Evaluating the Use of Security Tags in Security Policy Enforcement Mechanisms

Jim Alves-Foss, Jia Song, A. Scott Amack, Lawrence Kerr, Stuart Steiner
Center for Secure and Dependable Systems
University of Idaho, Moscow ID, USA
jimaf@uidaho.edu, [song3202, amac0723, lawrence.kerr]@vandals.uidaho.edu, ssteiner@ewu.edu

Abstract

Security tagging schemes are known as promising mechanisms for providing security features in computer systems. Tags carry information about the tagged data throughout the system to be used in access control and other security mechanisms. This paper discusses several different uses of security tags related to different security policies, highlighting appropriate uses of the tags. The evaluation of the use of tags is presented in the summary of three security tagging application domains. One domain, using hardware-based tagging to prevent high-level attacks, was not found to be feasible. A project to use hardware-based tagging for OS security enhancement and a project that uses software-based tagging for multi-level secure document management were successful.

1. Introduction

Security tagging schemes attach security attribute labels to data, to be propagated and evaluated during runtime. They can be used to ensure that the semantics of the computation are correctly implemented; to isolate code from data, and user from system processes; or to provide low-level support for enforcing security policies. Security tagging in hardware provides developers with enhanced security mechanisms with improved performance, compared to traditional processors. Software-based tagging in the operating system, middleware and infrastructure can provide additional common security mechanisms to enhance security and reduce the burden on developers.

After an evaluation of several studies involving security tagging, we have determined that the primary 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 concept of hardware-based memory tagging can be traced back to the Burroughs B5000, introduced in 1961, which used a simple 1-bit code to differentiate between data and instructions, partially to protect from corruption of the stack [4]. This later grew to a 3-bit tag in the B6500. Feustel proposed expanding this to 5-bit tag [7]. This eventually grew into a security co-processor based on fine-grain type enforcement in the LOCK program [16]. This type enforcement concept now lives on in software in SELinux [19].

Over the last few decades the microprocessor industry seems to have maintained simple tagging implementations. This includes bits to differentiate between instruction types, mode status bits to indicate operating modes/rings of protection, and simple access control bits (e.g., read/write) on memory pages.

Over the past 10 years there has been resurgence in interest in hardware-based security tagging. Initially these tags were designed and implemented to protect against low-level attacks, such as buffer overflow and format string attacks [15][20][27][29]. Tags have also been implemented in some architectures to support memory access control [21][22][28][30]. For example, Mondrian Memory Protection (MMP) [28] and trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA) [22] are fine-grained memory protection schemes. Recently, some researchers have developed hardware-based security tagging schemes that they claim support prevention of high-level attacks, including SQL (Structured Query Language) injection and cross-site scripting [5][8].

In this paper we look at some examples of hardware and software-based tagging for security policy enforcement, and evaluate the effectiveness of those approaches. We highlight these approaches through three example application domains.

In the remainder of this paper, concepts related to the security policies and tagging schemes are introduced in Section 2. Section 3 summarizes a project to evaluate the effectiveness of hardware-based tags for enhancing OS security for access control and low-level attacks. Section 4 evaluates the use of hardware based tags to prevent high-level attacks. Section 5 consists of a discussion of a software-based tagging solution for information management infrastructures. Section 6 concludes the work and provides some directions for future work.

2. Background

The literature on security tagging usually focuses on using tags for access control, or for attack prevention. This section provides a brief discussion of each of these concepts and the policies they enforce.
2.1. Access control policies

An access control policy specifies when a subject (a user or a process) can perform an action on an object (a resource of the system). That action can be as simple as a read or write operation, or as complex as business or control system transactions. Access control policies require an identification of system resources and users, an assignment of security labels to those entities, and relationship operations between different security labels. In addition, we need assurance that the mechanisms that enforce the policy satisfy the following constraints \(^1\) (these are based on the properties first set out in the Anderson report [3]):

* Always invoked: Upon every operation, there is a check for access control permissions. This guarantees that any changes to the permissions or security labels is enforced at the time of action and access is not granted based on a legacy decision. For example, an access control system for a file system may use metadata stored on the disk drive to make an access decision. If a user requests to open a file, the system may check the metadata and then cache the result of the check. Future requests, if consistency is not maintained and the system checks the cached result, may miss changes to the metadata and therefore violate this property.

