Enforcing Authorization Policies using Transactional Memory Introspection

Rutgers University, DCS-TR-628, April 2008

Arnar Birgisson
School of Computer Science
Reykjavík University
arnarb07@ru.is

Mohan Dhawan
Department of Computer Science
Rutgers University
mdhawan@cs.rutgers.edu

Úlfar Erlingsson
School of Computer Science
Reykjavík University
ulfar@ru.is

Vinod Ganapathy
Department of Computer Science
Rutgers University
vinodg@cs.rutgers.edu

Liviu Iftode
Department of Computer Science
Rutgers University
iftode@cs.rutgers.edu

Abstract

Correct enforcement of authorization policies is a difficult task, especially for multi-threaded software. Even in carefully-reviewed code, unauthorized access may be possible in subtle corner cases. We introduce Transactional Memory Introspection (TMI), a novel reference monitor architecture that builds on Software Transactional Memory—a new, attractive alternative for writing correct, multi-threaded software.

TMI facilitates correct security enforcement by simplifying how the reference monitor integrates with software functionality. TMI can ensure complete mediation of security-relevant operations, eliminate race conditions related to security checks, and simplify handling of authorization failures. We present the design and implementation of a TMI-based reference monitor and experiment with its use in enforcing authorization policies on four significant servers. Our experiments confirm the benefits of the TMI architecture and show that it imposes an acceptable runtime overhead.

1. Introduction

Security enforcement mechanisms must be implemented with special care, since any flaw in their implementation may open the door to a malicious attacker. Because attackers may exploit exceptional state transitions, enforcement must be correct even on uncommon code paths, for unusual interleavings of execution, or during abnormal error conditions. Experience shows that achieving these goals is challenging, especially for multi-threaded software [38, 60]. This paper introduces *Transactional Memory Introspection*, an architecture that can significantly simplify the task of correctly implementing security enforcement mechanisms.

Our work is based on *Software Transactional Memory* (STM), a technique for declarative concurrency control (*e.g.,* [33, 34, 36, 51]). STM techniques are an active area of research, especially in connection to multi-core trends in hardware concurrency [43]. With STM, a runtime system ensures that code sections have effects as if they were executed in serial order; typically, those code sections are marked using language-level annotations such as `atomic` or `transaction`. STM constrains concurrency without relying on error-prone locking and provides attractive guarantees, such as consistent recovery from failures. Although current STM systems incur high costs, hardware acceleration and language support may soon provide competitive overhead [43]. Therefore, in the near future, much software may be written using STM techniques.

Transactional Memory Introspection (TMI) is a reference monitor architecture that builds on STM semantics and machinery. TMI allows security enforcement to benefit from STM guarantees, yet remains compatible with existing security mechanisms, such as those based on history- or state-based enforcement [2, 18, 31]. TMI helps ensure that enforcement remains correct, even in corner cases. In particular, TMI-based enforcement needs neither consider concurrent interleavings of execution nor worry about remedial steps on policy violation.

Notably, TMI changes how security enforcement is integrated into software functionality. TMI builds on the precise bookkeeping, or *read/write sets*, that STM runtime system maintain to detect read/write conflicts for concurrent executions. A TMI reference monitor is implicitly invoked whenever a security-relevant shared resource is accessed in a transaction—*i.e.,* on all changes to security-relevant memory objects in the STM read/write sets—as well as when transactions commit. Application-specific security checks ensure security-policy compliance, *e.g.,* that all security-relevant accesses comply with an authorization policy. A transaction is aborted unless all such security checks are successful.

By triggering security checks on resource accesses, TMI can ensure complete mediation and can also avoid exceptional control paths and other complexities arising from explicit security checks. Furthermore, TMI can reduce the latency and performance overheads of expensive security checks, such as group membership tests. Because security checks need not be fully evaluated until a transaction attempts to commit, costly security-policy evaluation can be performed lazily, or in parallel with execution.

The TMI architecture is practical. We have designed and implemented a TMI reference monitor for enforcing authorization policies on server software. Our TMI implementation is based on Sun's Dynamic Software Transactional Memory (DSTM2) toolkit for Java [35]. In our design, TMI-based enforcement can integrate the functionality of existing authorization frameworks; we have performed such integration with Java stack inspection [31] and XACML [24].

We experimented with TMI-based enforcement of authorization policies on four servers, comprising a total of nearly 55,000 lines of code, converted to make use of DSTM2. Our experiments confirm that the TMI architecture can help ensure the correct enforcement

```
a1.   dispatch_request ( ) {
a2.       ...
a3.       perform_request ( );
a4.       ...
a5.   }

a6.   perform_request ( ) {
a7.       ...
a8.       if (allowed(principal, resource1, access1))
a9.           perform_access1(resource1);
a10.      else handle_failure1( );
a11.      ...
a12.      if (allowed(principal, resource2, access2))
a13.          perform_access2(resource2);
a14.      else handle_failure2( );
a15.      ...
a16.  }
```

(a) Current practice in authorization enforcement: Embeds reference monitor invocations in application code. Presents difficulties in (1) identifying resource accesses and ensuring complete mediation; (2) eliminating time-to-check to time-of-use bugs (in lines a8/a9 and lines a12/a13); and (3) correctly handling authorization failures.

```
  b1.   dispatch_request ( ) {
  b2.       transaction [ principal ] {
  b3.           ...
  b4.           perform_request ( );
  b5.           ...
★ b6.       }   /* Commits only if all authorization succeeds */
  b7.   }

  b8.   perform_request ( ) {
  b9.       ...
★ b10.      perform_access1(resource1);
  b11.      ...
★ b12.      perform_access2(resource2);
  b13.      ...
  b14.  }
```

Authorization manager, implicitly invoked by TMI on ★ lines:

```
    switch (resource access pattern)
        case (resource=R, access_type=A) ⟶
            if (¬allowed(principal, R, A)) then abort_tx;
```

(b) TMI-based authorization enforcement: Decouples application functionality from policy enforcement. (1) Ensures complete mediation of all resource accesses via introspection on memory-access bookkeeping performed by the STM runtime; (2) prevents race conditions by construction; and (3) allows simple handling of authorization failures via rollback.

Figure 1. A comparison of traditional and TMI-based enforcement of authorization policies.

of security policies, and that it can have acceptable enforcement overhead. We also found that retrofitting STM techniques and TMI-based authorization to existing server software was a manageable task; most of the work was due to DSTM2 limitations that should disappear in future, more mature, STM systems. Based on our experiments, we conclude that the TMI architecture is a useful new alternative for authorization-policy enforcement, and that it can be widely applicable, even to existing software.

In summary, the main contributions of this paper are:

- **Transactional Memory Introspection.** We introduce TMI, a new architecture for implementing reference monitors. We describe its components and applicability, and show that it can ensure complete mediation, freedom from TOCTTOU bugs, and can simplify handling of errors and authorization failures.
- **An implementation of a TMI reference monitor.** We show that TMI-based enforcement can be practically implemented and can integrate and build on existing security mechanisms and frameworks. We present one particular TMI implementation, suited to the enforcement of authorization policies.
- **Experimental validation of the benefits of TMI to security enforcement.** We have retrofitted server software with a TMI reference monitor in a way that integrates with the software's existing security mechanisms (*e.g.,* XACML or Java stack inspection). Our experiments show that adopting TMI-based enforcement can be straightforward, and that this results in simpler and less error-prone code. We measured an acceptable, average overhead of less than 11% for TMI-based enforcement of authorization policies.

The remainder of this paper is as follows. In Section 2, we explain the benefits of TMI for enforcing authorization in server software; we also give background material on STM techniques. In Section 3, we present the TMI architecture, give a concrete example of its application, and discuss enhancements. In Section 4, we present an implementation of TMI, based on a modified DSTM2, an object-based STM system for Java. In Section 5, we present our experiments with the enforcement of authorization policies in server software. Finally, we discuss related work in Section 6 and summarize in Section 7.

