Security Tagging for a Zero-Kernel Operating System

Jia Song and Jim Alves-Foss
Center for Secure and Dependable Systems
University of Idaho, Moscow, Idaho USA
song3202@vandals.uidaho.edu, jimaf@uidaho.edu

Abstract

Hardware-based security tagging schemes are promising mechanisms for enhancing the security of computer systems. The idea behind security tagging schemes is to attach labels to memory locations and/or registers to carry security information about the tagged data throughout the system. These tags are then used to protect system and user software from attacks and invalid information access. Researchers have also proposed using a “zero-kernel operating system” (ZKOS), a run-time kernel that avoids expensive context switches, by utilizing tags for access control. This paper evaluates key features of RTEMS (Real-Time Executive for Multiprocessor Systems), which is a single user multiple thread executive, and proposes a new hardware-based tagging scheme focused on securing RTEMS as a ZKOS and instantiates the tagging scheme for programs written in ‘C’.

1. Introduction

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 many ways to add security labels (tags) at the hardware level, to help with enforcement of system security properties. 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.

Security tagging schemes attach labels to memory or registers to carry information about tagged data during program execution. They can be used to ensure the semantics of computations are correctly implemented; to isolate code and data, users and system; or be used to enforce security policies at the hardware level. The implementation of tagging in hardware provides developers with enhanced security mechanisms with improved performance, as compared to traditional microprocessors. Therefore, tagging schemes are seen as promising mechanisms that help computer systems work properly and securely.

In this paper a new security tagging mechanism is designed for a “zero-kernel operating system” (ZKOS [11]) running on a security tagged architecture. We have chosen the RTEMS [1] run-time executive system to implement a new security tagging scheme, since it has been designed with almost no protection. Our tagging scheme secures RTEMS by providing access control for code and data by using an efficient hardware-based tagging scheme.

In the remainder of this paper, different tagging schemes from related work are described and summarized in Section 2. Section 3 summarizes the security issues in RTEMS and proposes the tagging features needed. In Section 4, we describes the design of our new tagging scheme. Section 5 shows an example of RTEMS code and associated tags for data and code.

2. Background

We conducted an evaluation of different security tagging approaches, and were able to divide them into two categories: use of security tags for attack prevention and use of security tags for access control. A more detailed survey can be found in [12].

2.1. Tagging Approaches for Attack Prevention

This section provides a summary of some of the proposed security tagging schemes that have been implemented to prevent low-level attacks, such as buffer overflow and format string attacks.

Dynamic information Flow Tracking (DIFT) [13], protects programs by tracking input data from untrusted
I/O input. In this approach, a one-bit tag is attached to each register and each byte in memory. Initially, all tags are initialized as normal, and a module in the operating system initializes untrusted I/O input as spurious data. All of the information that flows from the untrusted data is then tracked by the processor. If a dangerous use of a spurious value is found, the processor generates a trap to a software handler or terminates the program.

Raksha [4] is a flexible architecture based on DIFT. It uses information flow tracking to enforce software security. Raksha adds a 4-bit tag to registers, cache, and memory locations. Among the 4 bits, 2 bits are used for high-level attacks, 1 bit is used to detect memory corruption and the other 1 bit is for low-overhead exceptions.

LIFT [7] 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 or 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. [9] presented a new security tagged architecture that uses a tag table, a multilevel table, and tag caches to reduce the overhead of the security tagged architecture. Compared to the traditional security tagged architecture, the authors show that the new architecture significantly reduces memory overhead.

Yong and Horwitz’s [15] invalid pointer dereference tool is a security-enforcement tool for C programs which protects against attacks via unchecked pointer dereference. This tool can identify unsafe pointers and uses a 1-bit memory tag to indicate it is an appropriate or inappropriate memory location to be referenced by an unsafe pointer.

Decoupling DIFT [6], checks tags associated with the instruction’s memory location. Different from DIFT, it attaches a small coprocessor, which is used to store register tags, tag caches, and perform the tag propagation and checks, to the main processor.