* Non-bypassable: There is not a way to bypass the access control mechanisms and directly access the resource. Consider the file system example above. If the user is able to directly read data from the disk drive they can bypass the associated access controls.

* Tamperproof: The security system and metadata must be protected from unauthorized modification. In the file system example, if a user can modify the access control permission metadata, or the code that evaluates the checks, then the system will not be secure.

* Evaluatable: We need assurance that the security aspects of the system perform as specified. This can only be done if the security mechanisms are isolatable from the non-security relevant portions of the code. A secure system must be structured and designed so that it can be reviewed and provide a high level of confidence in the correctness of the system.

According to Schneider [17], we can implement access control security mechanisms using run-time enforcement monitors. These monitors are non-bypassable and always invoked, and they check the behavior of the monitored system (e.g., in the file system example they monitor the user’s file access requests). If an attempted security violation is discovered, they terminate the offending processes, suppress the behavior, or modify the behavior (e.g., change the request, insert new actions, etc.) [12]. Such run-time enforcement behavior requires a system that satisfies the preceding constraints. Many of the tagging schemes found in the literature fit into this category.

2.2. Attack prevention policies

There are a wide range of different attacks against computer systems. The ones that security tagging researchers and developers focus on are typically those that can be address through type-enforcement or run-time semantic violation detection.

* Type-enforcement uses tags to differentiate code from data, and system data from user data. In addition, type enforcement can place additional typing information on the data to limit the types of operations that hardware can perform on the data. Successful use of these tags can prevent user data from exploiting a buffer-overflow vulnerability, since the user data is not tagged as valid instructions or return addresses.

* Run-time semantic violations involve evaluation of user data in a different context than was expected. For example, an SQL injection attack (see Section 4) uses an application that embeds user data into an SQL query to trick the SQL interpreter into evaluating the user data as SQL commands. A security tagging mechanism that prevents this must enable the differentiation of the user data from the SQL commands (Section 4).

3. Hardware tagging for OS support

We developed our own hardware-based tagging mechanism that supports both access control and low-level attack prevention. This enables us to evaluate the limitations and effectiveness of a wide range of such mechanism. Further details of this project can be found in companion publications [23][24][25][26].

For this project we decided to examine using hardware-based tagging to add security to the RTEMS OS. RTEMS [2] is a real-time executive that provides a powerful runtime environment to allow various types of services and applications to be embedded. It is not a hosting OS, but provides services for a single-user multi-threaded execution model for embedded systems.

RTEMS provides a complex set of services that includes multitasking, inter-task communication, and dynamic memory allocation. RTEMS consists of super core (SCORE) services and 18 managers. The 18 managers provide different services for user code, and the SCORE provides kernel functions that support the managers. Each manager provides a well-defined set of services by using directives that take the place of system calls. By using directives, developers are able to develop applications with better control of different tasks. Each manager also has internal functions that it

---

\(^1\) Although these constraints date back decades, many modern systems do not correctly enforce them.
uses to support implementation. These manager internal functions are not intended to be used by user code. The SCORE provides services for all managers, and all managers interact with the SCORE via directives. These structures and functions are keys to the internal working of RTEMS.

However, since RTEMS executes in a single-user execution model in a single address space, there is no separation of RTEMS from user code, no support for multiple users, and no protection within the OS itself. Therefore RTEMS is a good candidate for evaluation of the benefit of hardware-based tagging.

For access and information flow control, we developed a three field tag consisting of owner, code-space, and control-bits fields. Each of these fields helps maintain the correctness and security of RTEMS. Although our new tagging scheme is focused on enhancing security in RTEMS, the issues 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.

3.1. Hardware tagging goals

We develop a tagging scheme with the ability to add security tags to sections of code and data, allowing for fine grain (word-level) protection. Our tags were developed to enhance the security of RTEMS in support of five goals:

1. Separate system from user data and code. In RTEMS, with all code in the same code space, there is no way to implicitly differentiate user's data and code from system's data and code. Because of this issue, we saw a need for a mechanism to identify the owner of code and data in RTEMS. By using tags to specify the ownership of the tagged data or code, the system can securely manage the data or code and can help with the protection of system code and data from users.