2. Motivation and background

In this section, we first present our focus application—enforcement of authorization policies in server software—and explain how TMI-based enforcement can help overcome many of the difficulties in the implementation of such enforcement. We then present background material on transactional semantics and STM techniques.

2.1 Benefits of TMI-based authorization in server software

Server software must protect shared resources from inappropriate client access by formulating and enforcing an authorization policy. Such policies specify, for each shared resource, what principals can perform which operations. At runtime, a reference monitor should ensure that each access to a shared resource is authorized.

Unfortunately, as prior work shows, integrating authorization enforcement into server software is time-consuming and error-prone. For example, it took almost two years to add invocations of the Linux Security Modules (LSM) reference monitor to the Linux kernel [56]. Similar recent attempts to enforce authorization policies in the X11 server [41, 54], the JVM [26], and IBM Websphere [37] have also become time-consuming, multi-year efforts.

To understand the key difficulties in the current practice of enforcing authorization policies, consider Figure 1(a), which shows pseudo-code from a server. This server accesses resources on behalf of a client (on lines a9 and a13). As an example, if this server is a chat server, the accesses may correspond to adding a user to a chat forum. Because a chat forum is a shared resource, accesses to modify the forum must be authorized, *e.g.,* to prevent users from joining private forums or forums where they are blacklisted.

As shown in Figure 1(a), authorization is typically enforced by embedding reference monitor checks with server functionality, using a programming language pattern such as if...then...else. Resource accesses are performed conditional on a predicate that checks whether the access is permitted (lines a8 and a12).

If the access is denied, an error handler is executed (lines a10 and a14) to perform any remedial steps necessary to restore the software into a consistent state. In this current practice of authorization policy enforcement, three major difficulties must be overcome:

- **Difficulty in completely mediating access to resources.** Two properties must be ensured for complete mediation of access to shared resource. First, each access must be checked and authorized. Second, each call to the reference monitor must provide the correct security-relevant metadata, such as the operating principal, the identities of accessed resources, and the types of access (in the chat server example, respectively, the user attempting to join, the chat server forum, and the operation of joining a forum).

  In current practice, ensuring these two properties is a challenge. First, the locations for authorization checks that guard each resource access (*i.e.,* lines `a9` and `a13`) are currently identified manually. This process can easily fail to identify resource accesses, especially along uncommon, easy-to-overlook code paths. For instance, consider Figure 2, which shows (simplified) code snippets from the Linux kernel. Both `vfs_read` and `page_cache_read` read the contents of a file object (the lines in bold font). However, the function `page_cache_read` does not check for file permissions, since it expects them to have been checked elsewhere. This omission may lead to an unauthorized read from a file: Zhang *et al.* [60, Figure 10] found a case where an unchecked file object is used by `page_cache_read` upon a page fault from a memory-mapped region.

```
ssize_t vfs_read(struct file *file,...) {
      ...
      if(check_permission(file, MAY_READ) == ALLOWED) {
            file->f_op->read(file,...);
      }
      ...
}
int page_cache_read(struct file *file,...) {
      struct address_space *mapping =
            file->f_dentry->d_inode->i_mapping;
      ...
      mapping->a_ops->readpage(file,...);
      ...
}
```

Figure 2. An example showing violation of complete mediation.

  Second, security-relevant metadata, such as the permissions to check for authorizing access, are also typically identified manually and hard-coded into server software. Such decentralized, ad hoc checking is highly error prone, especially as code is changed over time. Not surprisingly, Jaeger *et al.* [38, Pages 193-196] found several inconsistencies in the file-access permissions checked on different code paths in the Linux kernel.

- **Difficulty in preventing Time-Of-Check To Time-Of-Use race conditions (TOCTTOU bugs).** With the possibility of concurrent execution, the current practice of authorization enforcement shown in Figure 1(a) becomes even more problematic. In multi-threaded server software, the resource accesses authorized on lines `a8` and `a12` must still be valid at lines `a9` and `a13`, respectively, for all possible execution interleavings. Otherwise, an attacker may be able to maliciously exploit the resulting race condition, or TOCTTOU bug [7]. For instance, in the chat server example, if the forum is public when the authorization check is performed, then there must be no way for the forum to become private before a user joins; otherwise, an unauthorized user may be able to join a private forum.

  It is particularly difficult to prevent TOCTTOU bugs when enforcing authorization in efficient, multi-threaded server software, written using modern, modular techniques. Unless all code paths are accounted for, attackers may be able to induce context switches between authorization and access, and perturb shared state in ways that violate the security policy. For instance, such TOCTTOU bugs were found in the analysis of the LSM-protected Linux kernel [60].

- **Difficulty in correctly handling authorization failures and other errors.** Server software must continue to function despite errors and authorization failures due to one client. In Figure 1(a), this is the task of the error handlers (lines `a10` and `a14`), which must return the server software back into a consistent state. Depending on the service, this may involve executing a complex, uncommon error path, and performing compensating actions, *e.g.,* to undo other, previously-authorized operations related to a single server request [2, 8, 55]. If the security policy that is being enforced is stateful, the state of the enforcement mechanism may also have to be wound back. For example, if a chat server user requests to join a forum, and processing this request involves several steps, then—if the request is eventually not authorized—an error handler may need to undo all of the chat server state changes due to the processing of that request.

  Several studies show that error-handling code can be a large fraction of server software; one IBM survey reports error handling to be up to two-thirds of code [12]. Authorization failures and security exceptions account for a large fraction of errors, and they are no easier to handle than other errors [6, 25]. Much software simplifies this problem by treating all errors equally: by either ignoring them or by halting execution [55]. However, server software must correctly deal with the corner cases resulting from errors and authorization failures. In the current enforcement architecture shown in Figure 1(a), this results in error-prone and difficult-to-maintain software.

The TMI architecture helps avoid the above difficulties by decoupling security enforcement from application functionality in software that uses STM techniques. For this, TMI requires that accesses to security-relevant, shared resources be enclosed within a `transaction`{...} code section, as shown in Figure 1(b). Additionally, TMI requires an application-specific *authorization manager* that provides the security checks to be performed upon each access to a security-relevant resource.

As the server executes, all accesses to shared resources within a transaction are precisely monitored for the read/write sets of the STM runtime system. A TMI reference monitor builds on this bookkeeping to trigger authorization checks. Specifically, in Figure 1(b), resource1 and resource2 are accessed within a `transaction`{...} code section. Using TMI-based enforcement, those accesses will trigger authorization-manager-specified security checks corresponding to those in lines `a8` and `a12` of Figure 1(a). With TMI, if a transaction is to commit, all such authorization checks as well as a final transaction validation must succeed.

TMI can guarantee complete mediation by ensuring that the reference monitor is implicitly invoked whenever shared, security-relevant resources are accessed within a `transaction`{...} code section. In particular, with TMI, a developer need no longer identify and guard individual resource accesses as in Figure 1(a). Rather, as shown in Figure 1(b), `transaction`{...} code sections may span multiple resource accesses and method calls; in server software, such a code section may naturally encompass the code that dispatches and handles client requests.

TMI also simplifies failure handling: authorization failures simply cause a transaction abort, thereby reverting back into the consistent state at the start of a transaction. Thus, a server can be assured that upon an authorization failure, a request will have no effect, and the server will be able to give a default response indicating that access is denied. With TMI, the reference monitor state itself can also

```
acquire (S1.lock);
acquire (S2.lock);
value = S1.pop();
S2.push(value);
release (S2.lock);
release (S1.lock);
```

(a) Lock-based programming

```
transaction {
        value = S1.pop();
        S2.push(value);
}
```

(b) STM-based programming

Figure 3. A comparison of code for atomically moving a data item between two stacks.

have transactional memory semantics, which can further simplify correct failure handling for stateful authorization policies.

