Raksha: A Flexible Information Flow Architecture for Software Security

Michael Dalton, Hari Kannan, Christos Kozyrakis

Computer Systems Laboratory
Stanford University
Motivation

- Software security is in a crisis

- Ever increasing range of attacks
  - High-level, semantic attacks are now the main threat
    - SQL injection, cross-site scripting, directory traversal, …
  - Low-level, memory corruption attacks are still common
    - Buffer overflow, double free, format string, …

- Need an approach to software security that is
  - Robust & flexible
  - Practical & end-to-end
  - Fast
DIFT: Dynamic Information Flow Tracking

- **DIFT taints** data from untrusted sources
  - Extra tag bit per word marks if untrusted

- **Propagate** taint during program execution
  - Operations with tainted data produce tainted results

- **Check** for suspicious uses of tainted data
  - Tainted code execution
  - Tainted pointer dereference (code & data)
  - Tainted SQL command

- **Potential:** protection from low-level & high-level threats
DIFT Example: Memory Corruption

Vulnerable C Code

```c
int idx = tainted_input;
buffer[idx] = x; // buffer overflow
```

```
set  r1 ← &tainted_input
load r2 ← M[r1]
add  r4 ← r2 + r3
store M[r4] ← r5
```

Tainted pointer dereference ⇒ security trap
Software DIFT Systems

- DIFT through code instrumentation [Newsome’05, Quin’06]
  - Transparent through dynamic binary translation

- Advantages
  - Runs on existing hardware
  - Flexible security policies

- Disadvantages
  - High overhead (≥3x)
  - Does not work with threaded or self-modifying binaries
  - Cannot protect OS

- Coverage: control-based, low-level attacks
Hardware DIFT Systems

- DIFT through HW extensions [Suh’04, Crandall’04, Chen’05]
  - Extend HW state to include taint bits
  - Extend HW instructions to check & propagate taint bits

☑ Advantages
  - Negligible runtime overhead
  - Works with threaded and self-modifying binaries

✗ Disadvantages
  - Inflexible security policies
  - False positives & false negatives
  - Cannot protect OS

- Coverage: control-based & data-based, low-level attacks
Outline

- Motivation & DIFT overview
- The Raksha architecture
  - Technical approach
  - Architectural features
  - Full-system prototype
- Evaluation
  - Security experiments
  - Lessons learned
- Conclusions
Raksha Philosophy

- Combine best of HW & SW
  - HW: fast checks & propagation, works with any binary
  - SW: flexible policies, high-level analysis & decisions

Goals
- Protect against high-level & low-level attacks
- Protect against multiple concurrent attacks
- Protect OS code

Comprehensive evaluation
- Run unmodified binaries on full-system prototype
- What works on a simulator, may not work in real life
Raksha Architecture & Features

Unmodified binaries

User 1
App Binary

User 2
App Binary

SysAdmin
Security Manager

Set security policies
Control HW check/propagate
Further SW analysis

Cross-process info flow
Save/restore tags

Operating System
Tag Aware

Tag Aware

HW Architecture
Tags

4 tag bits per word
HW check/propagate
User-level security traps

Operating System

Security Manager

Unmodified binaries

User 1
App Binary

User 2
App Binary

SysAdmin
Security Manager

Set security policies
Control HW check/propagate
Further SW analysis

Cross-process info flow
Save/restore tags

Operating System
Tag Aware

Tag Aware

HW Architecture
Tags

4 tag bits per word
HW check/propagate
User-level security traps
Setting HW Check/Propagate Policies

- A pair of policy registers per tag bit
  - Set by security manager (SW) when and as needed

- Policy granularity: operation type
  - Select input operands to check if tainted
  - Select input operands that propagate taint to output
  - Select the propagation mode (and, or)

- ISA instructions decomposed to ≥1 operations
  - Types: ALU, logical, branch, load/store, compare, FP, …
  - Makes policies independent of ISA packaging (RISC/CISC)
Check Policy Example: load

load r2 ← M[r1+offset]

Check Enables

1. Check source register
   If Tag(r1)==1 then security_trap

2. Check source address
   If Tag(M[r1+offset])==1 then security_trap

Both enables may be set simultaneously
Propagate Policy Example: `load`

\[
\text{load } r2 \leftarrow M[r1+\text{offset}]
\]

**Propagate Enables**

1. Propagate only from source register
   \[\text{Tag}(r2) \leftarrow \text{Tag}(r1)\]

2. Propagate only from source address
   \[\text{Tag}(r2) \leftarrow \text{Tag}(M[r1+\text{offset}])\]

3. Propagate only from both sources
   - OR mode: \[\text{Tag}(r2) \leftarrow \text{Tag}(r1) \lor \text{Tag}(M[r1+\text{offset}])\]
   - AND mode: \[\text{Tag}(r2) \leftarrow \text{Tag}(r1) \land \text{Tag}(M[r1+\text{offset}])\]
User-level Security Traps