2. Classify RTEMS code. Although RTEMS and traditional operating systems give the kernel code ultimate privileges, some parts of the kernel do not need it. If malicious code gets permission to run as kernel code, it can gain ultimate privileges and do whatever it wants. By using tags to classify different code modules, we can specify the privileges needed for the code. The classification of RTEMS code will also help control information flow and access to code and data.

3. Limit the functions called by user. Since the directives of RTEMS provide a collection of services that are sufficient for users' tasks, the calling of other important kernel functions should be limited. The restriction of the use of these critical functions helps protect system code and data. Supported by the security tagged architecture, access control of functions calls provided by tags can be implemented in hardware, reducing the consumption of software resources.

4. Protect the return values. RTEMS' directives can return handles and descriptors to the user code, to be used in subsequent calls. For example, creation of a task returns a unique system-wide task id. To delete the task, the user code passes a task id (unvalidated) to the system. To prevent users from changing the return values (such as a task id) and sending back modified values to system code, we add the ability to protect these values with a copy-bit. Combined with other security mechanisms, this helps prevent malicious attacks, and simplifies design of the OS.

5. Control the information flows in RTEMS. To prevent RTEMS from leaking information to unauthorized users, information flows in RTEMS need to be controlled properly. Tags carry information about the tagged data, which help track and control the information flow.

3.2. Hardware-tagging design

Based on the preceding goals, we developed a hardware-based tagging scheme that assigns security tags to every word of memory and registers. Hardware provides tag propagation and checking, managed by new RTEMS tag and user managers [23][24][25][26].

In our tagging scheme, a tag consists of three fields: owner, code-space and control-bits. The owner field helps separate system from user resources (i.e., code and data). It identifies the owner of the data and code. 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 beyond what is typically found in an OS.

Together the owner and code space specify the security class of the tag. The third field of the tag, control-bits, consists of one copy bit and three memory type bits, one world-readable bit and three reserved bits. The details of the tag format are discussed in [10].

Tag format and meanings of each field. As shown in Figure 1, the tag in our tagging scheme consists of
three fields: Owner, Code-space, and Control-bits, written as (<Owner>, <Code-space>, <Control-bits>).

The Control-bits field is used to further support least privilege by providing some typing and access control information to the system resources. 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 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. Three bits are allocated for memory type, where 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. 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.

![Figure 1. Detailed tag format](image)

### 3.3 Protection of calls and functions.

To prevent a user from misusing or attacking RTEMS code, we implement access control mechanisms for the function calls.

In our approach, all system functions and calls have tags associated with them. 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. Their permissions are based on the hierarchy of code (user, manager, and SCORE) as well as additional classification of internal, private, and initialization functions. We call this “function execution control”. By performing access control for the function calls, we are able to control who can execute which function. In general, code can only call functions in the level directly above them in the hierarchy. We add some additional control logic in the hardware to allow for same-level calls between managers and for private functions for managers. We have also implemented a hierarchy for user programs [24] with the assumption that this will be used in an embedded system with a fixed set of possible users.

### 3.4 Limitations of hardware tagging

Most of the hardware based tagging schemes presented in the literature are limited to protection against buffer overflow attack (bounds checking violations), uninitialized memory usage, or other runtime violations of the code. Specifically, data is tagged with a simple indication of provenance, which can be modelled with our tags:

**User data:** Data that comes from an external source is considered tainted by these schemes and is marked by software (typically operating system drivers) that brings that data into the system. Hardware protections are in place to prevent the use of this tainted data in control flow operations; and to propagate the taint tags. This is the Dynamic Information Flow Tracking approach (DIFT) introduced by Suh [27].

**Internal data tagging:** In addition to user data, some of the schemes tag data with a color to indicate membership in a data group (usually adjacent buffers are differently colored) or to indicate that a memory region is uninitialized. Hardware protections are in place to prevent overflow between differently colored regions or use of uninitialized memory regions.

**Fat pointers:** Some of the tagging schemes add additional information to a pointer, indicating base and bounds of the region being reference. Hardware protections are put in place to ensure that the region access by a reference pointer is not out of bounds.