For multi-threaded software TMI can eliminate TOCTTOU bugs, by construction, by building on the STM serialization of `transaction{...}` code sections. Furthermore, TMI can also allow lazy or concurrent evaluation of security checks, such as the expensive, high-latency group-membership tests often used to check authorization. This enhancement can significantly reduce overheads, as shown in Section 5; it is discussed further in Section 1.

Because TMI reference monitors are integrated directly with an STM runtime system, they can perform introspection to determine security-relevant information, such as old or new data values, access types, or other authorization metadata. Such introspection also allows TMI-based enforcement of history-based or data-dependent authorization policies, as well as the enforcement of security policies based on well-formed transactions [14].

2.2 Background on software transactional memory

As defined by the database community [32], a transaction is a sequence of actions that must satisfy the ACID properties: Atomicity requires that all actions complete successfully, in their totality, (a transaction commit), or that none of them have any visible effects (a transaction abort). Consistency requires transactions to maintain application-specific data invariants that hold before transactions. Isolation requires transaction to give the same result, irrespective of other simultaneous transactions. Finally, Durability requires that the data changes of a committed transaction must be persistent and visible to all subsequent transactions.

STM techniques aim to ease the writing of correct, multi-threaded programs by providing an abstraction for transactional access to shared-memory data. Many STM systems extend a programming language with new code sections, identified by a special keyword (often `atomic`). These code sections execute with transactional semantics as defined above, except that memory is still transient (the D from ACID is missing). In this paper, we denote atomic code sections using `transaction{...}`, to avoid confusion with any given STM proposal.

To see the benefits of STM transactions as a programming abstraction, consider the example shown in Figure 3 (adapted from [43]). In this example, data is popped off a first stack, `S1`, and pushed onto a second stack, `S2`. If executed concurrently, this code must simultaneously synchronize access to both stacks—otherwise, execution interleaving may expose abnormal states, *e.g.,* states where `data` is on neither stack. Conventionally, this synchronization would be achieved as shown in Figure 3(a): by acquiring dedicated locks for `S1` and `S2`, before performing the `pop()` and `push()` operations.

Unfortunately, programming using locks is error-prone. The programmer must ensure that *all* the required locks are acquired; otherwise, a race condition may be possible. The programmer must also ensure that locks are always acquired in a correct order; otherwise, a deadlock may be possible. Thus, lock-based programming raises hard-to-answer questions, such as: Which locks must be acquired? In what order? Does this lock acquisition cause deadlock somewhere else? Programming with locks is especially tricky in large, complex software, with many layers of indirection and abstraction, or when dealing with third-party libraries, which may not expose their locks or their locking regime.

In contrast to locks, with STM, the programmer need not specify how concurrency control is achieved: automatically, each declared `transaction{...}` code section will execute atomically. In particular, STM frees the programmer from having to identify the resource instances for which accesses must be synchronized. Thus, the code in Figure 3(b) can provide the same functionality, yet be significantly simpler and less error-prone than the code in Figure 3(a). STM requires significant runtime support to make `transaction{...}` code sections atomic. An alternative is to use recently-developed techniques [11, 17] that use `transaction{...}` annotations to automatically infer the locking protocol. However, these techniques rely on whole-program analysis and are unsound in the presence of library code.

STM systems must be able to detect and resolve runtime conflicts between different transactions. There is great variability in the implementation of STM runtime support. For instance, implementations may use compiler support [3], support from software libraries [35], or hybrid schemes that combine hardware and software [19, 23, 49]. Similarly, STM systems also differ in the granularity at which they detect conflicts: *word-based* STM tracks individual memory words (these include most hardware STM systems), while *object-based* STM tracks language-level objects (these include most language-based STM systems). A comprehensive discussion is beyond the scope of this paper; the book by Larus and Rajwar gives a good overview of much recent research [43].

Any STM runtime system must track the data dependencies of transactions, *i.e.,* what data is read, as well as what data is written. The STM runtime system must *validate* that the sets of those reads and writes (the *read/write sets*) are not in conflict with other transactions; a *conflict* occurs when, concurrently, the same memory object is used by multiple transactions, and at least one transaction changes the value of that memory object. An STM runtime system that considers only conflicts between accesses within transactions can provide *weak atomicity*; STM systems that also detect conflicts from non-transactional activity, and provide *strong atomicity*, may incur greater costs and are less common [1, 34, 52]. Validation can happen eagerly or lazily, as long as each access is validated before a transaction commits. If validation detects conflicts, the STM runtime system consults a *contention manager* to decide which transaction to commit. To allow other, conflicting transactions to be aborted, the STM runtime system must also provide *rollback* mechanisms that undo the execution effects of those transactions. Finally, STM systems may support *nested transactions*, or other transaction composition.

For software that uses declarative concurrency control, existing STM mechanisms already perform the majority of the work needed for TMI-based enforcement. Thus, the adoption of TMI-based security enforcement is likely to add little in terms of complexity, mechanism, or performance overhead.

3. Transactional memory introspection

In this section, we describe the TMI architecture and its use in enforcing authorization policies. We also show a concrete example of TMI-based enforcement and discuss enhancements.

3.1 The TMI architecture

The TMI architecture aims to raise the level of abstraction in the implementation of security enforcement mechanisms. It does so by decoupling application functionality from security enforcement code, much as STM techniques decouple applications from concerns about lock acquisition order. TMI-based enforcement can

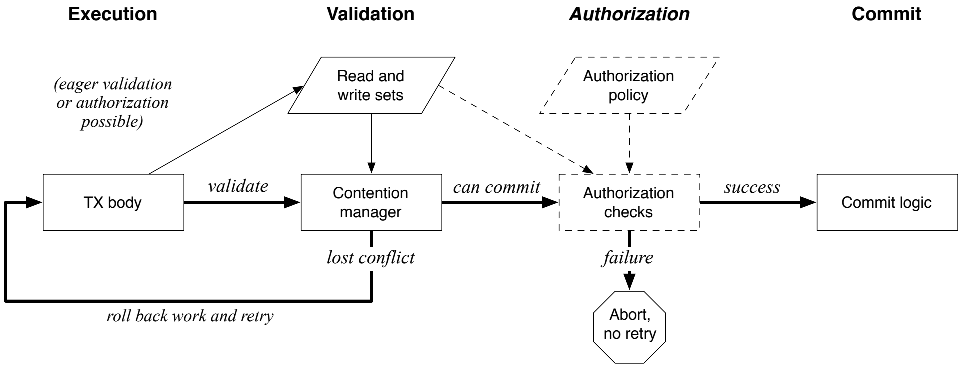

Figure 4. An overview of a TMI reference monitor. The solid boxes are standard components of STM systems, while the dotted boxes show components added by the TMI architecture.

thereby eliminate concerns about check placement, race conditions, and exceptional execution paths.

A TMI reference monitor observes the execution of transactions in an STM system. Each transaction is associated with a specific principal; this principal may be given as an explicit argument to `transaction`{...} code sections, as shown in Figure 1. The TMI reference monitor is invoked, implicitly, whenever a security-relevant, shared resource is accessed, so that compliance with the security policy can be checked. The TMI reference monitor is also invoked at the end of a transaction to ensure that only policy-compliant transactions are allowed to commit.

With TMI, the application software defines the security policy, the set of resources that are security-relevant, as well as the code that checks security-policy compliance. In particular, when TMI is used for authorization in a server, the server code must identify the resources for which access must be authorized. Depending on the underlying STM system, this may require code changes, such as annotations. The server must also define an authorization manager that contains code for security checks; upon each access to a security-relevant resource, the TMI reference monitor consults the authorization manager. The authorization manager may evaluate application-specific expressions and maintain its own security state. Furthermore, TMI supports introspection on security-relevant information, such as the old or new values for a memory access. Thus, a TMI reference monitor can observe a rich trace of execution activity, and evaluate predicates on that trace; this allows the enforcement of most practical security policies—in particular, history-based policies and other EM-enforceable policies [50].

