Secure Program Execution via Dynamic Information Flow Tracking

G. Edward Suh, Jae W. Lee, David Zhang, Srinivas Devadas

Massachusetts Institute of Technology
Program Vulnerabilities

- Program bugs cause serious security risks
  - Attackers can gain total control of victim processes
  - Very difficult, if not impossible, to eliminate the bugs

- Existing solutions have limitations
  - Safe languages → re-programming, performance hit
  - Fix programs: new libraries, compilers → partial protection, re-compilation
  - Run-time monitoring: program shepherding → overheads
  - Other hardware solutions → partial protection
Our Goal

- Architectural support to defeat a broad range of security exploits (possibly all)
  - Focus on attacks to gain total control (shell)
  - Should work for legacy code and shared libraries
    - transparent to applications, run-time checks
  - Should have low overhead (performance and memory space)

- Need to find common requirements for successful security exploits
Attack Model: Example - Stack Smashing

- **Step 1.** Inject *malicious data* through legitimate channels
  - *Long input* for buffer overflows

- **Step 2.** Bugs modify unintended memory locations
  - The data flows into `buf[]`, overwrites a return address

- **Step 3.** Take control over
  - Jump to *injected target address* (return address in the example)
  - Execute *injected code*

```c
int func(void)
{
    char buf[256];
    while (gets(buf)) {...}
}
```

```
Stack

Return Address

buf (256 Bytes)

Other variables

Malicious Input data from gets()

Attack

Used for return

Stack

Other variables
```
Observation: Common Requirements for Successful Attacks

- All attacks come from identifiable I/O channels
  - Both OS and applications explicitly manage I/O

- Malicious inputs should be used for a few security sensitive operations to take control of a process
  - Instructions: executes malicious code from I/O
  - Code pointers: arbitrarily redirect the control flow
  - Data pointers for stores: overwrite a critical program variable (valid_passwd = 1)

- In most applications, instructions and pointers usually do not come directly from I/O
Our Protection Scheme

1. **Step 1. OS tags**
   - Potentially malicious inputs as *spurious*
   - Security Tags:
     - 0 – *authentic*
     - 1 – *spurious*

2. **Step 2. Processors track the flow**
   - of the spurious values
   - Dynamic Information Flow Tracking

3. **Step 3. Detect attacks**
   - Check and restrict the use of spurious values
   - Processors checks + trap handler
Implementation Overview

1. Which I/O to tag spurious
2. Which flows to track
3. When to trap

Security Policy

I/O

Operating System

Execution Monitor

I/O interface

Trap handler

Tags

Information flow tracker

Tag checker

Processor
Architectural Support
Security Tags

- 1-bit information to indicate whether a piece of data can be trusted
  - 0 – *authentic*
  - 1 – *spurious*

- Granularity
  - One for each general purpose register (GPR)
  - One for each byte in memory – 12.5% overhead is a naïve management
  - Multi-granularity tags - Only 1.4% space overhead, 2.1% bandwidth overhead on average (based on experiments)

At the start-up, all instructions and initial data will be tagged “authentic”
During the execution, the execution monitor sets the tag for each I/O input according to the security policy
Dynamic Information Flow Tracking

- Compute a new security tag for each operation
  - If *spurious* data controls a result, the result is also *spurious*

- Various types of dependencies exist
  - Direct copy: load/store spurious data
  - Computation: compute from spurious data
    - Pointer additions
    - Other computations
  - Load address: load from spurious address
  - Store address: store into spurious address

- Propagation Control Register (PCR) determines which dependencies to track
  - Execution monitor sets the register based on the security policy
Security Tag Computation Examples

T[R3] = T[R1] OR T[R2]

T[R2] = T[MEM] OR T[R1]

T[MEM] = T[R3] OR T[R1]
Tag Checker

Processor traps when spurious values are used for sensitive operations

Sensitive values to be checked
- Instructions
- Load addresses
- Store addresses
- Jump target addresses

Trap Control Register (TCR) determines which uses of spurious values generate a trap
Hardware Support Summary

- 1-bit tag for each GPR
- Small modification to ALU
  - Tag computation (logical OR)
- TLB contains tag types and tag pointers
- Separate tag caches
  - Allow parallel accesses to data and tags
  - Exploit multi-granularity tags
    - Tags will be often less than 1/8 of data
Security Policy
Security Policy