More complex tagging schemes, such as the one we presented in this section, add security labels or data types in the tags. These labels provide a richer set of control over data access, information flow and flow control. Hardware based tagging schemes, including ours, have the following limitations:

**Management of tags:** All tagging scheme require some software support from the OPS. Any tagging scheme beyond the basic DIFT approach requires tag management functions. This includes: initial tagging of memory and registers, starting and stopping of the tag co-processor (to allow for initialization and error handling), exception processing, including retrieving the tags used to generate the error, setting or changing the tags, and access controls to the tagging functions. More complex tagging may involve user tag definition and strong access controls to the tagging functions; requiring more controls to be implemented.

**Memory constraints:** Most proposed tagging schemes have large amounts of tag overhead, because
the hardware does not have organizational knowledge of the data structures and tag domains in the programming language and therefore must be conservative and provide more tags than necessary. A software-based approach can use compile-time and run-time knowledge of data structures within a hierarchical memory organization to provide implicit tags to the data to reduce the overhead.

**Semantic restrictions:** The hardware designer, and therefore the tag co-processor have limited semantic knowledge of the run-time tagging needs of the software. The needs change with each new language, protocol and service. A tagging scheme provides basic labeling capabilities with tag propagation and checks based on a fixed (or limited) set of rules, and cannot provide direct enforcement of higher level security policies. An example of this is provided in the following section.

4. **High-level application tagging**

There have been claims in the security tag architecture community that hardware tags can now be used to address higher level attacks; those that are specifically attacking application logic, and not fundamental flaws in the underlying operating system, services or programming language [5][8]. We call these types of attacks semantics attacks, in that they result from a semantic disconnect between the programmer’s view of interactions with the world, and the attacker’s actions. In this section, we provide a brief overview of these types of attacks and then discuss the attempted use of a hardware-based tagging mechanism to prevent them.

4.1. **Semantic and code injection attacks**

The precise definition of a semantic attack varies greatly. In Schneier's original article [18], when he referred to semantic attacks, he was referring to the “human element” and attacks such as modern day phishing or pump-and-dump schemes. This type of definition has become the definition of “semantic hacking.” According to PC Magazine one definition of semantic attack is: “The use of incorrect information to damage the credibility of target resources or to cause direct or indirect harm” [14].

Other definitions of semantic attacks refer to some type of malicious code injection into a web browser or running code. For this paper, a semantic attack is defined as an attack that uses unexpected input to force unexpected behavior in the code, but not a violation of the normal run-time behavior of the language (e.g., not a buffer overflow). This definition of semantic attack focuses on the semantic disconnect between the application programmer’s mental model of the external environment and reality. An injection attack is one type of these semantic attacks.

In most cases an injection attack will occur based on the application programmer not correctly validating user supplied input. If one examines the OWASP top ten vulnerabilities [13], it is evident that most of those vulnerabilities are considered semantic attacks; yet, they are more precisely classified as injection attacks, which refer to part of the attack process.

**SQL Injection Security Mechanisms.** OWASP classifies injection attacks as the number one attack for web applications, in particular SQL Injection. SQL injection occurs when an application builds an SQL query using user input that is not appropriately filtered and then sends that query to the interpreter.

A SQL injection is very difficult to detect because, to the database and the underlying operating system and hardware, the SQL injection appears as a normal SQL query from an authorized client (the web server). There are three main security mechanisms that protect against SQL injection. None of these can directly use hardware-based tagging.

**Application Developer:** Most mechanisms to combat SQL Injection are implemented on the application developer's end, forcing the application developer to filter the user input. This is best accomplished through language specific filtering functions. For example, in the PHP language there is a function called `mysql_real_escape_string()`. This function prepends backslashes to a set of characters, including the single quote. The problem with leaving it to the application developer is that the developer may not know about or understand SQL injection attacks.

**Filtering and Monitoring Software:** Filtering and monitoring tools at the Web application and database levels will help block attacks and detect attack behavior. At the application level, organizations can possibly prevent SQL injection by implementing runtime security monitoring; furthermore web application firewalls can help organizations by creating behavior-based rule sets to block SQL injection attacks. Database activity monitoring can filter attacks on the back end, especially for known SQL injection attacks. There are generic filters that query for typical SQL injections such as uneven numbers of quotes. The drawback is that the software can be very expensive, and even if the business is able to afford the software, the installation/setup is often beyond the database/business owner's skill set.