Figure 4 shows how a TMI reference monitor extends the mechanisms of an STM runtime system. As transactions execute, the STM runtime system tracks accesses to shared resources in its read/write sets. All accesses within the scope of a `transaction`{...} code section are tracked, including those that happen indirectly (*e.g.,* through deeply-nested method calls). To ensure complete mediation, the TMI reference monitor is invoked whenever a security-relevant resource is added or updated in the read/write sets. For some STM systems, a language-level type, or annotation (*e.g.,* `@atomic` in DSTM2) is sufficient to guarantee these TMI reference monitor invocations. Alternatively, security-relevant resources may be placed in the read/write sets of concurrently-executing, dummy transactions, so that they trigger a conflict. (A similar technique is used to trigger efficient retry in some STM systems [33].)

As seen in Figure 4, TMI extends the validation step for STM transactions to also enforce correct authorization, or general compliance with security policies. With TMI, a transaction is aborted if it performs unauthorized resource access or otherwise violates the security policy. (Alternatively, certain authorization failures such as those due to a timeout, might trigger a retry, although we have not pursued that option.) Upon an abort, the STM runtime system will roll back all effects of a transaction, including any changes to the state of the TMI reference monitor. Thus, the application is restored into the same consistent state as before the transaction, apart from an error code that is returned to the application to indicate the security violation.

The results of security checks must be established before transactions attempt to commit. However, security checks need not be fully evaluated when the TMI reference monitor is invoked; the bulk of the work can often be performed lazily, or even overlapped with the execution of the transaction. We have used both lazy and overlapped enforcement of authorization policies for our experiments in Section 5, and found that they can significantly reduce overheads.

The TMI reference monitor is invoked implicitly upon access to security-relevant resources, as described above. However, there are cases when code within an STM transaction must explicitly query the security policy. For example, a tar utility that archives all the files in a directory might need to create a consistent archive, even when one file is not accessible. Therefore, for such exceptional cases, the TMI architecture supports immediate evaluation of security policy, without the risk of a transaction abort. Such support can be implemented in many ways, *e.g.,* with a `try`...`catch` block in the application, or by exposing a direct, Boolean authorization query interface to application software.

The underlying STM system, and its language integration, has an influence on aspects of TMI-based enforcement that range from efficiency to applicability. For example, most of the performance overheads that we measured in our experiments in Section 5 were due to inefficiencies of the library-based DSTM2 system. Also, as an extreme data point, TMI may not be not compatible with some limited, hardware-only transactional memory proposals, such as [20, 21]. More commonly, TMI-based enforcement must take into account any gaps in complete mediation that may arise as a result of the limited, weak atomicity provided by most STM systems. For example, in server software, this may entail placing `transaction`{...} code sections around all handling of requests, since any request processing might access a security-relevant, shared resource.

A TMI reference monitor is best built on an STM system with comprehensive support for I/O and external state, since security policies often aim to constrain externally-observable effects. For example, some STM systems extend transactional semantics to I/O in `transaction`{...} code sections using the support for distributed transactions provided by an increasing number of I/O systems (*e.g.,* the NTFS file system [45] and MySQL database [44]). Techniques for extending STM semantics to I/O and external state are an active area of research [22, 43, 48]. For this paper, we have extended an STM implementation to provide transactional file and database I/O, using off-the-shelf transactional I/O packages (*e.g.,* [39, 44, 45]).

3.2 Example of TMI-based authorization policy enforcement

As a concrete example of using a TMI reference monitor to enforce authorization policies on client requests, consider Figure 5, which shows a snippet of code taken from GradeSheet, a Java-based multi-threaded server to manage student grades. For our experiments in Section 5, we modified GradeSheet to make use of the object-based STM system DSTM2.

The key data structure in GradeSheet is `sheet`, a two-dimensional array of `GradeCell` objects, each of which stores the details of a student, a project, and a granted grade or an average grade. The `sheet` table can be accessed by multiple principals and each access must be properly authorized. For instance, GradeSheet may enforce that a teaching assistant can only access/modify the student grades in projects that she supervised. GradeSheet parses commands issued by clients in the top-level function `doRequest`, which dispatches requests; Figure 5 shows how `getGrade` and `setGrade` are dispatched and access the `sheet` table.

The GradeSheet code must be changed in three ways to make use of DSTM2 and TMI-based enforcement; Figure 5 shows the relevant code in bold. First, GradeSheet code that may access shared resources is wrapped in a `transaction`{...} code section that correctly identifies the acting principal. Second, with TMI-based enforcement, server code must correctly handle a transaction abort because of authorization failure; for GradeSheet, no code change was required, since `doRequest` already indicated failure with `return null`. Third, DSTM2 requires shared state to be especially marked, hence the `@atomic` annotation before the class definition of `GradeCell`.

Aside from these changes, a separate authorization manager also must also be provided to enable TMI-based enforcement. We have specified GradeSheet authorization policy using XACML; the policy details are given in Section 5.1. Parts of the authorization manager are shown in the lower half of Figure 5. The TMI reference monitor is invoked on each read or write of a `GradeCell` object, since those objects are the security-relevant resources. The GradeSheet authorization manager specifies how `xacml_getdecision` checks must be invoked based upon the type of access that was performed. For example, a read access to the `grade` field indicates a that the `getGrade` security-relevant operation is being performed, which triggers the relevant check in the authorization manager.

3.3 Enhancements

The basic TMI reference monitor architecture presented above can be augmented in several ways, as discussed below.

(**1**) **Eager, lazy and overlapped enforcement.** With *eager enforcement*, authorization checks happen immediately upon each security-relevant update to the STM read/write sets of a transaction; each authorization check must be fully completed before the transaction continues execution. Explicit authorization queries, such as described in the tar utility example on page 5, Section 3.1, always trigger eager enforcement.

In contrast, with *lazy enforcement*, only the inputs to authorization checks are evaluated, or copied, when the TMI reference monitor is invoked. Validation and authorization happens at the end of the transaction, when all security-relevant operations performed during the transaction are authorized en masse. If any of the operations is not authorized, the entire transaction fails.

With *overlapped enforcement*, each transaction may spawn an auxiliary thread to perform policy evaluation. Subsequently, when the transaction performs a security-relevant operation, the inputs for authorization checks are dispatched in a message to this auxiliary thread, which then performs the authorization checks concurrently. During validation, transaction execution joins with the auxiliary thread, and the transaction is aborted if any authorization checks failed. Our experimental evaluation in Section 5 shows that overlapped enforcement is effective in improving performance when both the transaction body and policy evaluation have high latency, or are computationally expensive.

With lazy and overlapped TMI-based enforcement, transactions are speculatively executed with optimistic assumptions. Like other speculative security enforcement, such as that of [46, 59], this may benefit performance but may also expose new side channels and increase the risk of leaking information. (Also, like all substantial runtime mechanisms, the STM itself system may add new covert channels.) For example, given an STM runtime system that provides weak atomicity and updates memory in-place (*e.g.,* that in [3]), lazy and overlapped TMI-based enforcement will expose information to non-transactional activity. Thus, lazy and overlapped TMI-based enforcement is more suited for policies for integrity, auditability, *etc.*, than for policies where confidentiality is critical.

(**2**) **Stateful authorization policies.** In stateful authorization policies, such as those expressed using security automata [18, 50], each security-relevant operation potentially alters the state of the policy. Enforcing such policies requires two enhancements to the basic design of the TMI reference monitor.

First, the order in which security-relevant operations happen decides how the state of the policy changes, and must therefore be recorded. This is achieved by building TMI an *introspection log* on top of the STM read/write sets. Each read/write to an object in a transaction is added to the end of this log, thereby preserving the order in which these operations happen. During enforcement, the introspection log is used to update the state of the authorization policy. Second, for stateful policies, it is also important to have transactional semantics on the state of the policy. This is because upon an authorization failure, the state of the policy may also have to be restored.