2.2. Tagging Approaches for Access Control

This section provides a summary of some of the proposed security tagging schemes that have been implemented to provide access control.

Mondrian memory protection (MMP) [14] is a fine-grained memory protection scheme that extends the traditional memory protection scheme. Compared with the memory protection, which sets access permissions at the page level, MMP uses two permission bits to support different access control for individual words. These 2-bits represent four meanings: no permission, read-only, read/write, and execute/read.

Sentry [10] 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. Each entry in the metadata cache has 2-bit permission metadata to represent access permission to a specific virtual page.

Loki [16], is a hardware architecture that supports a word-level memory tagging approach. Loki is 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 tag and a 3-bit permission, which represents read, write, and execute.

Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA) [11] 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 control for individual words in memory and registers. At a higher level, another access control mechanism is imposed. 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.

2.3. Summary of Tagging Schemes

Table 1 summarizes the reviewed tagging schemes (with our new approach in the last row), indicating the types of attacks that can be prevented1, 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.

3. Tagging Features Needed in RTEMS

RTEMS [1] is a run-time executive for multiprocessor systems (although we only address a single-processor version in this paper). RTEMS is designed with a single-user multiple thread execution model, and

---

1 The attack list is based on claims by the cited authors, we were unable to verify the claims for “semantic level” attacks.
Table 1. Summary of Tagging Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Buffer overflow</th>
<th>Format string</th>
<th>Command injection&lt;sup&gt;1&lt;/sup&gt;</th>
<th>SQL injection&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Cross-site scripting&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Access control</th>
<th>Hardware support</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIFT [13]</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Raksha [4]</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>LIFT [7]</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dereference pointers [15]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Decoupling DIFT [6]</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low-overhead tagging [9]</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMP [14]</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Sentry [10]</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Loki [16]</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UI Tagging [12]</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Therefore does not have concepts of different users or support for separation of users from the core “operating systems”. Therefore this system was a prime target for initial evaluation of a hardware-based security tagging scheme that provides fine-grain access controls, and that its structure follows the concept of a ZKOS [11].

3.1. RTEMS

RTEMS [1] is a real-time executive that provides a powerful runtime environment to allow various types of services and applications to be embedded. It can perform a complex set of services that includes multitasking, inter-task communication, and dynamic memory allocation. The bulk of RTEMS is written in both the ADA and C programming languages, which makes it easier to modify and port to other processor families, such as ARM, MIPS, PowerPC, SPARC, Atmel AVR etc. In this paper, we focus on the C-language implementation.

RTEMS currently has 18 managers that provide services to user code in support of 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 (Super Core). The SCORE provides services for all managers, and all managers interact with the SCORE via directives. 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 security policies we implement in RTEMS and discuss how to support them using tagging.

3.2. Enhanced RTEMS Security Policy

Based on the analysis of RTEMS, we propose a new security tagging scheme. Our policies are based on a hierarchy of access (see Section 4.3) and classification of functions as external, restricted internal (accessible only by other RTEMS code) or private internal (restricted to a single manager/module). The main security properties of the tagging scheme are:

**Separate system and user data/code.** User code will not have unauthorized access to system data/code, or other user data/code. Instead of using traditional permission rings or supervisor/user mode bit for protection of system data and code from users, we used fine-grain security tags to specify the ownership of the tagged data and code. *Reason:* In RTEMS, there is currently no way to specify the distinction between user’s data and code and system’s data and code. Therefore, both the user...
and system have full access.

RT
erms code will support least privilege. System code is classified by functionality (code space of a single manager or module) to allow lattice-based access control restrictions implementing a least privilege model [8]. Using tags, we specify the privileges for each tagged data item or section of code to control information flow and access to code and data. Reason: All RTEMS system code has full privileges.

