 sclass2 spec)
16                  (security-class-dominate sclass2 sclass1 spec))
17      (security-class-eq sclass1 sclass2)))
18   :hints  (("Goal"
19       :use ((:instance label-anti-symmetry-thm  (label1 (car sclass1))
           (label2 (car sclass2)))
20             (:instance label-anti-symmetry-thm (label1 (cadr sclass1))
                (label2 (cadr sclass2))))
21       :in-theory (disable label-format-check label-dominate))))
```

Figure 5.20: Prove anti-symmetry theorem in ACL2

- **Least upper bound**: for each pair of classes A and B in the lattice, there exists a unique class $C = A \oplus B$ such that:

  $A \leq C$ and $B \leq C$, and

  $A \leq D$ and $B \leq D$ implies $C \leq D$ for all D in the lattice.

- **Greatest lower bound**: for each pair of classes A and B in the lattice, there exists a unique class $E = A \otimes B$ such that:

  $E \leq A$ and $E \leq B$, and

  $D \leq A$ and $D \leq B$ implies $D \leq E$ for all D in the lattice.

Figure 5.21 shows the code that was used to prove the `LUB` and `GLB` properties of the security lattice. The labels' format are checked first to ensure the correctness of the labels. Then `label3` dominates `label1` and `label3` dominates `label2`, implies that `label3` dominates the `LUB` of `label1` and `label2`. The function `(label-lub label1 label2)` on line 7 checks the least upper bound rules for `label1` and `label2` and returns the result class.

On lines 8 to 15, the greatest upper bound relation is proven. If all the labels are in correct format, and if they all satisfy the relations that `label1` dominates `label3` and `label2` dominate `label3`, then implies that `label3` is dominated by the GLB of `label1` and `label2`. Similar to the `label-lub` function, `label-glb` checks the GLB rules and returns the label which is the greatest upper bound of `label1` and `label2`.

After proving the reflexive, transitive, antisymmetric, LUB and GLB properties, the proofs can be put together to prove that the model satisfies a lattice. According to Denning [8], a flow policy is a lattice if the classes are partially ordered and there exist least upper bound and greatest lower bound relations of the classes. Therefore, to prove that the system is a lattice, all of the properties: reflexive, transitive, anti-symmetric, LUB and GLB have to be proved to be satisfied by the system. Figure 5.22 presents the code for proving that the model is a lattice.

```
(defthm lub-thm
    (implies (and (label-format-check label1)
                  (label-format-check label2)
                  (label-format-check label3)
                  (label-dominate label3 label1)
                  (label-dominate label3 label2))
    (label-dominate label3 (label-lub label1 label2))))

(defthm glb-thm
    (implies (and (label-format-check label1)
                  (label-format-check label2)
                  (label-format-check label3)
                  (label-dominate label1 label3)
                  (label-dominate label2 label3))
    (label-dominate (label-glb label1 label2) label3)))
```

Figure 5.21: Prove LUB and GLB theorems in ACL2

```
(defthm is-lattice
  (implies (and (label-format-check label1)
                (label-format-check label2)
                (label-format-check label3))
       (and (transitive-func-lab label1 label2 label3)
            (reflexivity-func-lab label1)
            (anti-symmetry-func-lab label1 label2)
            (lub-func-lab label1 label2 label3)
            (glb-func-lab label1 label2 label3)))
  :hints  (("Goal"
           :in-theory (disable label-format-check label-dominate
              transitive-func-lab
                    reflexivity-func-lab anti-symmetry-func-lab
                    lub-func-lab glb-func-lab))))
```

Figure 5.22: Prove the flow policy is a lattice in ACL2