(**3**) **Fingerprints.** A security-relevant operation on a resource may often consist of a several low-level accesses to that resource. A *fingerprint* maps each such security-relevant operation to the low-level resource accesses that constitute the operation [28]. For example, in FreeCS, a chat server that we evaluated (Section 5.3), the security-relevant operation of a user joining a chat room involves reading the field `usrList` of an object called `Group`, followed by writing to a field `grp` of an object called `User`. Prior work presented techniques to automatically mine such fingerprints by analyzing server source code [28].

For each security-relevant operation that consists of multiple resource accesses, we supply the TMI reference monitor with its fingerprint. The reference monitor matches the resource accesses specified in the fingerprint against updates to the read/write sets to determine when security-relevant operations are performed during the transaction. In some cases, such as the FreeCS example discussed above, the fingerprint may consist of a *sequence* of resource accesses (rather than a *set*). Thus, the TMI reference monitor must be extended to support an introspection log, and fingerprints must be matches against this log to determine when a corresponding security-relevant operation is performed.

4. Implementation

We implemented a TMI reference monitor by extending Sun's Dynamic Software Transactional Memory (DSTM2) package [35].

```
/* @atomic */ class GradeCell {
    StudentID sid;
    ProjectID pid;
    int grade;
};
/* Other class declarations omitted for brevity */

GradeCell[][] sheet;

String getGrade(int s, int p)
{
    return "" + sheet[ s ][ p ].grade;
}

String setGrade(int s, int p, int g) {
    sheet[ s ][ p ].grade = g;
    return "" + g;
}
```

```
/* Server handling of a command for a client principal */

String doRequest(Object principal, String command)
{
    transaction[ principal ] {
        Tokenizer st = new Tokenizer( command );
        String action = st.nextToken();
        int student = st.nextInt();
        int project = st.nextInt();
        if (action.equals("getGrade")) {
            return getGrade(student, project);
        } else if (action.equals("setGrade")) {
            int grade = st.nextInt();
            return setGrade(student, project, grade);
        } ...
    }
    return null;
}
```

XACML GradeSheet authorization manager: if `xacml_getdecision` evaluates to "permission denied", TMI aborts the transaction.
switch (resource access pattern)
case (resource=`GradeCell G`, field=`grade`, access_type=read) ⟶ `xacml_getdecision(principal, G, getGrade);`
case (resource=`GradeCell G`, field=`grade`, access_type=write) ⟶ `xacml_getdecision(principal, G, setGrade);`

Figure 5. Code fragment showing the changes needed for TMI-based policy enforcement in GradeSheet.

DSTM2 provides a framework for Java object-based STM systems, along with some concrete STM runtime mechanisms. DSTM2 supports STM fundamentals but not uncommon features (*e.g.,* nested transactions). Thus, DSTM2 mechanisms track read/write sets, perform validation and contention management, and commit/abort transactions. In particular, DSTM2 contains two mature, substantial STM mechanisms for detecting and resolving conflicts (the *obstruction-free* and *shadow* `@atomic` object factories).

Our implementation is compatible with any STM mechanism that fits into the DSTM2 framework; in particular, we have applied TMI-based enforcement with several different DSTM2 contention managers and both types of `@atomic` objects. Our implementation also allows TMI reference monitors to make use of other, existing security mechanisms; for our experiments, we have integrated TMI with XACML [24] and Java Stack Inspection [31].

Our implementation extends DSTM2 by interposing on updates to read/write sets and on the transaction commit/abort logic. A TMI reference monitor is invoked at the start and end of transactions (just before commit or abort), as well as on each access that may change a transaction's read/write set (*i.e.,* on all accesses to `@atomic` objects). These invocations trigger the application-specific authorization manager, which filters out access to security-critical objects and performs authorization checks. If an access is not authorized, the transaction is aborted and an `AccessDeniedException` is thrown. (This is the only exception a transaction can throw: for other exceptions, DSTM2 implements a fail-stop model.)

In our implementation, most authorization checks can be deferred by copying all security-relevant metadata into an *introspection log*. Security-relevant information can change during a transaction, so the introspection log must contain immutable copies of this metadata for input to authorization checks. Our implementation supports both lazy-until-commit and overlapped enforcement of deferred authorization checks. (Explicit, functional checks may not be deferred, as discussed in Section 3.1, and are always eagerly evaluated.) In our experiments, we have also used the introspection log to enforce history-based authorization that detects the fingerprints of security-relevant operations.

We also extended DSTM2 to support transactional I/O, by adding to the commit/abort logic support for two-phase distributed transactions [32]. Using this support and the Apache Commons library [39], we extended STM semantics to file I/O. We also added partial DSTM2 support for transactional modifications to back-end databases using `java.sql.Connections`.

Overall, not counting library code, our implementation adds less than 500 lines of Java code to DSTM2; each application-specific authorization manager is between 100 and 200 lines. We also created `@atomic` versions of standard Java data structures and containers; these, and other modifications comprise a few thousand lines of code changes.

Our TMI reference monitor implementation applies only to DSTM2 server software—just as the TMI architecture applies only to software that uses declarative concurrency control. Because no such software existed, we first had to retrofit DSTM2 onto server software in order to experimentally validate our implementation. This porting involved three changes (the first two, substantial changes would not be required with a language-integrated STM system, such as [3, 33]). First, all shared objects had to be identified, and their class replaced with an equivalent, transactional class, annotated with the DSTM2 `@atomic` keyword. For this, we had to implement `@atomic` versions of common data structures, such as `java.util.HashMap` and `java.util.Vector`; for simpler data structures, containing only `@atomic` objects, scalar values, or strings, we could use DSTM2 support for automatic generation of `@atomic` classes. Second, the reads and writes of fields in `@atomic` objects had to be changed to use DSTM2 accessor functions. Third, and most simply, a `transaction`{...} block had to be introduced around the handling of client requests, as well as other code that accesses shared resources.

Given the above modifications, adding TMI-based enforcement of authorization policy was easy for the server software in our experiments. First, the principals had to be identified and exposed for each `transaction`{...} code section. (In our implementation, principals are transaction arguments, as shown in Figure 5, where the `principal` variable contains the principal.) Second, an application-specific authorization manager had to be written for instantiation at the start of server execution. This task primarily involved understanding the server's authorization policy, the server's security-relevant resources, and what metadata to copy into the introspection log. Finally, for some experiments, we had to interface the authorization manager with external security mechanisms (namely, XACML [24] and Java Stack Inspection [31]).

5. Evaluation

This section reports on the retrofitting of four servers, comprising nearly 55,000 lines of Java code, with TMI-based authorization using our DSTM2-based implementation. Our intent with these experiments was to evaluate whether the TMI benefits held in practice.

The results of our evaluation confirm that TMI-based enforcement is practical, and can be easily adopted by STM servers to facilitate the writing of simple, correct authorization code. Furthermore, our results confirm that TMI can be integrated with existing security mechanisms, that TMI has acceptable enforcement overhead, and that TMI can adapt enforcement and overhead to each workload—in some cases allowing absolute performance improvements over traditional authorization.

5.1 GradeSheet: A simple grade management system

As discussed in Section 3.2, GradeSheet is a simple client/server Java application to manage student grades, containing about 900 lines of code. Principals are either graders (professors or TAs) or students. GradeSheet enforces the following authorization policy: (1) a professor may read/write all grades and read all grade averages; (2) a TA may read/write grades for projects that she supervised and read any project's grade average; and (3) a student can only read her own grades and project grade averages.

We ported GradeSheet to use TMI-based authorization policy enforcement and converted all shared objects to their DSTM2 `@atomic` equivalents. `GradeCell` objects must be authorized based upon their security-relevant attributes, namely grades, student IDs and project IDs. We integrated the TMI authorization module with both a custom-built policy engine as well as another one that used XACML [24] policies.