Limit the functions called by user and system code. Each module classifies APIs (e.g., external, internal) such that function calls will only execute code at authorized function entry points. Using tags, these access control checks are more efficient than software checks. Reason: RTEMS directives provide a collection of services that are sufficient for users’ tasks, therefore the calling of other important kernel functions should be limited. Restricting use of these critical functions helps protect the system code and data.

Protect the return values. System identifiers returned from APIs are tagged with a bit to indicate if the value was modified or forged. Reason: To prevent users from changing the return values (such as task_id) and sending back modified values to system code, some values returned to the user should be protected. Combined with other security mechanisms, this helps prevent malicious attacks.

4. New Security Tagging Scheme Design

Our tagging scheme uses a 3-part tag consisting of an owner field, code-space field, and control-bits field. Each of these fields helps maintain the correctness and security for RTEMS. Although our new tagging scheme is focused on the security for RTEMS, the issues addressed are common to many types of operating systems and thus the tagging scheme should be usable for other operating systems as well. This tagging scheme will allow us to use the principle of least privilege in protecting resources in the system with a finer-granularity than traditional supervisor and user modes, protection rings or memory management units. We assume the existence of a hardware tag engine that enforces the tag checking and propagation rules (see Sec. 4.4) and generates an exception when the checks fail.

4.1. Tag Format

Fig. 1 depicts the format of our tags, consisting of three fields: Owner, Code-space, and Control-bits. The tag can be written as \(\langle\text{Owner}\rangle^2, \langle\text{Code-space}\rangle, \langle\text{Control-bits}\rangle\).

4.1.1. Owner field. The owner field, helps separate system code/data and user code/data. The intent of this field is to indicate the intended ownership of data. For example, when the thread manager creates a thread for a user, it also creates internal data structures related to that thread. The owner field indicates the user that “owns” the thread and helps prevent one user from asking the thread manager to modify a thread owned by another user.

The values of the Owner field can be classified into six major classes, where each class contains several members: SCORE internal, SCORE Manager internal, Manager, Startup and User. We further divided the SCORE internal class into SCORE internal init group, SCORE internal private internal group, and SCORE internal group. The SCORE internal functions are shared among different SCORE modules. The SCORE private internal and SCORE internal init functions are only used by a specific SCORE module. The Manager internal class is also broken into three similar groups.

By using the owner field, the owner of data or code can be easily identified. For example the tag \((\text{task manager}, \langle\text{Code-space}\rangle, \langle\text{Control-bits}\rangle)\) shows that the data or code is owned by the task manager. In the remaining sections, \(\langle\text{SCORE}\rangle\) in the tag means this field could be one of the possible values in the SCORE class, and the same holds for other classes. Although RTEMS currently only supports a single user, our plan is to expand it to a multiuser system. Therefore, we list multiple users\(^3\) in the possible values of the User class. By having different users for the Owner field, we can ensure that user1 can only access its own data and code, but not other users’ resources.

4.1.2. Code-space field. The code-space field is used to show the class of the code or data and helps control function calls. We do not want users to use some of the system functions, such as SCORE, SCORE internal, and Manager internal functions.

\(^3\)For the User class, each user could be further divided into tasks owned by the user, such as user1-task1, user1-task2, …
fore, firstly, we use the Code-space to indicate which class the code belongs to, and then provide rules to control which classes of code can use which other classes of code (See Section 4.2). The Code-space field is critical for information flow control and memory access control. 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 the manager1’s code for user1.

4.1.3. Control-bits field. The 8-bit Control-bits field is used for further control. We use a copy bit 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) 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, it 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. For example, if user1 uses directive code from 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 indicate this ID has been changed. This allows the system to trust identifiers that come from users without having to keep an internal table of indirect identifiers, and therefore improving performance and simplifying system code.

We allocate three bits 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. 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.

