Enforcing Authorization Policies using Transactional Memory Introspection

Arnar Birgisson
Úlfar Erlingsson
Reykjavik University

Mohan Dhawan
Vinod Ganapathy
Liviu Iftode
Rutgers University
Overview

• Three main difficulties in policy enforcement
• Transactional Memory Introspection as a solution
• Variants of TMI
• Implementation and evaluation
• Future work
Difficulty 1

Time of check vs. time of use

if (allowed(principal, resource, operation)) {
   perform operation on resource
}

Difficulty 1
Time of check vs. time of use

```java
if (allowed(principal, resource, operation)) {
    perform operation on resource
}
```
Difficulty 1
Time of check vs. time of use

if (allowed(principal, resource, operation)) {
    Other thread may run here!
    perform operation on resource
}

Difficulty 1
Time of check vs. time of use

```java
if (allowed(principal, resource, operation)) {
    Other thread may run here!
    perform operation on resource
}
```

Interleaving code may invalidate the check
if (allowed(principal, resource, operation)) {
    perform operation on resource
}
Solution 1

Use locks

```c
lock(resource);
if (allowed(principal, resource, operation)) {
    perform operation on resource
}
release(resource);

... but locks are difficult to manage and prone to errors.
```
Solution 1
Software Transactional Memory

atomically {
    if (allowed(principal, resource, operation)) {
        perform operation on resource
    }
}

}
Solution 1
Software Transactional Memory

```java
atomically {
    if (allowed(principal, resource, operation)) {
        perform operation on resource
    }
}
```

Uses parallel, speculative execution
Solution 1
Software Transactional Memory

atomically {
  if (allowed(principal, resource, operation)) {
    perform operation on resource
  }
}

Uses parallel, speculative execution
Monitors all access to memory
Solution 1
Software Transactional Memory

\[
\text{atomically \{ }
\text{  if (allowed(principal, resource, operation)) \{ }
\text{    perform operation on resource }
\text{  \}}}
\]

Uses parallel, speculative execution
Monitors all access to memory
Can roll back and retry on conflict
Solution 1
Software Transactional Memory

atomically {
  if (allowed(principal, resource, operation)) {
    perform operation on resource
  }
}

STM guarantees atomicity, consistency and isolation of atomic blocks.
Nothing new here...
Nothing new here...

... *but we can do more*
TMI

Transactional Memory Introspection
Where TMI fits in with STM

- TX body
- Contention Mgr.
- Log
- Commit

Flow:
- TX body to Contention Mgr.
- Contention Mgr. to Log
- Log to Commit
- OK from Commit

Branches:
- Rollback & Retry
- Conflict
Where TMI fits in with STM

TX body → Contention Mgr. → Log

Log → OK

OK → Commit

TX body → Conflict

Conflict → Rollback & Retry
Where TMI fits in with STM

1. TX body → Contention Mgr.
2. Contention Mgr. → Log
3. Log → Policy
4. Policy → Authorization
5. Authorization → Commit
   - Allow
6. Authorization → Abort & Stop
   - Denied
7. TX body → Rollback & Retry
   - Conflict
Where TMI fits in with STM

TX body → Contention Mgr. → Log

- Conflict → Rollback & Retry
- OK → Authorization

Authorization:
- Allow → Commit
- Denied → Abort & Stop

TMI
Difficulty 2 : Error handling

if (allowed(principal, resource1, op1)) {
    perform op1 on resource1
} else {
    clean up and report error
}
if (allowed(principal, resource1, op1)) {
    perform op1 on resource1
} else {
    clean up and report error
}

if (allowed(principal, resource2, op2)) {
    perform op2 on resource2
} else {
    clean up after op1;
    clean up after op2 and report error
}
Difficulty 2: Error handling

```java
if (allowed(principal, resource1, op1)) {
    perform op1 on resource1
} else {
    clean up and report error
}

if (allowed(principal, resource2, op2)) {
    perform op2 on resource2
} else {
    clean up after op1;
    clean up after op2 and report error
}
```

This quickly becomes hard to manage
Difficulty 2 : Error handling

- Error handling accounts for a large fraction of server software, over two-thirds [IBM’87]
- Exception handling code itself is prone to errors [Fetzer and Felber ’04]
- SecurityException is the one most often handled incorrectly [Weimer & Necula OOPSLA’04]
Difficulty 3 : Complete mediation

```java
if (allowed(principal, resource1, op1)) {
    perform op1 on resource1
} else {
    clean up and report error
}

if (allowed(principal, resource2, op2)) {
    perform op2 on resource2
} else {
    clean up after op1;
    clean up after op2 and report error
}
```
Difficulty 3 : Complete mediation

if (allowed(principal, resource1, op1)) {
    perform op1 on resource1
} else {
    clean up and report error
}

if (allowed(principal, resource2, op2)) {
    perform op2 on resource2
} else {
    clean up after op1;
    clean up after op2 and report error
}

Easy to forget or miss checks in complex code
Difficulty 3 : Complete mediation

- A real problem in current practice
- Bugs of this kind found in the Linux kernel, page_cache_read did not check for file permissions [Zhang et al. USENIX Security ’02]
- Decentralized, ad-hoc hard-coded access checks, leads to errors when code changes.
- Also a problem in Linux [Jaeger et al. ’04]
TMI takes care of