5.2 A Tar archive service

We experimented with a 5,000 line Java service that allows standard Tar archives to be created and processed [53]. We converted this code to use TMI-enhanced DSTM2, and perform each service invocation within a transaction. Few lines were changed in this conversion: we used a simple, static escape analysis to establish that most state was transaction-local, and that no `@atomic` objects were required.

Subsequently, we added TMI-based enforcement of file-system authorization policies to the converted Tar service. We implemented this enforcement with an authorization manager that is also installed as, and inherits from, the Java `SecurityManager` [31]. Thus, our TMI reference monitor accurately models system-level access control in Java.

In particular, our TMI reference monitor is invoked whenever any files are opened for reading or writing. As before, it can perform authorization checks lazily or in an overlapped fashion, by copying into an introspection log any security-relevant metadata, including the Java stack-based security context.

We have used our implementation for lazy and overlapped enforcement of existing Java stack inspection security policies. We used `java.security.AllPermission`, the simplest policy available for Java stack inspection. Coupled with overlapped enforcement, more complex policies would amplify the trends shown in our experiments. In particular, potentially more work could be performed in parallel with the main execution.

5.3 FreeCS: A chat server

FreeCS is a Java-based chat server that consists of about 22,000 lines of code [27]. FreeCS allows its users to broadcast messages; a message broadcast by a user is visible to all other users in the same group (FreeCS's equivalent of a chat room). A user can issue several commands via a FreeCS interface, including commands to join a new group, invite other users to her group and ban members from her group; in all, FreeCS supports 47 such commands.

A FreeCS user is associated with a privilege level, *e.g.,* Super-User, Guest, Punished, Banned; the set of commands that a user can issue is based upon her privileges. Similarly, a group can also be Open or Locked: users can freely join Open groups, while special privileges are required to join Locked groups. FreeCS enforces a variety of policies on users and groups. However, these policies are hard-coded in FreeCS (using language constructs, such as `if`...`then`...`else`).

We ported FreeCS to use TMI-based authorization policy enforcement. We used the TMI reference monitor to both replace FreeCS's enforcement mechanisms for several commands, as well as augment FreeCS to enforce several policies that it currently does not. Our implementation uses the XACML framework to express FreeCS authorization policies. In each case, if a user is not authorized to perform an operation (*e.g.,* join a group), FreeCS rolls back the failed operation, and sends a failure message to the user; no additional failure-handling code was required.

Overall, our port of FreeCS to use TMI-based enforcement required about 860 changes in seven classes. In all, we introduced transactions for all 47 FreeCS client requests. Most of the changes to FreeCS involved replacing reads/writes of transactional objects with DSTM2 accessor functions. As described in Section 4, this is a limitation of any library-based STM, which can be overcome with compiler-based or language-based support for transactions.

Several security-relevant operations in FreeCS consisted of multiple low-level object accesses. We therefore supplied the TMI reference monitor with fingerprints to recognize these security-relevant operations. These fingerprints were also sensitive to the order of accesses; we therefore used introspection logs as the basis for TMI-based enforcement.

5.4 WeirdX: A window management server

WeirdX is a Java-based X window server that consists of about 27,000 lines of code [40]. WeirdX supports the X protocol; therefore X clients can connect to WeirdX, and communicate with each other and with WeirdX in much the same way that they do on the X11 server [58].

Much like the X11 server, WeirdX does not enforce any policies on X clients that connect to it. Therefore, a malicious X client can access/modify resources that belong to other clients of WeirdX. This has serious consequences; an X client can register to receive events (*e.g.,* keystrokes) sent to other clients, or even shut them down. Prior work has motivated the need for window management servers to enforce authorization policies on clients to prevent such attacks [41].

We ported WeirdX to enforce authorization policies using TMI. We used TMI in conjunction with the XACML framework to formulate and enforce several policies that have been discussed in prior work [41], including preventing a rogue X client from killing arbitrary X clients, and mediating copy/paste operations. For example, we enforced the Bell-LaPadula policy on how data copied from an X client can be pasted to other X clients.[1] Overall, our port of WeirdX required about 4,800 changes in 25 classes, and introduced 108 transactions to the code that dispatches X protocol requests to handlers; as with FreeCS, most of these changes were related to replacing reads/writes of transactional objects with DSTM2 accessor functions. We also had to make a few changes to WeirdX code that handled output to the screen. In particular, we modified WeirdX to buffer writes that happen within transactions, and flush the buffers only upon a transaction commit.

[1] We used IP addresses of X clients as their security labels, though finer-grained security labels are possible with OS support [54].

	No-STM	STM-only	STM-TMI/Eager	STM-TMI/Lazy	STM-TMI/Overlapped
GradeSheet	398 μs	451 μs	**452 μs (0.3%)**	458 μs (1.4%)	694 μs (54%)
Tar	4.96 s	15.40 s	15.24 s (−1.0%)	16.87 s (9.5%)	**12.96 s (−15.8%)**
FreeCS	321 μs	3907 μs	5471 μs (40%)	**4075 μs (4.3%)**	4244 μs (8.6%)
WeirdX	0.23 ms	6.40 ms	69.12 ms (10.8×)	7.74 ms (21%)	**7.15 ms (11%)**

Figure 6. Performance measurements for servers. The overhead of using TMI (shown in parentheses) is calculated by comparing the STM-TMI variant against the STM-only variant. Numbers in bold show the most efficient TMI variant for each server.

5.5 Performance

We evaluated the performance of three variants of the GradeSheet, Tar, FreeCS and WeirdX servers. The first variant, No-STM, is an unmodified server. The second variant, STM-only, is a server that has been ported to use an unmodified DSTM2, for concurrency control only (STM-only). The third server variant, STM-TMI, is the server ported to use our modified DSTM for concurrency control and to use TMI-based authorization. In this variant, each client request to the server is handled as an STM transaction, the TMI reference monitor mediates on all access to security-relevant resources, and a server authorization manager performs security checks.

The same authorization policy is enforced in all three server variants. In the STM-TMI variant, enforcement uses our added TMI-based enforcement mechanisms, while the No-STM and STM-only variants use the original server authorization mechanisms. However, only the STM-TMI variant of WeirdX performs enforcement, since the unmodified server had no security policy.

We ran experiments and measured the performance of the four servers and their variants. Furthermore, for the STM-TMI variant, we ran three experiments, using implementations of eager, lazy and overlapped authorization managers. Figure 6 reports the arithmetic mean of the measured wall-clock execution time for the following processing: the handling of a client request in GradeSheet (avg. over 60,000 requests), the archiving of 10,000 empty files to/from a ramdisk using Tar (avg. over 10 runs), the addition of a user to a FreeCS forum (avg. over 750 runs), and creating and mapping subwindows (as performed by the x11perf/create benchmark [57]). These experiments included no contention and no authorization failures; they ran on a quiescent system with Intel Core 2 Duo processors. The measurements did not significantly vary from the reported averages.

As Figure 6 shows, lazy TMI enforcement (STM-TMI/Lazy) always incurs acceptable overheads—under 21% in all cases. Eager TMI-based enforcement (STM-TMI/Eager) has even lower overhead for GradeSheet and Tar (which shows a not-statistically-significant 1% speedup). However, eager enforcement is not a good strategy for FreeCS and WeirdX, where it results in a significant slowdown. This is because both FreeCS and WeirdX use a complex fingerprint to identify security-relevant operations, such as joining a forum and creating/mapping windows. While lazy enforcement can match fingerprints once and for all, before commit, eager enforcement checks for a match on every security-critical access. In particular, the x11perf/create benchmark creates and maps several hundred subwindows; with eager enforcement, each of the create and map operations entails fingerprint matching and policy lookup, which results in a very significant slowdown of WeirdX. Thus, the choice between lazy and eager TMI-based enforcement can depend on the authorization strategy, as well as the server software and its workload.