1xx - memory is readable and writable
0xx - memory is read-only
x11 - an entry point to an executable function
x10 - memory is executable but not an entry point
x01 - stack memory (needs to be treated special)
x00 - memory is data memory

We use a world-readable bit to indicate that the tagged data can be read by all entities. This is used when the system (higher level) wants to give the user (lower level) permissions to access the data manipulated by the higher level, such as configuration data.

We reserve the remaining 3 bits for future use (e.g., DIFT-style protection [13]).

4.2. Protection of Function Calls

To prevent a user from misusing or attacking RTEMS code, we implement access control mechanisms for function calls. Conventionally, access control mechanisms specify who can access system resources. In our approach, all functions have tags associated with them. For example, SCORE internal functions have tags \{(SCORE internal)\} and Manager functions (directives) are tagged \{(Manager)\}. When executing a function call, the tag engine will first check the access control rules for function calls to see if the code has permission to call the function. We call this “function execution control”. The manager and SCORE levels of our lattice (see Section 4.3) provide directives which allow external access only to the level below them or to peers. For example, in our tagging scheme, we only want user1 code, which is created in user1, to call its own functions and managers’ code (directives) but not SCORE code or internal functions.

In addition, the internal functions are grouped into internal-private functions for each manager (and callable only by code tagged for that manager), and other internal functions (callable by all peers). Initialization functions are only callable by the initialization subroutines and are permitted to initialize the tags data of their associated managers.

The function execution control rules can even be made more specific. For example, we can set up a rule that user1 code (tagged \{(user1, user1)\}), is not allowed to call an interrupt manager directive that has tag \{(interrupt manager)\} or region directive, which has tag \{(region manager)\}

We can prohibit timer manager internal function from calling protected heap functions (one of the SCORE functions). This allows function execution controls to be explicit and more flexible.

4.3. The Lattice and Our Tagging Scheme

Since the Owner field and Code-space field are used to define the security class of the data, we can implement ideas similar to those of the Data Mark Machine (DMM) [5] to control the information flows within RTEMS. Our solution is a bit different from DMM since we are defining a hierarchy with confidentiality and integrity controls, and not for traditional
multi-level security. However, we still can control access preventing “lower-level” entities from unauthorized access to higher level entities.

We defined two parallel lattices for the information flow control within RTEMS, for each of the Owner and Code-space fields of the tags (Fig. 2). To simplify the diagram, we assume that boxes higher on the figure have higher integrity and security classes than those below them, and that the $\leq, +$ and $\otimes$ operators support that hierarchy. This dual-use (security and integrity) is different from traditional security classes, but maps well to the standard use of privilege rings or supervisor/user execution modes in hardware. In addition, we have merged the private internal and other internal functions into a single box. The connections between managers and the internal functions of other managers will be restricted to the non-private internal functions discussed in Sec. 4.2.

In our model the security class of an entity, $a$, will be denoted as $\mathbb{a}$, and it can be written as $\mathbb{a} = \{\text{Owner}(a), \text{Code-space}(a)\}$. We have ignored the control-bits in this discussion since they are used separately from the lattice-based controls and formulas. The definition of the least upper bound, $\oplus$, of the security classes of two tags is:

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

For example, if $x = \{\text{user1}, \text{semaphore manager}\}$ and $y = \{\text{user2}, \text{semaphore manager}\}$ then according to our lattice, $x \oplus y = \{\text{user}, \text{semaphore manager}\}$.

### 4.4. 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 we use those classifications, in the context of the C programming language, to define tagging rules needed to satisfy our security concerns based on the partial ordering and lattice concepts just introduced. We have further refined these rules at the assembly-language level, but have not included that discussion due to space constraints (see [12] for details).

#### 4.4.1. Tagging rules for basic values in C

During execution of a program, the tag of the current thread is denoted using the program counter’s tag, which has the security class $PC$. The tag of the variable $a$ is the tag of the memory location of $a$ and its security class is denoted $\mathbb{a}$. The tag of a literal, or constant, $n$, is the same as the tag of the PC. We made this choice since 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 $a[i]$.\(^5\) The tag of a value referenced by a pointer, $*p$ or structure $p->f$ld or $p.f$ld 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).