**Database Patches:** The risks associated with SQL injections are increased when the databases tied to the web applications are poorly maintained, including poor patching and configuration. Part of the configuration
process requires better management on web application's associated accounts, especially with accounts that interact with the back-end databases. Many problems arise due to database administrators not understanding security, so the administrators give the web application accounts greater privileges than required. These super accounts are very vulnerable to attack and thus greatly broaden the risks to databases.

Cross-site Scripting Security Mechanisms. Cross-site Scripting (CSS) flaws occur when an application includes user-supplied data in a page sent to the browser without properly validating or “escaping” that content. CSS vulnerabilities target scripts embedded in a page which are executed on the client-side, in the victim's web browser, rather than on the server-side, where the web page is held. These vulnerabilities can have significant consequences such as tampering and sensitive data theft.

Like SQL Injection, CSS is also very hard to detect. There are three main approaches to combat CSS.

Filtering and Monitoring Software: Similar to SQL Injection filtering and monitoring tools can be applied at the Web application level. As with SQL injection there are the two drawbacks of expense and insufficient skills.

Validate input: Validating input is very complex. A good programmer will validate the input received; however, a novice programmer has no concept of CSS and how to validate the input. Validating input quickly fails when the programmer has no knowledge of the security problems.

Allow the client to disable client-side scripts: For CSS to be effective, the malicious script needs to run in the victim's browser. By disabling scripts the client browser cannot be compromised. This is an effective mechanism; but it relies on web sites not requiring or needing scripts to display the web page. In our modern world, most users rely on dynamic web content to enhance their web experience, negating this solution.

The preceding outlines the two most prevalent web vulnerabilities and the security mechanisms required to prevent them. In every case, the required security mechanisms relies on the programmer or system/database administrator to implement/configure the mechanism, and then handle any security violations. First, it presumes the programmer fully understands which security mechanism is required and presumes the programmer knows how to properly code that mechanism. Other defenses are also problematic. Filtering and monitoring are often prohibitive based on the cost or the system administrator knowledge.

These problems have led researchers to try to find a common base mechanism that can be used to prevent these attacks. Some claim that their hardware-based security tagging solve these problems for SQL [5][8] and CSS [5] attacks. We attempted to duplicate their claims by enhancing our OS security tags, but found that the hardware tagging just added complexity to the problem and no new functionality.

4.2. Hardware tagging for semantic attacks

Hardware tagging schemes have been successful in access control, least privilege and buffer overflow prevention, as our technique demonstrates. So why not use them for SQL, CSS and other attacks?

The concept is understandable; this is user supplied data that is being processed in an unexpected manner, resulting in a violation. Prevention of a semantic attack requires proper handling of user input, which often assumed to be similar to the problem of a buffer overflow. However, there are key differences.

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 storage of the return address on the stack is 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. For an application-level injection attack, things are a bit different. Taking SQL injection as an example, a hardware-based security tagging approach needs a process that receives user input and tags it — similar to the buffer overflow use case. That user input is then combined with server generated 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 the 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 concept that we can extract from query
processing to configure the hardware to generate a security exception. Therefore, the approach taken behind the claims 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 software.

The impact of this is twofold. First 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. In addition, 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. Software controls need to tell the tagging hardware to monitor the processing; which requires the application programmer to be involved in configuring and appropriately using the mechanism, since the operating system will not be aware of the processing.

Since hardware can not differentiate what part of the query processing is occurring, all queries with user data will have to 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 CSS attack) and appropriately process the SQL query to remove errors, with the correct SQL version and the appropriate context. This is actually more complex than requiring the developer to use a standard SQL query library with security features and adds no new functionality to that approach.

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.

We agree that there is a need for further research to determine common elements of high level semantic attacks and common security mechanisms for them. However, 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 hardware-based tagging mechanisms.

5. **Software tagging for infrastructure security**

In the preceding sections we discussed hardware based tagging capabilities and limitations. In this section we look at software-based tags.

Operating systems use a variety of tags to support security. These tags can be the simple user/group identifiers and rwx access controls found in Unix-based systems. SELinux takes this much further by providing “type” labels to system resources and then controlling access to a wide range of operations (system calls) based on these types.

Higher-level software that is part of the infrastructure includes web and database servers. These also typically implement security tagging at least in terms of users and access control values. However, there is often little coordination between the applications and servers in support of complex tags.