With overlapped TMI enforcement (STM-TMI/Overlapped), our implementation creates a thread for each transaction. Therefore, one may expect significant overhead on short transactions; indeed, as shown in Figure 6, we measured 54% overhead for GradeSheet. However, we observed a *speedup* of 15.8% for Tar; this is because expensive stack-inspection-based authorization of Tar can be usefully overlapped with transaction execution. FreeCS and WeirdX can similarly benefit from having fingerprint matching performed on a parallel thread. (However, for FreeCS, the best strategy is still lazy, one-shot matching upon commit.) Thus, whether to overlap TMI-based enforcement can depend on transaction length, the cost of thread creation and synchronization, and the cost of authorization checks.

As shown in Figure 6, TMI-based authorization has acceptable overhead when applied to servers written to use STM techniques; TMI-based enforcement can even improve the performance of such software. However, the No-STM and STM-only columns show that simply using DSTM2 for concurrency control results in a very substantial performance overhead. It must be emphasized that this high overhead (11× for FreeCS and 28× for WeirdX) does not apply in general to other STM systems (see [43]). Rather, it is an artifact of the library-based DSTM2 implementation, which is a research prototype without compiler or language-runtime support. STM systems that are compiler-based [3], language-based [33], or hardware-accelerated [19, 23, 49] incur much lower overheads than DSTM2. The overhead of future, language-integrated STM systems is likely to be competitive with other means of concurrency—especially as STM techniques are widely adopted, and optimized, and accelerated with hardware support.

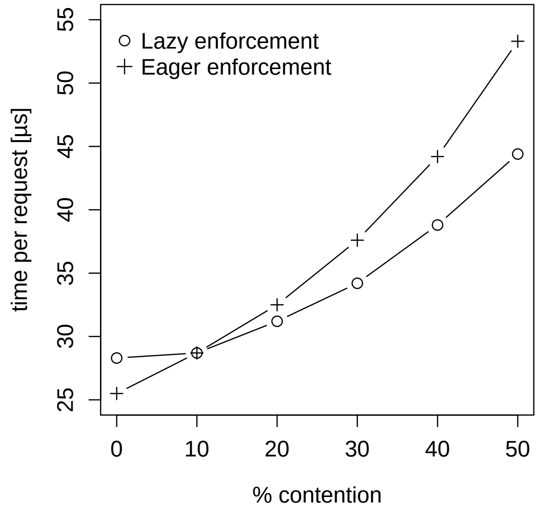

Figure 7. Comparing lazy versus eager enforcement in the presence of contention.

We also evaluated the overhead of lazy and eager authorization managers at different contention levels; Figure 7 presents the results of this experiment. At *x*% contention, a transaction execution has *x*% chance of being retried because it conflicts with another, concurrent transaction; thus, 100−*x*% of execution attempts will return a response (by commit or authorization failure). We measured the time for GradeSheet requests while varying the contention from 0–50%; of all requests, 5% failed an authorization check. At low contention, eager enforcement is more efficient, since it performs no work in copying metadata into an introspection log. However, as contention grows, lazy enforcement becomes more efficient. This is because eager enforcement performs authorization checks on all accesses, even for transactions that must be retried because they encounter a conflict due to contention. In contrast, with lazy enforcement, authorization checks happen only once, at commit, no matter how often the transaction was retried.

Determining the most efficient enforcement strategy requires measuring factors such as contention, the cost of authorization checks, the time to execute a transaction body, and the rate of authorization failures. We have implemented simple measure-and-adapt authorization managers that switch between eager and lazy enforcement. Our experiments with such adaptive enforcement confirms that it can follow the more efficient of the two curves shown in Figure 7.

6. Related work

We focus our discussion of related work to four areas: applications of transactions to security; comparing TMI and virtual machine introspection; work on exception handling and recovery; and aspect-oriented software development.

Applications of transactions to security. Clark and Wilson's influential model for commercial security policies is defined in terms of separation of duty and well-formed transactions that preserve data integrity [14]. Our TMI-based approach is well-suited to enforcing such security policies: TMI can ensure that transactions never commit unless all actions are authorized, *e.g.,* based upon separation of duty policies, and that application data is consistent, *i.e.,* the integrity of all data items has been verified.

The Vino operating system used transactions to isolate the effects of misbehaving and/or malicious kernel extensions, such as device drivers, by executing them in the context of a transaction. If an extension fails or violates system policy (*e.g.,* by hoarding resources), Vino simply terminates the corresponding transaction, thereby isolating the extension. In contrast to TMI, which uses existing STM mechanisms to enforce policies, Vino used a custom-built transaction manager in the kernel. Further, Vino only relied on transactions for remediation; though resources accessed during a transaction are logged, they are only used to restore system state. In contrast, TMI uses access logging in a key way to achieve complete mediation.

Chung *et al.* recently used transactional memory to implement thread-safe binary translation and applied it to implement information-flow tracking [13]. This work primarily uses transactional memory to ensure thread-safe access to security metadata (*i.e.,* taint bits). The TMI architecture, coupled with a dynamic binary translation system, can also be applied to track information flow; in this role, TMI will primarily help avoid TOCTTOU bugs when accessing security metadata. However, Chung *et al.* show that information-flow tracking can be implemented using hardware-only transactional memory techniques as well; in contrast, the TMI architecture requires software support.

Peyton-Jones and Harris proposed a framework to support programmer-supplied data invariants in the Haskell STM [47]. These invariants are Boolean functions that are evaluated just before a transaction commits. Much like TMI, any invariants that are violated result in a transaction abort. However, this framework does not allow security checks at each access to a security-relevant resource, or the maintenance of introspection logs. Even so, a TMI reference monitor could potentially benefit from its supporting mechanisms.

Virtual machines for security. Virtual machine monitors (VMM) have recently emerged as a popular location to implement security enforcement mechanisms [10]. Indeed, at least superficially, they offer many of the same benefits as a TMI-based security mechanism. They allow introspection of runtime state of the guest operating system, thereby easing the construction of intrusion detection systems that resist evasion and attack [30]. They also permit rollback and replay of system state, thereby allowing the construction of malware detection and forensic tools (*e.g.,* [16, 42]).

TMI offers several advantages over the VMM-based approach. Foremost, TMI extends declarative concurrency control and therefore applies at the instruction-level of granularity, in contrast to VMM-based techniques, which apply at a much coarser level of granularity. This difference is significant. It allows TMI-based techniques finer-grained control over program execution, thereby permitting instruction-level rollback. It also improves application development by easing the integration of security enforcement mechanism: TMI eliminates TOCTTOU bugs by construction, simplifies the handling of security exceptions, and ensures complete mediation of all resource accesses within transactions. Such fine grained control over program execution may possibly be implemented within a VMM as well, but the engineering overheads of doing so are much higher.

In contrast, the VMM-based approach provides better control over system level events, such as I/O. For example, file system changes can be undone by simply rolling back to an earlier state; TMI must be coupled with transactional I/O libraries to support rollback of system-level events. In addition, the VMM-based approach can be used to enforce security policies on legacy binaries; in contrast, TMI requires changes to server code. Combining TMI with VMM-based techniques to construct security mechanisms is an interesting area for future work.

Exception handling and recovery. As argued earlier, TMI leverages transaction rollback to simplify the handling of security exceptions. An IBM survey reports that a large fraction of server code relates to exception handling [12]. Weimer and Necula [55] found that up to 46% of code on several Java benchmarks was exception handling code (or reachable from it) and that `SecurityException` was one of the most common exception classes that these benchmarks handled erroneously. Exception handling code is often complex, especially when it must consider several corner cases [9, 25, 55]. Indeed, there is experimental evidence that exception handling code is more likely to contain bugs [15]. Because security exceptions account for a large fraction of exception handling code, the TMI-based approach can result in easier-to-maintain, and less cluttered code.