```
1  (defthm ld-exe-exception-thm
2    (let* ((mem-tag (third mem-op1-list))
3           (pc-tag (third pc-list))
4           (mem-tag-owner (get-owner-label mem-tag))
5           (mem-tag-control (get-control-label mem-tag))
6           (pc-tag-owner (get-owner-label pc-tag)))
7      (implies
8         (and
9           (state-validatep state-list spec)
10          (valid-op-list mem-op1-list)
11          (not (has-security-exception state-list)))
12        (equal
13          (has-security-exception
14             (ld-exe mem-op1-list reg-op2-list pc-list state-list spec))
15          (not (ld-inst-tag mem-tag pc-tag spec))))))
```

Figure 5.23: Prove the security policy for LD instruction

5.6 Verifying the Security Policies for Instructions

As described in Section 5.2, the tag checking rules have been modeled for each class of the instructions. Other theorems can be proven for the security policies of the instruction executions.

The code shown in Figure 5.23 proves that the execution of the `LD` instruction will throw an exception if and only if the `ld-inst-tag` function returns `NIL`. The code first ensures the initial state list is a valid state, the operands are valid and there is no security exception generated. Then after execution of the `LD` instruction, there will be an exception generated if and only if the tag checking function for `LD` fails, in other words, the `ld-inst-tag` function returns `NIL`.

Section 5.2.4 presents the detailed tagging rules for `LD` instruction. The lines 12-15 in Figure 5.23 can be rewritten with lines in Figure 5.24. The code in Figure 5.24 replaces the `ld-inst-tag` with the detailed tag checking rules for `LD`.

There are similar theorems for the other instruction classes, such as `BRANCH` instructions, `ST` instruction and etc. These theorems provide the formal proofs that the model