“security boilerplate”
Step 1: Implicit abort

atomically {
    if (allowed(principal, resource1, op1)) {
        perform op1 on resource1;
    } else {
        clean up and report error
    }

    if (allowed(principal, resource2, op2)) {
        perform op2 on resource2;
    } else {
        clean up after op1;
        clean up after op2 and report error
    }
}
Step 1 : Implicit abort

atomically {
    if (allowed(principal, resource1, op1)) {
        perform op1 on resource1;
    }
    if (allowed(principal, resource2, op2)) {
        perform op2 on resource2;
    }
}
Step 1 : Implicit abort

atomically {
  if (allowed(principal, resource1, op1)) {
    perform op1 on resource1;
  }
  if (allowed(principal, resource2, op2)) {
    perform op2 on resource2;
  }
  on abort {
    report error;  // no cleanup necessary
  }
}
Step 2 : Implicit access checks

atomically {
    if (allowed(principal, resource1, op1)) {
        perform op1 on resource1;
    }
    if (allowed(principal, resource2, op2)) {
        perform op2 on resource2;
    }
}

} on abort {
    report error;  // no cleanup necessary
}
Step 2: Implicit access checks

```java
atomically [principal] {
    if (allowed(principal, resource1, op1)) {
        perform op1 on resource1;
    }
    if (allowed(principal, resource2, op2)) {
        perform op2 on resource2;
    }
}
```

on abort {
    report error;  // no cleanup necessary
}
Step 2 : Implicit access checks

atomically [principal] {
    perform op1 on resource1;
    perform op2 on resource2;
} on abort {
    report error; // no cleanup necessary
}
Step 2 : Implicit access checks

atomically [principal] {  
    perform op1 on resource1;  
    perform op2 on resource2;  
} on abort {  
    report error;    // no cleanup necessary  
}  

TMI invokes reference monitor  
- on every security-relevant memory access  
- before every transaction commit
Step 2: Implicit access checks

```plaintext
atomically [principal] {
    perform op1 on resource1; ★
    perform op2 on resource2; ★
} on abort {
    report error; // no cleanup necessary
}
```

TMI invokes reference monitor
- on every security-relevant memory access ★
- before every transaction commit
Step 2: Implicit access checks

```plaintext
atomically [principal] {
    perform op1 on resource1;
    perform op2 on resource2;
}
```

TMI invokes reference monitor
- on every security-relevant memory access
- before every transaction commit
Step 2 : Implicit access checks

atomically [principal] {
    perform op1 on resource1;
    perform op2 on resource2;
} on abort {
    report error; // no cleanup necessary
}

Reference monitor can delay policy evaluation
- logs a copy of relevant metadata
- security policy evaluation based on this log
- evaluation can be delayed until commit
Pseudo-code for policy evaluation

before commit of each transaction \( T \) {
    for (resource, op) in \( T.\text{log} \) {
        if (not allowed(\( T.\text{principal}, \text{resource, op} \))
            abort \( T \);
    }
}

- \textit{TMI security manager} evaluates the policy
- \textbf{Supplied by the programmer,} decoupled from application logic
Pseudo-code for policy evaluation

before commit of each transaction \( T \) { 
for (resource, op) in \( T.log \) {
    if (not allowed(T.principal, resource, op)
        abort \( T \);
    }
}

- *TMI security manager* evaluates the policy
- *Supplied by the programmer*, decoupled from application logic
- *Reference monitor invoked on* all accesses. *Complete mediation for free.*
Variants of TMI reference monitors

- TX body
- Contention Mgr.
- Commit
- Log
- Rollback & Retry
- Conflict
- OK
Variants of TMI reference monitors

Lazy

Log metadata

Log

Validate log

TX body

Contention Mgr.

OK

Commit

Conflict

Rollback & Retry

Denied

Abort & Stop
Variants of TMI reference monitors

**Eager**

1. **TX body**
2. **Contention Mgr.**
3. **Commit**

- **Abort & Stop**
- **Denied**
- **Log**
- **OK**
- **Conflict**
- **Rollback & Retry**
- **Validate access**
Variants of TMI reference monitors

Overlapped

Send metadata

TX body

Contention Mgr.

Log

OK

Commit

Denied

Abort & Stop

Rollback & Retry

Conflict

Send decision

Authorization Thread
Implementation

• Builds on the DSTM2 library for Java
• Programmer specifies security metadata
• Reference monitor invoked with metadata
• Lazy, eager, overlapped or custom
• Adds less than 500 LOC to DSTM2
Evaluation

Ported four servers to use STM and TMI

STM only  Eager TMI  Lazy TMI  Overlapped TMI

GradeSheet  Tar  FreeCS  WeirdX

10.8x
Ported four servers to use STM and TMI

Evaluation

STM only  Eager TMI  Lazy TMI  Overlapped TMI

GradeSheet: 0.3%  -15.8%  4.3%  11%
Tar: 10.8x
FreeCS: 11%
WeirdX: 11%
Transactional Memory
Introspection in summary

- A new reference monitor architecture
- Decouples application logic from policy enforcement
- Freedom from TOCTTOU bugs
- Easier handling of authorization failures
- Easier to ensure complete mediation
Bedankt voor jullie aandacht!