To address one aspect of this issue, we have been examining the use of security tags in XML databases and documents. We began our work by developing mechanisms that support security tags in an XML database [9][10]. The important aspects of these mechanisms are:

**Portable and Extensible tags**: Our XML-based tagging scheme allows for portability and extensions of the tag data using standard XML mechanisms.

**Multi-level tags**: Our mechanisms support multi-level security in the tags. This requires support of polyinstantiation [6] in the database hierarchy to prevent unauthorized information leakage.

**Database Operations**: Our mechanisms support insertion, deletion, modification and retrieval operations that are compliant with the multi-level security policies and tags.

**Polyinstantiation** describes a situation in which information related to a particular primary key exists in the database under multiple security classifications databases [6]. An example is depicted in Figure 2. Each field in the relation is assigned a classification. The *starship* attribute is the apparent key – the attribute which appears to the user to be the primary key. However, in order to allow the second tuple, an effective key is generated by combining the apparent key with the highest classification of a field in the tuple, the tuple classification. Creating and using this as our key allows for the existence of both tuples. In Figure 2, a low clearance user (at the U level) views this relation as containing only a single tuple, tuple 1. In this situation, the polyinstantiation enables a *cover story* for the true classified mission.

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Starship</th>
<th>Mission</th>
<th>Target</th>
<th>Tuple class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Enterprise U</td>
<td>Explore U</td>
<td>Talos U</td>
<td>U</td>
</tr>
<tr>
<td>2</td>
<td>Enterprise U</td>
<td>Explore U</td>
<td>Rigel S</td>
<td>S</td>
</tr>
</tbody>
</table>

**Figure 2. Polyinstantiated tuple**

The security levels are stored as XLM attribute tags and are manipulated by tagging aware software. Any changes to a tuple in the database will be processed at the security level of the user, and the changes will modify the polyinstantiated data correctly. For example, a deletion of the unclassified mission will still preserve the classified mission data in the database along with notations about the deletion.
Using these XML tags we have been successful in supporting the creation, insertion, extraction, deletion and modification of tuples in the database in support of the multi-level security policy [9][10]. We have used this base technology to explore other aspects of software-based tagging in this environment.

5.1. Tagging word processing documents

XML is used as a portability standard and is used in the Microsoft .docx and other related formats. It is relatively easy to read the XML contents of the compressed .docx files, to store the contents in an XML database, and retrieve the contents back into a .docx file format. We have implemented tools to insert and extract these documents from an XML database. We can enhance the documents with the new security attributes by adding special XML attributes to subtrees in the XML document.

Our original goal was to explore the use of the XML security tags to enhance the security of the base document. Of primary interest to us is the ability to have multi-level documents supported by a system that can generated views of the document appropriate to a particular security level. We are able to use security tags to support this goal if the stored documents are in a read-only archive, tagged with security tags with a granularity as small as the paragraph level. Editing documents is harder, and will require additional tools.

One solution is to have an extraction tool automate the conversion of the XML security tags to traditional text tags at the beginning of the paragraph (e.g., a paragraph could start with [S//NOFORN]) to indicate secret no foreign national access. The tool that brings data back into the database can parse these tags and convert them to internal XML tags. We are exploring different mechanisms to utilize enhanced tags that may contain many attributes beyond this simple example.

Another solution maintains parallel document outlines that preserve the tags, and detects changes in the document as they are reinserted into the database, propagating changes and the new tags appropriately, in support of the security policy and polyinstantiation.

5.2. Derivative classification

Derivative classification refers to the reproduction or use of originally classified information in the course of producing other classified information [9].

One benefit of the XML tagging of documents is that we can use the security tags to assist in the appropriate tagging of new documents derived from existing documents. As part of our tagging research we have evaluated the use of tags for automating the derivative classification process [1].

Complete automation is still in the future, since it involves a contextual understanding of rules in the original documents classification guide and a mapping of those rules to the context of the new document. However, using the XML-based tags from our tagged word processing documents, we can assist in providing some of the context. To date this has been demonstrated through development of a mechanism for original classifiers to create a consistent rule file that can be used to automate the derivative classification process and is then incorporated into our XML tagging project as a filter for input into the database [1].

6. Conclusions

The objective of our on-going security tagging research is to evaluate different classes of security policies and different implementations of security tagging to develop a general understanding of the appropriate uses, strengths and limitations of security tagging technologies. This section summarizes some of the key findings to date.