```
1  (equal
2    (has-security-exception
3       (ld-exe mem-op1-list reg-op2-list pc-list state-list spec))
4    (or (not (tag-format-check mem-tag spec))
5        (not (tag-format-check pc-tag spec))
6        (and (not (and (equal 'WORLDSET (get-control-world
             mem-tag-control))
7                       (data-stack-mem mem-tag-control)))
8             (or (and (equal 'CPSET (get-control-copybit mem-tag-control)
                 )
9                      (not (label-dominate pc-tag-owner mem-tag-owner
                          spec)))
10                (and (not (equal 'CPSET (get-control-copybit
                       mem-tag-control)))
11                     (not (tag-dominate pc-tag mem-tag spec)))))))
```

Figure 5.24: Prove the security policy for LD instruction 2

satisfies the security policy specified in this dissertation and defines in the `xx-inst-tag` formulas.

5.7 Summary

This chapter presents the process of developing an execution model of the UI tagging system. A tag can be represented in a list in ACL2. There is a state list in the system which consists of all the registers, memory, special registers and a security exception number. Tag checking functions for different groups of instructions are written to check tags to figure out whether the specific instruction is allowed to be executed or not. If it is permitted, instruction execution functions are used to update the specific lists in the state list. If it is not permitted, the security exception number will be changed to a positive number which indicates a invalid instruction execution.

After setting up the execution model, instructions from RTEMS modules have been extracted and executed in the model. ACL2 starts the model from an initial state list, and then execute all of the instructions. The state list keeps track of the changes in the

system by updating the corresponding lists in the state list when executing an instruction.

To prove that the information flow in the model satisfies a lattice, the theorems of reflexive, transitive, antisymmetric, LUB and GLB have been proven to be satisfied by the system. In addition we have proven that the model of execution satisfies the security policies specified in the tag check functions. These proofs rely on the lattice nature of the system to ensure correctness.

Chapter 6: Conclusion and Future Work

This chapter concludes the dissertation work and proposes future work that could be conducted for the UITags project.

6.1 Conclusion

In recent years, security tagging schemes have been seen as promising mechanisms to enhance the security of computer systems. As a result, a number of security tagging schemes and corresponding security tagged architectures have been proposed to address current security problems. However, the majority of them are designed to prevent classes of runtime attacks on the system and not as a fundamental enhancement to the operation of the microprocessor.

The goal of this project was to design a new security tagging scheme that enhances access control through least privilege while enabling good system performance. As a first step towards this goal, this dissertation introduced a design of the UI tagging scheme that was focused on securing RTEMS, a single user, multi-thread model runtime executive. The partner team from Cornell University was responsible for developing the security tagged microprocessor which could support the tagging scheme.

RTEMS was chosen as the target system of this project because it is effectively a zero-kernel operating system that was in need of security enhancements. An evaluation of the RTEMS architecture provided a better understanding of what tags can and cannot do. According to the security enhancements that were needed, the tag was designed with three fields: Owner field, Code-space field, and Control-bits field. By using these tags, the system is able to separate RTEMS (i.e., code and data) and user (i.e., code and data), divide RTEMS code into different levels, prevent users from using critical system functions, and protect return values.

Since most of the RTEMS code is written in the C programming language, security

rules have been developed to control information flow within RTEMS with the support of tags. These rules cover all of the major C-language constructs, such as assignment statement, `if` statement, and `while` statement. A presentation of a security lattice is introduced in this dissertation to specify the authorized flow of information. The combination of Owner field and Code-space field tags represents the security class of the tagged data. With the security classes, lattice, and security rules, the information flow can be controlled in RTEMS.

After presentation of a C-language tagging scheme, tagging rules have been refined to support assembly language level tagging. Specifically, the tagging scheme was developed for the Leon3 processor, which is based on the SPARC architecture. SPARC instructions have been divided into logical groups and specific refined security rules were presented. These refined rules took into consideration the use of registers, register windows, memory stack, program counter, and differences between code and data in memory.

Because the hardware was not available for this project, a simulator for the SPARC architecture was enhanced to support the UI tagging scheme by the UI tagging research team. Hooks and additional instructions were successfully added in SIS for the UI tagging scheme.

Modifications of RTEMS source code have also been made to support the tagging scheme. This includes addition of a tag manager which provides tag checking and propagation that can not be done in hardware. Some tagging functions have been inserted into specific RTEMS functions to satisfy special requirements of the tagging scheme, such as setting the copy bit of the ID's tag that is going to be returned from some RTEMS functions. RTEMS has also been expanded from a single user multi-threaded model of execution to a multiuser system. RTEMS now supports the concepts of non-privileged user and superuser, where the superuser has the authority to create, delete, and control non-privileged users. A user manager has been added into RTEMS to support the tagging scheme for multiple users.

In total, 76 test cases (with more than 2000 subcases) were manually developed to test the design of the UI tagging scheme and the correctness of the SIS simulator that has modified interpretation of instructions deployed in it. These test cases cover many possibilities of tag checking and propagation, including all combinations of relationships between the PC's tag and the variable tags, memory types, and control bits.

Lastly, a formal model of the security policy enforced by the UI tagging scheme has been developed in ACL2. With the UI tagging scheme applied, the system state is monitored while executing different instructions. This model is used to prove the correctness of the tagging implementation as well as validate that the implemented system can support higher-level system security properties.