- Defines “spurious” values
  - I/O channels to be tagged
  - Dependencies to be tracked

- Defines illegal uses of spurious values
  - Trap conditions
  - Software checks in the handler

- Can be general for many programs, or customized for each program
Take 1: Maximum Security

- Untrusted I/O
  - ALL
- Tracked Dependencies
  - ALL
- Trap Condition
  - Instruction
  - Jump target address
  - Store address
- Trap Handler
  - Terminate the process

False alarms from spurious pointers

Need to balance security and false positives
I/O inputs are often used as offsets for pointer tables after a bound check

```
<table>
<thead>
<tr>
<th></th>
<th>PTR_1</th>
<th>PTR_2</th>
<th>PTR_3</th>
<th>...</th>
<th>PTR_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

From I/O
Take 2: Allow Legitimate Uses

- Untrusted I/O
  - ALL
- Tracked Dependencies
  - ALL but pointer offsets
- Trap Condition
  - Instruction
  - Jump target address
  - Store address
- Trap Handler
  - Terminate the process

For pointer additions such as 
\([4*r1+r2]\) in x86, 
`s4addq r1, r2, r3` 
(r3 ← r2+4*r1) in Alpha

The new tag = T[r2] 
assuming the bound check is done.
Example – Stack Smashing

- Load a return address
- JR R1 - Return
- Trap – spurious jump target address

<table>
<thead>
<tr>
<th>Memory</th>
<th>Data</th>
<th>I/O</th>
<th>I/O</th>
<th>I/O</th>
<th>Inst</th>
<th>Inst</th>
<th>Inst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I/O</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I/O</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inst</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Registers

Processor

Tag Check
Evaluation
Simulation Frameworks

- **Bochs (Intel x86)**
  - Keyboard and network I/O are tagged spurious
  - Used to evaluate the effectiveness of our scheme
  - x86 applications on Debian Linux (3.0r0)

- **SimpleScalar (Alpha)**
  - All I/O are tagged spurious
  - sim-fast: functional evaluations (false alarms, space overheads for tags)
  - sim-outorder: performance evaluations
  - SPEC CPU2000 benchmarks
Detecting Security Attacks

- **Buffer overflow testbed (by Wilander, 2003)**
  - Covers all 20 combinations possible in practice
    - Overwrite technique: direct, pointer redirection
    - Buffer location: stack, heap/BSS/data
    - Attack targets: return address, base pointer, function pointer, and longjmp buffers
  - The best protection scheme in 2003 detected only 50%

- **Format string attacks (from TESO security group)**
  - Overflow a buffer or use %n conversion specification

- **Detects and stops ALL security attacks tested**
  - So far, all known attacks directly inject pointers or instructions → lenient tag propagation does not matter
No False Alarms

- **Common x86 applications**
  - Debian Linux 3.0 (keyboard, network marked spurious)
  - System commands: ls, cp, vi, ping, etc.
  - openSSH server/client

- **Dynamically generated code**
  - A simple http server (TinyHttpd2) – marked spurious
  - SUN’s JAVA SDK 1.3 HotSpot VM with JIT

- **SPEC2000 CPU benchmarks**
  - Input files are marked spurious
Performance Degradation

- Various L2 sizes with 1/8 tag caches – 1.1% degradation on average
  - Pessimistic overhead: baseline case gets 12.5% larger caches if it helps

With the same cache sizes, the performance degradation is less than 0.1% in the worst case.
Conclusion

- **Dynamic information flow tracking provides a powerful tool for system security**
  - Tells whether a value came from untrusted I/O or not
  - Can restrict the use of potentially malicious input values
- **Our protection scheme is effective against large class of attacks**
  - Stops both buffer overflow and format string attacks
  - No false alarms for real-world applications
- **The overhead of tagging can be small**
  - 1.4% space, 2.1% bandwidth, 1.1% performance overhead
- **Many extensions are possible**
  - Automatically identify bound checks and strictly follow dependencies
  - Combine with static analysis
  - Other applications such as protecting private information or debugging
Questions?

- Our website
  http://www.csg.csail.mit.edu

- Contact Info
  Edward Suh (suh@mit.edu)