#### 4.4.2. Rules for arithmetic and logic operations

The rules for arithmetic and logic operations are used to specify the tag of the resulting value of the operation. For example, for the mathematical expression $a + b = c$, $a$ and $b$ are two operands, and $c$ is the result of this operation. 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 $\forall p$, then we just need to consider the security class of $b$. If the copy-bit of both operands are $\forall p$, then the security classes of both $a$ and $b$ need to be used and the security class of the result will be $a \oplus b$. 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 we assume it has been modified. The particular rules for arithmetic and logic operations are shown in Table 2.

#### 4.4.3. Rules for comparison

In the C programming language, a comparison expression gets a Boolean ex-

<table>
<thead>
<tr>
<th>Copy bit of a</th>
<th>Copy bit of b</th>
<th>Security class of the result</th>
<th>Copy bit of the result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$cp$</td>
<td>$cp$</td>
<td>$PC$</td>
<td>$\forall p$</td>
</tr>
<tr>
<td>$cp$</td>
<td>$\forall p$</td>
<td>$b$</td>
<td>$\forall p$</td>
</tr>
<tr>
<td>$\forall p$</td>
<td>$cp$</td>
<td>$a$</td>
<td>$\forall p$</td>
</tr>
<tr>
<td>$\forall p$</td>
<td>$\forall p$</td>
<td>$a \oplus b$</td>
<td>$\forall p$</td>
</tr>
</tbody>
</table>

\(^5\)In cases where the copybit of the memory location is set, we use the tag of that memory location and not the least upper bound.
pression as the result. For the comparison $a > b$, to make sure we maintain the proper tag, the result of the comparison should get the appropriate security class. Similar to the rules 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 comparison is always reset since it is a new value. The explicit rules for comparing $a$ and $b$ are shown in Table 2.

4.4.4. Information flow rules for assignment statement. For assignment statements, such as $y = x$, we need to check whether the information is allowed to flow from $x$ to $y$. In the lattice model of information flow, the assignment statement of $y = x$ requires the relation $x \leq y$ to ensure confidentiality. In addition, the security class of the current thread is also important to ensure integrity. We do not want unauthorized copying or modification of data. Therefore we require $y \leq PC$ for integrity. The value $x$ can be a variable, a constant, a complex expression, etc. The rules for assignment statement ($y = x$) are:

- When the copy-bit in the tag of $x$ is $cp$ and if $y \leq PC$, 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 $cp$ and if $x \leq y \land y \leq PC$, the information flow is allowed to flow from $x$ to $y$ and tag of $y$ is unchanged, 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. We use our rules for arithmetic and logic operations (Section. 4.4.2) to define the tag of the expression that $x$ represents. Therefore, 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 $y \leq PC$ seems to violate the traditional security “no write down” model of Bell-LaPadula [2], however, it satisfies the integrity controls of Biba [3].

4.4.5. Information flow rules for if statement. For the if statement, such as if(expression)
commands; and if(expression) commands1; else commands2;, we need to check whether the expression can be read by code executing this if statement. To ensure that the expression can be read by code the relation between PC and expression needs to be checked before running the if statement. Therefore, the rule for if statement is:

- If $expression \leq PC$, then the statement is allowed.

As with assignment, the expression can be a variable, a constant, a result of comparisons etc., and its tag is set using the appropriate rules. The commands in the then and else statements are controlled by their appropriate rules.

4.4.6. Information flow rules for while statement. For the while statement, such as while(expression) commands;, we need to check 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 $expression \leq PC$, 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 while statement will be used at every loop iteration.

4.4.7. Information flow rules for function calls. When performing a function call, we first check the function execution control rules (see Section 4.2). If the function is allowed to be called by the caller, then we need to make sure the parameters passed to the function are allowed to be accessed by the code. 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 $parameter1 \leq PC$ and $parameter2 \leq PC$, then the function call is allowed to execute.