In summary, the work presented in this dissertation proposes a UI tagging scheme which protects the system and user software from security attacks and invalid information access. The tagging scheme has been partially implemented for RTEMS. Test cases were generated to ensure the tagging rules and the modified interpretation of instructions are correct. A formal model of the security policy enforced by the UI tagging system has been developed and security related lemmas and theorems have been proven.

6.2 Future Work

This section summarizes the future work that could be conducted as an extension of the security tagging project.

Fully tag RTEMS code to enable RTEMS to always use tagging. Currently, all of the test cases work by enabling and disabling tagging as needed. Therefore only some of the RTEMS code is tagged. For future work, all of the RTEMS functions have to be tagged, including libraries. The goal is to have a tagged RTEMS running from the beginning of system initialization. In addition to the RTEMS functions, global variables need to be tagged with care. This is especially true when RTEMS is modified to a

multiuser system. Some global variables can be shared among different users while other global variables cannot, due to the possibility that they may affect other users or the system. Therefore these global variables need to be made usable only by a specific user manager.

In addition, the C library functions need to be considered and properly tagged. Because of time restrictions, not all of the C library functions that are used by RTEMS can be tagged manually. Therefore software tools will have to be used to generate a list of the C library functions that are used in the RTEMS benchmarks. These functions will be given special tags. By doing this, additional checks will be applied when using these functions, to prevent malicious usage of the important functions.

The process of tagging the complete RTEMS may be time consuming, because RTEMS requires a fresh build and installation even with a small change of the code, which takes around 20 minutes.

Reduce the overhead of the tagging system. Since almost every instruction execution requires tag checking of the source or destination operands' tag, it increases the overhead of the system. For example, normally, the `LD` instruction loads values from memory space to registers, but in the UI tagging scheme, the `LD` instruction has to check the value's tag and store it as the register's tag additionally. To minimize the overhead, tags can be cached as many as possible to speed up the tag checking operations. Another way that could reduce the overhead of the tagging system is to use a tag compression scheme, and data and code spatial locality information to reduce the overhead.

Add security rules to floating point instructions. Among SPARC instructions, tagging rules are only given to the integer instructions, and not to the floating point instructions. Currently, all of the floating point instructions are left without analysis and corresponding tagging rules. Future efforts could be made on the floating point instructions and any other instructions that have not been examined.

Persistent tags and user-defined tagging rules. The current tagging model focuses on the protection of code and data from unauthorized access and modification. It is a first step toward enhanced security tagging, such as user-defined tags. The support of user-defined tags may require the support of persistent tags (tags that are maintained between system resets), that may also be used for multi-processor (or multi-core) execution models and storage of objects in file systems or other permanent storage.

In addition, the UI tagging scheme only supports fixed tagging rules. In the future, it would be desirable to have user-defined tagging rules. A second manager could be implemented to support user-defined tagging rules. However, to be able to support user-defined tagging rules, the system should provide not only the interface for users, but protections for the system in case a user tries to do something that could violate the security of the system.

Port sample applications to evaluate the performance of the tagging scheme. In addition to the test cases, some sample applications could be ported to RTEMS. These benchmark applications could help evaluate the performance of the UI tagging scheme. This may require additional changes in RTEMS code. For example, the portions of RTEMS and C libraries that are needed by the benchmark applications would need to be tagged. What's more, RTEMS is only compatible with a few applications and none of them are usable for evaluation purposes. Additional efforts are needed to port applications to RTEMS. Based on evaluation results, new methods for improving performance, may be proposed.

Bibliography

[1] J. Alves-Foss, J. Song, S. Steiner, L. Kerr, and A. S. Amack, "Evaluating the use of security tags in security policy enforcement mechanisms," *Accepted by 48th Hawaii International Conference on System Sciences (HICSS)*, Jan 2015.

[2] J. Alves-Foss, J. Song, S. Steiner, and S. Zakeri, "A new operating system for security tagged architecture hardware in support of multiple independent levels of security (MILS) compliant system: AFRL project final report," U. of Idaho Center for Secure and Dependable Systems, Tech. Rep. AFRL-RI-RS-TR-2014-088, 2014.

[3] D. Bell and L. LaPadula, "Secure computer system unified exposition and multics interpretation," MITRE Corp., Bedford, MA, Tech. Rep. MTR-2997, Jul. 1975.

[4] K. J. Biba, "Integrity considerations for secure computer systems," The Mitre Corporation, MTR-3153, Rev. 1, 1977.

[5] *The Operational Characteristics of the Processors for the Burroughs B5000*, Revision a, 5000-21005 ed., Burroughs Corporation, Detroit 32, Michigan, 1962. [Online]. Available: http://www.bitsavers.org/pdf/burroughs/B5000_5500_5700/5000-21005_B5000_operChar.pdf

[6] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proceedings of the 34th annual international symposium on Computer architecture*, vol. 35, May 2007, pp. 482–493.

[7] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, ser. DSN '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12.

[8] D. E. R. Denning, *Cryptography and Data Security*. Reading, Massachusetts: Addison-Wesley, 1982.

[9] *SPARC instruction set simulator manual*, version 3.0.5 ed., European Space Research and Technology Center (ESTEC), August 2006.

[10] J. S. Fenton, "Memoryless subsystems," *The computer journal*, vol. 17, no. 2, 1974.

[11] E. A. Feustel, "On the advantages of tagged architecture," *Transactions on Computers*, vol. C-22, no. 7, pp. 644–656, July 1973.

[12] *Debugging with GDB: the GNU Source-Level Debugger for GDB*, version 7.8.50 ed., Free Software Foundation, Inc., 2014. [Online]. Available: http://www.gnu.org/software/gdb/

[13] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks*. Estoril, Lisbon, Portugal: IEEE, 2009, pp. 105–114.