Another related approach to handling exceptions without impacting availability is Microreboot [8]. This technique offers fine-grained control over the server, *e.g.,* by allowing parts of it to be rebooted, without impacting server availability. TMI is similar to Microreboot in that it also offers fine-grained, instruction-level control over exception handling in server software.

Aspect-oriented software development. Aspect-oriented programming languages, such as AspectJ [5] and AspectC++ [4], allow concerns that crosscut an application (*e.g.,* security and error-handling) to be developed separately and integrated with the application. An *aspect weaver* matches the application against a set of patterns (called pointcuts) and integrates appropriate *advice* (akin to actions) at each program point that matches a pattern.

Because TMI enforces authorization policies by introspecting on the STM's read/write sets, it is a dynamic aspect weaver. However, the key advantage that TMI provides over traditional aspect weavers is that it does not require advice to deal with authorizations exceptions, which automatically trigger transaction rollback. In contrast, traditional aspect weavers must be supplied with advice to restore application state on an exception. In addition, TMI also provides thread-safe aspect weaving, and does not introduce TOCTTOU bugs or deadlocks.

7. Summary

The correct implementation of security mechanisms is a difficult task, due to the challenges of providing complete mediation, preventing TOCTTOU bugs, and ensuring correct handling of policy violations. The TMI architecture can significantly reduce the difficulties of correctly implementing security enforcement. A TMI-based authorization mechanism can get precise information about all security-relevant runtime accesses, without having to worry about race conditions, and can handle security violations by rolling back to a consistent software state. TMI-based enforcement is flexible, and can integrate with other, existing security mechanisms. For some policies and workloads, TMI-based enforcement can lower the overhead and latency of enforcement; in particular, TMI allows authorization checks to be overlapped with execution. We believe that the combination of TMI and declarative concurrency control is a highly attractive architecture for the creation of future, secure software.

There are several avenues for future work on the TMI architecture. For instance, while this paper has focused on enforcing authorization policies, TMI can also implement many other security services, ranging from runtime information-flow tracking to intrusion forensics. Similarly, TMI may be combined with static analysis, in particular, to automatically identify authorization points (*e.g.,* as in [29]), to determine transaction boundaries, or to eliminate unnecessary reference monitor invocations. Finally, TMI enforcement might integrate some of the developments in the rapidly-progressing field of STM systems, such as the recently-proposed techniques for handling I/O within transactions [22, 48].

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *ACM SIGPLAN Conference on Principles of Programming Languages*, Jan 2008.

[2] M. Abadi and C. Fournet. Access control based on execution history. In *Networked and Distributed System Security Symposium*, 2003.

[3] A. Adi-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM Conference on Programming Language Design and Implementation*, June 2006.

[4] The home of AspectC++. http://www.aspectc.org.

[5] AspectJ project. http://www.eclipse.org/aspectj.

[6] F. Besson, T. Blanc, C. Fournet, and A.D. Gordon. From stack inspection to access control: a security analysis for libraries. In *Computer Security Foundations Workshop*, June 2004.

[7] M. Bishop and M. Digler. Checking for race conditions in file accesses. *Computer Systems*, 9(2):131–152, Spring 1996.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. In *Symp. on Operating System Design and Implementation*, Dec 2004.

[9] T. Cargill. Exception handling: A false sense of security. *C++ Report*, 6(9), November 1994.

[10] P. M. Chen and B. Noble. When virtual is better than real. In *Workshop on Hot Topics in Operating Systems*, May 2001.

[11] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *ACM Conference on Programming Language Design and Implementation*, June 2008.

[12] F. Christian. Exception handling. Technical Report RJ5724, IBM Research, 1987.

[13] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translaction using transactional memory. In *IEEE Symp. on High-Performance Computer Architecture*, Feb 2008.

[14] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *1999 IEEE Symposium on Security and Privacy*, May 1987.

[15] F. Cristian. Exception handling and tolerance of software faults. In *Software Fault Tolerance*. Wiley, 1995.

[16] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symposium Operating System Design and Implementation*, December 2002.

[17] Michael Emmi, Jeff Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *ACM Conference on Principles of Programming Languages*, January 2007.

[18] Ú. Erlingsson and F.B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, 1999.

[19] C. Cao Minh *et al.*. An effective hybrid transactional memory system with strong isolation guarantees. In *Intl. Symp. on Computer Architecture*, June 2007.

[20] K. E. Moore *et al.*. LogTM: Log-based transactional memory. In *Intl. Conf. on High Performance Computer Architecture*, Feb 2006.

[21] L. Hammond *et al.*. Transactional memory coherence and consistency. In *Intl. Symp. on Computer Architecture*, June 2004.

[22] M. J. Moravan *et al.*. Supporting nested transactional memory in LogTM. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[23] P. Damron *et al.*. Hybrid transactional memory. In *Architectural Support for Programming Languages and Operating Systems*, Oct 2006.

[24] Extensible access control markup language. http://xml.coverpages.org/xacml.html.

[25] C. Fetzer, P. Felber, and K. Hogstedt. Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering*, 30(8):547–560, 2004.

[26] B. Fletcher. Case study: Open source and commercial applications in a Java-based SELinux cross-domain solution. In *Annual Security-enhanced Linux Symposium*, March 2006.

[27] FreeCS—the free chatserver. http://freecs.sourceforge.net.

[28] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *IEEE Symposium on Security and Privacy*, May 2006.

[29] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *International Conference on Software Engineering*, May 2007.

[30] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Networked and Distributed System Security Symposium*, February 2003.

[31] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security*. Addison-Wesley, second edition, September 2003.

[32] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[33] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

[34] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *ACM Conference on Principles and Practice of Parallel Programming*, 2005.

[35] M. Herlihy, V. Luchango, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2006.

[36] M. Herlihy, V. Luchango, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *ACM Symposium on Principles of Distributed Computing*, July 2003.

[37] M. Hocking, K. Macmillan, and D. Shankar. Case study: Enhancing IBM Websphere with SELinux. In *Annual Security-enhanced Linux Symposium*, March 2006.

[38] T. Jaeger, A. Edwards, and X. Zhang. Consistency analysis of authorization hook placement in the Linux security modules

framework. *ACM TISSEC*, 7(2):175–205, May 2004.

[39] Jakarta apache commons. `http://commons.apache.org/transaction`.

[40] JCraft. WeirdX—pure Java window system server under GPL. `http://www.jcraft.com/weirdx`.

[41] D. Kilpatrick, W. Salamon, and C. Vance. Securing the X Window system with SELinux. Technical Report 03-006, NAI Labs, March 2003.

[42] S. T. King and P. M. Chen. Backtracking intrusions. In *ACM Symposium on Operating System Principles*, October 2003.

[43] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan Claypool, 2006.

[44] Mark D. Matthews. Distributed transactions with MYSQL XA, 2005.

[45] Microsoft. Transactional NTFS in Windows Vista. `http://msdn2.microsoft.com/en-us/library/aa363764.aspx`.

[46] E. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[47] S. Peyton-Jones and T. Harris. Transactional memory with data invariants. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[48] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing transactional memory in an operating system. In *21st ACM Symposium on Operating System Principles*, October 2007.

[49] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Intl. Symp. on Microarchitecture*, Dec 2006.

[50] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[51] N. Shavit and D. Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, August 1995.

[52] T. Shpeisman, V. Menon, A. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[53] Tar for Java: The com.ice.tar package. `http://trustice.com/java/tar/`.

[54] E. Walsh. Integrating X.Org with security-enhanced Linux. In *Annual Security-Enhanced Linux Symposium*, March 2007.

[55] W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2), Mar 2008.

[56] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symp.*, August 2002.

[57] x11perf: The X11 server performance test program suite.

[58] The X11 Server, version X11R6.8 (X.Org Foundation).

[59] A. Yumerefendi, B. Mickle, and L. Cox. TightLip: Keeping applications from spilling the beans. In *USENIX Symposium on Networked System Design and Implementation*, Apr 2007.

[60] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security Symposium*, August 2002.