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

Directive code, which needs to set the copy-bit of the tag of the return value, will call the directive `rtems_tag_copy` of the tag manager (see Section 4.5) before returning the value. The `rtems_tag_copy` copies a value and sets the copy-bit of its tag, for return to the caller, bypassing the standard tagging rules for assignment.

4.4.8. Other C constructs. We will not define tagging rules for other C-language constructs, such as the for loop or switch statement since they can be mapped to other constructs we have defined. We have also not defined rules for break and continue, since their only function is to change location of execution within the same code space and therefore do not have information flow concerns for our tagging model.

4.5. Tag Manager

To support operating system level tagging for RTEMS, we need to add a tag manager to 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, to allow the directives to set and validate the copy-bit, and to manage the tagging exceptions thrown by the hardware when a tag violation occurs. The directives of tag manager can be divided into three classes according to their functionalities: tag a section of memory, control the copy-bit and control the tag engine.

5. Adding Security Tags to RTEMS code

Fig 3 depicts a small segment of user code and two directives to illustrate our security tagging scheme. Starting in user code (lines 1-18), the tag for the PC is $(user1, user1, \overline{cp})$.

On lines 2-5, user code creates 4 variables `Task_id`, `Task_id2`, `Task_name` and `Task_name2`. These variables should be tagged $(user1, user1, \overline{cp})$ to indicate that they are created by the user1 in user1 code-space.

In the `Init()` function, status is given the tag $(user1, user1, \overline{cp})$ on line 7.

Then, on lines 9-11 (the function call), we check permissions for the user to make the call to `rtems_task_create()`, using our function execution rules. Since `rtems_task_create()` is a directive of task manager, which should have tag $(task manager, task manager, \overline{cp})$ associated with its code, and the PC's tag is $(user1, user1, \overline{cp})$, the function call is allowed. All parameters are either local variables whose tags are PC's tag, or constants,
/* User code: */
rtmess_id Task_id;
rtmess_name Task_name;
rtmess_task Init(rmess_task_argument argument)
{
rtmess_status_code status;
...
status = rmess_task_create(Task_name , 1,
RTMESS_STACK_SIZE+2, RTMESS_DEFAULT_MODES,
RTMESS_DEFAULT_ATTRIBUTES, &Task_id);
status = rmess_task_create(Task_name , 1,
RTMESS_STACK_SIZE+2, RTMESS_DEFAULT_MODES,
RTMESS_DEFAULT_ATTRIBUTES, &Task_id2);
...
status = rtems_task_delete(Task_id);
status = rtems_task_delete(Task_id2 + 1);
}

/* Directive code: */
rtmess_status_code rtems_task_create(
rtmess_name name,
rtmess_task_priority initial_priority,
size_t stack_size,
rtmess_mode initial_modes,
rtmess_attribute attribute_set,
Objects_Id *id)
{

{register Thread_Control *the_thread;
...
the_thread = RTEMSS_tasks_Allocate();
...
rtmess_tag_copy (the_thread->Object_id , id);
...
return RTMESS_SUCCESSFUL;
}

rtmess_status_code rtems_task_delete(
Objects_Id id)
{
{register Thread_Control *the_thread;
Objects_Locations location;
Objects_Information *the_information;
...
if (!rtems_tag_validate_copybit(id)) {
    return RTMESS_ACCESS_VIOLATION;
} else {
    the_thread = _Thread_Get(id, &location);
...
    _Thread_Close (the_information , the_thread);
    _RTMESS_tasks_Free (the_thread);
...
    return RTMESS_SUCCESSFUL;
}}

Figure 3. Sample C code

therefore they are valid parameters. Since the function call is allowed, the PC’s tag will be changed to (user1, task manager, cp), when the system executes the directive code.