- Why user-level security traps?
  - Fast switch to SW ⇒ combine HW tainting with SW analysis
  - No switch to OS ⇒ DIFT applicable to most of OS code

- Requires new operating mode, orthogonal to user/kernel

<table>
<thead>
<tr>
<th>Untrusted</th>
<th>Trusted</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Kernel</td>
</tr>
<tr>
<td>Limited instructions; limited address ranges; VM transparent</td>
<td>Direct access to tag bits &amp; tag instructions; Access to all instructions &amp; address ranges; VM/PM</td>
</tr>
</tbody>
</table>

- On security trap
  - Switch to trusted mode & jump to predefined handler
  - Maintain user/kernel mode (no address space change)
Protecting the Trap Handler

- Can malicious user code overwrite handler?
  - Use one tag bit to support a sandboxing policy
  - Handler data & code accessible only in trusted mode
Protecting the Trap Handler

Can malicious user code overwrite handler?

- Use one tag bit to support a sandboxing policy
- Handler data & code accessible only in trusted mode
Raksha Prototype System

- **Full-featured Linux system**
  - On-line since October 2006…

- **HW: modified Leon-3 processor**
  - Open-source, Sparc V8 processor
  - Single-issue, in-order, 7-stage pipeline
  - Modified RTL for processor & system
  - First DIFT system on FPGA

- **SW: custom Linux distribution**
  - Based on 2.6 kernel (modified to be tag aware)
  - Set HW policies using preloaded shared libraries
  - ≥120 packages (GNU toolchain, apache, postgresql, …)
Processor Pipeline

- Registers & memory extended with tag bits
- Tags flow through pipeline along with corresponding data
  - No changes in forwarding logic
  - No significant sources of clock frequency slowdown
Tag Granularity & Storage

- **Tag granularity**
  - HW maintains per word tag bits
  - What if SW wants byte or bit granularity for some data?
  - Maintain in SW using sandboxing & fast user-level traps
    - Acceptable performance if not common case…

- **Tag storage**
  - Initial HW ⇒ +4 bits/word in registers, caches, memory
    - 12.5% storage overhead
  - Multi-granularity tag storage scheme [Suh’04]
    - Exploit tag similarity to reduce storage overhead
    - Page-level tags ⇒ cache line-level tags ⇒ word-level tags
Prototype Statistics

- **Overhead over original**
  - Logic: 7%
  - Storage: 12.5%
  - Clock frequency: none

- **Application performance**
  - Check/propagate tags \(\Rightarrow\) no slowdown
  - Overhead depends on SW analysis
    - Frequency of traps, SW complexity, …

- **Worst-case example from experiments**
  - Filtering low-level false positives/negatives
  - Bzip2: +33% with Raksha’s user-level traps
  - Bzip2: +280% with OS traps
## Security Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Lang.</th>
<th>Attack</th>
<th>Detected Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceroute</td>
<td>C</td>
<td>Double Free</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>polymorph</td>
<td>C</td>
<td>Buffer Overflow</td>
<td>Tainted code ptr</td>
</tr>
<tr>
<td>Wu-FTPD</td>
<td>C</td>
<td>Format String</td>
<td>Tainted ‘%n’ in vfprintf string</td>
</tr>
<tr>
<td>gzip</td>
<td>C</td>
<td>Directory Traversal</td>
<td>Open tainted dir</td>
</tr>
<tr>
<td>Wabbit</td>
<td>PHP</td>
<td>Directory Traversal</td>
<td>Escape Apache root w. tainted ‘..'</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>C</td>
<td>Command Injection</td>
<td>Execve tainted file</td>
</tr>
<tr>
<td>ProFTPD</td>
<td>C</td>
<td>SQL Injection</td>
<td>Tainted SQL command</td>
</tr>
<tr>
<td>htdig</td>
<td>C++</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>PhpSysInfo</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>Scry</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
</tbody>
</table>

- Unmodified Sparc binaries from real-world programs
  - Basic/net utilities, servers, web apps, search engine
## Security Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Lang.</th>
<th>Attack</th>
<th>Detected Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceroute</td>
<td>C</td>
<td>Double Free</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>polymorph</td>
<td>C</td>
<td>Buffer Overflow</td>
<td>Tainted code ptr</td>
</tr>
<tr>
<td>Wu-FTPD</td>
<td>C</td>
<td>Format String</td>
<td>Tainted ‘%n’ in vfprintf string</td>
</tr>
<tr>
<td>gzip</td>
<td>C</td>
<td>Directory Traversal</td>
<td>Open tainted dir</td>
</tr>
<tr>
<td>Wabbit</td>
<td>PHP</td>
<td>Directory Traversal</td>
<td>Escape Apache root w. tainted ‘..’</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>C</td>
<td>Command Injection</td>
<td>Execve tainted file</td>
</tr>
<tr>
<td>ProFTPD</td>
<td>C</td>
<td>SQL Injection</td>
<td>Tainted SQL command</td>
</tr>
<tr>
<td>htdig</td>
<td>C++</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>PhpSysInfo</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>Scry</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
</tbody>
</table>