[14] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[15] *RTEMS C User's Guide*, edition 4.10.1, for rtems 4.10.1 ed., On-Line Applications Research Corporation, July 2011.

[16] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006)*. IEEE Computer Society, 2006, pp. 135–148.

[17] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 19, pp. 1278–1308, Sep. 1975.

[18] R. Shioya, D. Kim, K. Horio, M. Goshima, and S. Sakai, "Low-overhead architecture for security tag," in *Proceedings of the 15th IEEE Pacific Rim International*

Symposium on Dependable Computing. Shanghai, China: IEEE Computer Society, 2009, pp. 135–142.

[19] A. Shriraman and S. Dwarkadas, "Sentry: Light-weight auxiliary memory access control," in *Proceedings of the 37th International Symposium on Computer Architecture (37th ISCA'10)*. Saint-Malo, France: ACM SIGARCH, Jun. 2010, pp. 407–418.

[20] H. Shrobe, A. DeHon, and T. Knight, "Trust-management, intrusion tolerance, accountability, and reconstitution architecture (TIARA)," AFRL Technical Report AFRL-RI-RS-TR-2009-271, Tech. Rep., December 2009.

[21] J. Song and J. Alves-Foss, "Security tagging for a zero-kernel operating system," in *Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS)*, Wailea, HI, USA, Jan. 2013, pp. 5049–5058.

[22] J. Song, "Development and evaluation of a security tagging scheme for a real-time zero operating system kernel," Master Thesis, University of Idaho, May 2012.

[23] J. Song and J. Alves-Foss, "Hardware security tags for enhanced operating system security," *Issues in Information Systems*, vol. 14, no. 1, pp. 61–71, 2013.

[24] *The SPARC Architecture Manual: Version 8*, SPARC International Inc., 1992.

[25] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, USA, Nov 2004, pp. 85–96.

[26] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002, pp. 304–316.

[27] S. H. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," in *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference*. Helsinki, Finland: ACM, Sep, 2003, pp. 307–316.

[28] S. Zakeri, "Modified SPARC instruction simulator (SIS) to support experimental tagging architectures," Master Thesis, University of Idaho, Aug 2014.

[29] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, R. Draves and R. van Renesse, Eds. San Diego, California, USA: USENIX Association, December 2008, pp. 225–240.

Appendix A:

Tag Manager and User Manager Directives

This appendix summarizes the code implemented in RTEMS for the UITags project. Section A.1 illustrates the tag manager directives. These directives support the tagging scheme at the software level, which includes enabling and disabling the tagging coprocessor, tagging a word, validating specific tags, etc. Directives for the user manager are described in Section A.2. The user manager supports multiuser tags for the expanded RTEMS, helps isolate different users from each other by using tags as described in Section 4.1.1.

A.1 Tag Manager Directives

We have developed a tagging scheme for an operating system that utilizes features of a security tagged architecture microprocessor being developed as part of this larger research project. To support the operating system level tagging, we need to have a *tag manager* in the system. The purpose of the tag manager is to allow system initialization software to configure the initial tags of the system, to allow trusted software to set and modify tags, and to manage the tagging exceptions thrown by the hardware when a tag violation occurs. This section outlines the directives of the tag manager for the RTEMS system. The directives of the tag manager are:

- rtems_tag_partition - tag every word in an RTEMS memory partition.
- rtems_tag_segment - tag every word in an RTEMS memory segment of a specified memory region.
- rtems_tag_word - tag a specified memory word.
- rtems_tag_copy - copy the value and the tag, set the copy bit of the tag.
- rtems_take_ownership - upgrade the tag to higher level tag, so the higher level code could manage the tagged data.

- rtems_release_ownership - downgrade the tag to PC tag.
- rtems_tag_enable - enable the tagging mechanism (this may end up being a startup function that is only accessible as part of the initialization routines and security exception handler).
- rtems_tag_disable - disable the tagging mechanism (a dangerous function, and maybe restricted to just the shutdown phases and security exception handler of RTEMS).
- rtems_tag_set_handler - set the exception handling function – this is for user-level security exceptions that are not already processed by the tag manager.
- rtems_tag_set_lattice - set the rules to form a lattice based on both the Owner field and the Code-space field in the tag.
- rtems_tag_initialize - initialize the tag manager.
- rtems_tag_validate_copybit - check the copy bit of a tag.
- rtems_get_user_tag - get corresponding user tags for users.
- rtems_superuser_check - check whether the Owner field is superuser or not.
- rtems_tag_exception_initialize - handle the tagging exception.

A.1.1 TAG_PARTITION - Tag an RTEMS memory partition

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_partition(
   rtems_id  id,
   rtems_tag tag
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - partition tagged successfully

RTEMS_INVALID_ID - invalid partition id

RTEMS_INVALID_TAG - invalid security tag

RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the security tags of every word of memory within the specified memory partition.

A.1.2 TAG_SEGMENT - Tag a segment of an RTEMS memory region

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_segment(
   rtems_id  id,
   void *segment,
   rtems_tag tag
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - segment tagged successfully

RTEMS_INVALID_ID - invalid region id

RTEMS_INVALID_ADDRESS - segment is null

RTEMS_INVALID_TAG - invalid security tag

RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the security tags of every word of memory within the specified segment of a memory region.

A.1.3 TAG_WORD - Tag a word of memory

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_word(
   int32 *word,
   rtems_tag tag
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - word tagged successfully
RTEMS_INVALID_ADDRESS - word is null
RTEMS_INVALID_TAG - invalid security tag
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the security tag of the specified word of memory.

A.1.4 TAG_COPY - Set the copy bit and tag of a word of memory

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_copy(
   int32 value,
   int32 *word
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - copy bit set successfully
RTEMS_INVALID_ADDRESS - word is null
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the copy bit in the security tag of the specified word of memory, while copying a tag and data value. This bit is used to allow a user function to possess a tagged identifier for future use by directives, but to ensure that it cannot be modified.

A.1.5 TAKE_OWNERSHIP - Upgrade the tag to higher level

CALLING SEQUENCE:

```
rtems_status_code _rtems_take_ownership (
   addr_t addr,
   size_t size
```

```
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - word tagged successfully

RTEMS_INVALID_ADDRESS - word is null

RTEMS_INVALID_TAG - invalid security tag

RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive upgrades the security tag of the specified word of memory to the PC's tag, therefore the higher level system code could manage the data owned by users or lower level entities.

A.1.6 RELEASE_OWNERSHIP - Downgrade the tag to PC tag

CALLING SEQUENCE:

```
rtems_status_code _rtems_release_ownership (
    addr_t addr,
    size_t size
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - word tagged successfully

RTEMS_INVALID_ADDRESS - word is null

RTEMS_INVALID_TAG - invalid security tag

RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive downgrades the security tag of the specified word of memory to the lower level tags.

A.1.7 TAG_ENABLE - Turn on the tagging subsystem of the hardware

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_enable(
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - system enabled successfully

RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to turn on the tagging subsystem. This may end up being a startup function that is only part of the initialization routine and security exception handler.

A.1.8 TAG_DISABLE - Turn off the tagging subsystem of the hardware

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_disable(
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - system disabled successfully
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to turn off the tagging subsystem. This is a dangerous directive and may end up being a startup function that is part of the shut-down routine.

A.1.9 TAG_SET_HANDLER - Set the user-level security exception handler

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_set_handler(
   void *handler()   /* Need to check syntax of this */
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - handler bit set successfully

RTEMS_INVALID_ADDRESS - invalid function pointer

RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the user-level exception handler for tagging exceptions not handled by the tag manager.

A.1.10 TAG_SET_LATTICE - Set the relations in the lattice

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_set_lattice(
   rtems_tag tag1,
   rtems_tag tag2
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - lattice set successfully

RTEMS_INVALID_TAG - invalid security tag

RTEMS_INVALID_PARAMETER - requested relationship inconsistent with current lattice

DESCRIPTION:

This directive sets rules to form the relationship between two tags based on both of the Owner field and the Code-space field in the tags.

A.1.11 TAG_INITIALIZE - initialize the tag manager

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_initialize(
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - RTEMS tagged successfully

RTEMS_INVALID_TAG - invalid security tag

DESCRIPTION:

This directive initializes the tag manager.

A.1.12 TAG_VALIDATE_COPYBIT - Check the copy bit of a tag

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_validate_copybit(
       rtems_tag tag
)
```

DIRECTIVE STATUS CODES:

RTEMS_COPYBIT_NOT_SET - the copy bit is not set

RTEMS_COPYBIT_SET - the copy bit is set

RTEMS_INVALID_TAG - invalid security tag

DESCRIPTION:

This directive checks the copy bit of the tag.

A.1.13 GET_USER_TAG - get corresponding user tags for users

CALLING SEQUENCE:

```
tag_t rtems_get_user_tag(
int user_num
)
```

DESCRIPTION:

This directive forms a user tag for a specific user number and returns it. For example, if the user number is 2, then a tag specific for user 2, `(USER2, USER2, true, READWRITE | DATA_MEMORY | WORLD_NOT_READABLE)`, will be returned.