In the directive code rtems_task_create(), lines 21-36, the values of initial_priority, name, stack_size, initial_modes, attribute_set, and *id are all from the user, consequently they still have the tags (user1, user1, cp) associated with them. The pointer, the_thread, is declared in task manager for the user1, with tag (user1, task manager, cp).

On line 31, the task manager private internal function RTEMSS_tasksAllocate() is called. According to our function execution control rules, the current running thread which has tag (user1, task manager, cp), is allowed to call this function.

On line 33, rtems_tag_copy () is used to copy the value and tag of the_thread->Object_id to id and sets the copy-bit of the tag. Therefore, *id is tagged (user1, task manager, cp). The settings of the copy-bit allow the user to receive an authenticated ID and reuse it later.

When returning from the directive, the PC’s tag will change to (user1, user1, cp) and Task_id will get the tag (user1, task manager, cp).

Similar to the code on lines 9-11, after executing the code of lines 12-14, Task_id2 gets the tag (user1, task manager, cp).

On line 16, rtems_task_delete() is called to delete the task associated with Task_id. Because Task_id has tag (user1, task manager, cp), the copy-bit is set. Therefore rtems_task_delete trusts the value of the Task_id after checking it on line 44, and deletes the task for the user1 (see below).

However, on line 17, for Task_id2 + 1, Task_id2 has tag (user1, task manager, cp), and i has the PC’s tag ((user1, user1, cp)). According to the rules for arithmetic operation, the copy-bit of the Task_id2’s tag is set, so the tag of the result value will be 1’s tag ((user1, user1, cp)) and the copy-bit will be reset to cp. This means that the parameter is data from user1, and not from the task manager. On line 17, the user passes the modified Task_id2 to the rtems_task_delete() directive. This directive, on line 44, calls rtems_tag_validate_copybit() and will not trust the parameter because the copy-bit in the tag is not set. Consequently the directive will not perform the deletion of the task associated with Task_id2 for the user1.

On line 47, rtems_task_delete() calls _Thread_Get(), which is a SCORE function and it has tag (thread, thread, cp). The _Thread_Get() function can be called by the directive (Manager class) using our function execution con-
control rules and the parameter is allowed to be accessed under our information flow rule for function calls. This function, not shown here, will check the tag of id, and if it matches the tag of the calling function, then it will return a pointer to the appropriate thread context with the tag (user1, task manager, \( \overline{\text{77}} \)). Then the calls, lines 49 and 50, to _Thread_Close(), which is a SCORE function and _RTEMS_Tasks_Free(), which is a task internal function, will be allowed.

6. Conclusion

This paper proposed a new security tagging scheme that enhances access control through least privilege, while enabling good system performance. We chose RTEMS as our target system, as an exemplary ZKOS. Our tagging scheme has been designed to replace the classical supervisor/user mode operating model of operating systems and therefore protect the operating system code and data. The core pieces of our scheme focus on identifying controlling “code-space” and “owner’s” for all code and data in the system, and providing appropriate access control.

The intent of this work was to develop and evaluate a tagging-based security scheme for a ZKOS, adding access control protections. We still need to investigate performance enhancements to make it realizable in hardware. Since almost every execution of instructions needs a tag checking of the source or destination operands’ tag, it increases the overhead of the system. For example, normally, the LOAD instruction loads values from memory space to registers, but in our tagging scheme, the LOAD instruction has to check the value’s tag and store it as the register’s tag additionally. To minimum the overhead, we plan to cache as many tags as we can to speed up the tag checking operations and we are confident that we can use a tag compression scheme, and data and code spatial locality information to reduce the overhead. That development and evaluation is a companion part of this project.

We validated our tagging scheme by extending the Sparc Instruction Simulator in GDB, to simulate the tag engine. A companion part of this project is developing an FPGA version of the hardware for further experimentation and validation.

7. Acknowledgements

This work was funded in part by AFRL contract FA8750-11-2-0047. The opinions and findings expressed in this paper are solely those of the authors and do not represent the position of the AFRL.

References