6.1. Security tagging for a real-time OS

Our OS research involved development of a new tagging scheme and associated security policy, simulation and testing of that tagging scheme, and implementation of an operating system prototype that utilizes the tagging scheme [23][24][25][26].

We decided to utilize the RTEMS [2] operating system as a basis for a hardware tagging based operating system instead of starting from scratch. Utilizing RTEMS enabled us to avoid much of the lower level intricacies of a real system and focus on the security aspects of the tagging and its interactions with operating system core components.

Conclusions: It is possible to use tagging hardware to provide increased memory protection, separation and isolation in the operating system and among system applications. Enhanced tags can be used to type memory in terms of function entry points, executable code and data. The tags can also be used to limit control flow between functional units, and limit access to code modules, even in the same address space.

6.2. Security policy research

Prior research in policy enforcement mechanisms defines run-time solutions as meeting a set of enforceable security polices, which are a subset of all policies [12][17]. We found that a general purpose hardware-based tagging implementation can only enforce a subset of the run-time enforceable security
policies, those based on checks for type mismatches. Specifically, hardware based tagging is only aware of the security tags and associated domains that it supports. This mechanism can be used to support higher level security policies similar to how current microprocessors, which are not aware of different users, can still support separation between those users.

Conclusions: Hardware-based tagging can be used to detect type mismatches in memory access, control flow operations and machine code operations. The assignment of types to memory addresses and registers must be under the control of software. Therefore additional software, in the operating system or even at the middleware and application level, is needed to set the tags, and interpret errors generated by the hardware. This software will be more complex than traditional operating system memory protection software and will therefore require increased verification and validation.

Tagging hardware provides continual checks for type mismatches, relieving software of that burden, and providing greater confidence in the correct behavior of the system. However, care must be taken to not assume more functionality in the tagging hardware than really exists. Additional work is needed to understand the tradeoffs between tagging-supported security features and software-only based security features.

In addition, to provide strong assurance of run-time enforceable security policies we contend that all executable hardware of the system (Direct Memory Access controllers, co-processors, network cards, etc.) will have to conform to tagging principles, or will have to be isolated by tagging hardware.

6.3. Software-based tagging of documents

Our tagging research project is investigating several operational scenarios where data tagging could be used. For the purposes of this work, consider a web-based information system. A user enters a query into the system, much like a browser search term. The system then automatically generates a report based on the query. The report contains summary information as well as links to information objects (e.g., reports or just portions of reports, data repositories, charts, tables). The user can select a summary, or an object, and get data about the information (e.g., creation and modification dates, original source identification, data corroboration, and security data such as release controls, and security level). Each source of information is tagged with a set of attributes that includes this metadata. These attributes are used by the system search engine, in conjunction with the user’s security credentials, attributes and environmental attributes, to determine accessibility to the data. The system will also need to automatically determine the “derivative classification” of the generated data, since combinations of data from different sources can result in composite data of a higher security classification.

In addition to retrieving data, the user is able to add new data to the system, and to create their own reports by combining information from a variety of sources. The system must be able to propagate the metadata from the various sources, and derive the appropriate security classification of the data so that others can securely access the new data.

Conclusion: An enabling technology for this type of system associates metadata with each information object within a document, such as a MS Word document. The tags should be associated with each informational construct, such as sentences, paragraphs, tables, figures, etc. The technology must support the ability to store a multi-level document, and selectively release parts of it to a user who can edit it. The edited material must then be securely merged back into the stored document. Software based tagging using XML attributes can support such a system.

6.4. Future work

Additional research is needed to determine the best mapping of tagging features and tag data types to OS and application needs. A major drawback of enhanced tagging is the increased overhead of maintaining the tags in memory, and accesses to that memory concurrently with system data.

In addition, there is a lot of research needed for the use of tags in information infrastructure systems. Can a document management system store multi-level documents, or at least security tagged documents (tagged at the paragraph and figure/table level, or lower) in a way that provide users the ability to dynamically generate views of the system (as we do with modern web searches) with appropriately propagated and derived security tags?

7. Acknowledgements

This work was funded in part by AFRL contract FA8750-11-2-0047 and NSF grant DUE 1027409. The opinions and findings expressed in this paper are solely those of the authors and do not represent the position of the sponsors.

8. References