A.1.14 SUPERUSER_CHECK - check whether the user is superuser or not

CALLING SEQUENCE:

```
rtems_status_code rtems_superuser_check(
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUPERUSER - the owner of the tag is superuer

RTEMS_NOT_SUPERUSER - the owner of the tag is not the superuser

RTEMS_INVALID_TAG - invalid security tag

DESCRIPTION:

This directive checks whether the owner of the PC's tag is superuser or not.

A.1.15 TAG_EXCEPTION_INITIALIZE - handle the tagging exception

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_exception_initialize(
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - exception handler initialized successfully
RTEMS_INIT_FAIL - failed to initialize the exception handler

DESCRIPTION:

This directive initializes the tag exception handler.

A.2 User Manager Directives

Since the RTEMS has been changed from a single user system to a multiuser system, a user manager has been added to RTEMS to support the UI tagging scheme for multiuser system. The purpose of the user manager is to provide the superuser authorities to create, delete and control the non-privileged users. A `usertask_table` is maintained to keep track of the non-privileged users and their tasks. The directives for user manager include:

- rtems_user_create - create a new non-privileged user.

- rtems_user_delete - delete a non-privileged user.
- rtems_user_allocate - determine the next user number.
- rtems_user_reset - reset the user number when deleting a user.
- rtems_usertask_allocate - determine the next task number for a specific user.
- rtems_usertask_reset - reset the next task number for a specific user.
- rtems_user_manager_initialization - initialize the user manager.

A.2.1 USER_CREATE - create a new non-privileged user

CALLING SEQUENCE:

```
rtems_status_code rtems_user_create(
    rtems_id *user_id_ptr,
    rtems_name *user_name_ptr,
    int user_memsize
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - a new user created successfully
RTEMS_INVALID_USER - invalid user tag

DESCRIPTION:

This directive creates a non-privileged user in the system. It first checks whether the thread which request creating a user is the superuser. If this is the case, tag all of the user resources (calculated by using the `user_memsize`) with the proper user tag.

A.2.2 USER_DELETE - delete a specific non-privileged user

CALLING SEQUENCE:

```
rtems_status_code rtems_user_delete(
    rtems_id *user_id_ptr,
    rtems_name *user_name_ptr,
    int user_memsize
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - specific user deleted successfully

RTEMS_INVALID_USER - invalid user tag

DESCRIPTION:

This directive deletes a specific non-privileged user from the system. It first checks whether the thread which request deleting a user is the superuser. If this is the case, tag the user resources with the `LOW` tag.

A.2.3 USER_ALLOCATE - determine the next user number

CALLING SEQUENCE:

```
int rtems_user_allocate(
)
```

DESCRIPTION:

This directive returns the the number for the next user and return the proper user number. For example, if there are 3 non-privileged users in the system, then the new allocated user will have a user number 4.

A.2.4 USER_RESET - reset the user number when deleting a user

CALLING SEQUENCE:

```
rtems_status_code rtems_user_allocate(
    int user_num
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - user number reset successfully
RTEMS_INVALID_USER_NUM - invalid user number

DESCRIPTION:

This directive resets the user number in the `usertask_table` when deleting a non-privileged user.

A.2.5 USERTASK_ALLOCATE - determine the next task number for a specific user

CALLING SEQUENCE:

```
int rtems_usertask_allocate(
    int user_number
)
```

DESCRIPTION:

This directive checks the `usertask_table` for a specific non-privileged user and returns the next available task number. For example, if user 2 has 8 tasks already, then the returned next available task number will be 9.

A.2.6 USERTASK_RESET - reset the next task number for a specific user

CALLING SEQUENCE:

```
rtems_status_code rtems_usertask_reset(
    int user_number,
    int task_number
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - task number reset successfully

RTEMS_INVALID_USER_NUM - invalid user number

RTEMS_INVALID_TASK_NUM - invalid task number

DESCRIPTION:

This directive first checks the `usertask_table` to verify that the user is a active user. If it is, then the directive checks to ensure the task is a active task. If this is the case, resets the task number for the specific user.

A.2.7 USER_MANAGER_INITIALIZATION - initialize the user manager

CALLING SEQUENCE:

```
rtems_status_code rtems_user_manager_initialization(
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - user manager initialized successfully

RTEMS_INIT_FAIL - the initialization is failed

DESCRIPTION:

This directive initializes `the usertask_table` for the user manager.