- Protection against low-level memory corruptions
  - Both control & non-control data attacks
## Security Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Lang.</th>
<th>Attack</th>
<th>Detected Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceroute</td>
<td>C</td>
<td>Double Free</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>polymorph</td>
<td>C</td>
<td>Buffer Overflow</td>
<td>Tainted code ptr</td>
</tr>
<tr>
<td>Wu-FTPD</td>
<td>C</td>
<td>Format String</td>
<td>Tainted ‘%n’ in vfprintf string</td>
</tr>
<tr>
<td>gzip</td>
<td>C</td>
<td>Directory Traversal</td>
<td>Open tainted dir</td>
</tr>
<tr>
<td>Wabbit</td>
<td>PHP</td>
<td>Directory Traversal</td>
<td>Escape Apache root w. tainted ‘..’</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>C</td>
<td>Command Injection</td>
<td>Execve tainted file</td>
</tr>
<tr>
<td>ProFTPD</td>
<td>C</td>
<td>SQL Injection</td>
<td>Tainted SQL command</td>
</tr>
<tr>
<td>htdig</td>
<td>C++</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>PhpSysInfo</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>Scry</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
</tbody>
</table>

- 1st DIFT architecture to detect semantic attacks
  - Without the need to recompile applications
## Security Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Lang.</th>
<th>Attack</th>
<th>Detected Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceroute</td>
<td>C</td>
<td>Double Free</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>polymorph</td>
<td>C</td>
<td>Buffer Overflow</td>
<td>Tainted code ptr</td>
</tr>
<tr>
<td>Wu-FTPD</td>
<td>C</td>
<td>Format String</td>
<td>Tainted ‘%n’ in vfprintf string</td>
</tr>
<tr>
<td>gzip</td>
<td>C</td>
<td>Directory Traversal</td>
<td>Open tainted dir</td>
</tr>
<tr>
<td>Wabbit</td>
<td>PHP</td>
<td>Directory Traversal</td>
<td>Escape Apache root w. tainted ‘..’</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>C</td>
<td>Command Injection</td>
<td>Execve tainted file</td>
</tr>
<tr>
<td>ProFTPD</td>
<td>C</td>
<td>SQL Injection</td>
<td>Tainted SQL command</td>
</tr>
<tr>
<td>htdig</td>
<td>C++</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>PhpSysInfo</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>Scry</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
</tbody>
</table>

- Protection is independent of programming language
  - Catch suspicious behavior, regardless of language choice
HW Policies for Security Experiments

- Concurrent protection using 4 policies
  1. Memory corruption (LL attacks)
     - Propagate on arithmetic, load/store, logical
     - Check on tainted pointer/PC use
     - Trap handler untaints data validated by user code
  2. String tainting (LL & HL attacks)
     - Propagate on arithmetic, load/store, logical
     - No checks
  3. System call interposition (HL attacks)
     - No propagation
     - Check on system call in untrusted mode
     - Trap handler invokes proper SW analysis
  4. Sandboxing policy (for trap handler protection)
     - Handler taints its code & data
     - Check on fetch/loads/stores in untrusted mode
Lessons Learned

- **HW support for fine-grain tainting is crucial**
  - For both high-level and low-level attacks
  - Provides fine-grain info to separate legal uses from attacks

- **Lesson from high-level attacks**
  - Check for attacks at system calls
  - Provides complete mediation, independent language/library

- **Lessons from low-level attack**
  - Fixed policies from previous DIFT systems are broken
    - False positives & negatives even within glibc
  - Problem: what constitutes validation of tainted data?
  - Need new SW analysis to couple with HW tainting
    - Raksha’s flexibility and extensibility are crucial
Conclusions

Raksha: flexible DIFT architecture for SW security

- Protects against high-level & low-level attacks
- Protects against multiple concurrent attacks
- Protects OS code (future work)

Raksha characteristics

- Robust – applicable to high-level & low-level attacks
- Flexible – programmable HW; extensible through SW
- Practical – works with any binary
- End-to-end – applicable to OS
- Fast – HW tainting & fast security traps
Questions?

- Want to use Raksha?
  - Keep an eye on [http://raksha.stanford.edu](http://raksha.stanford.edu)
  - Raksha port to Xilinx XUP board ($300 for academics)
  - Full RTL + Linux distribution
  - Expected release date in early July