Raksha: A Flexible Architecture for Software Security

Hari Kannan, Michael Dalton, Christos Kozyrakis

Computer Systems Laboratory

Stanford University

http://raksha.stanford.edu

Motivation

- Software security is in a crisis
- Ever increasing range of attacks on vulnerable SW
  - Low-level, memory corruption attacks are still common
    - Buffer overflow, double free, format string, …
  - High-level, semantic attacks are now the main threat
    - SQL injection, cross-site scripting, directory traversal, …
- 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

```
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
```

TRAP

T	Data
	r1:&input
	r2:idx=input
	r3:&buffer
	r4:&buffer+idx
	r5:x

- Tainted pointer dereference ⇒ security trap

DIFT Example: SQL Injection

Username:
Password:

Vulnerable SQL Code

SELECT * FROM *table*

WHERE *name*= 'username';

T	Data
	WHERE name=
	username

DIFT Example: SQL Injection

Username: christos' OR '1'='1
Password:

Vulnerable SQL Code

SELECT * FROM *table*

WHERE *name*= 'christos' OR '1'='1' ;

T	Data
	WHERE name=
	christos
	OR
	1=1

TRAP

❑ Tainted SQL command ⇒ security trap

DIFT in Software

❑ DIFT through code instrumentation [Newsome'05, Quin'06]

- Transparent through dynamic binary translation

☑ Software Advantages

- Runs on existing hardware
- Flexible security policies

☒ Software Disadvantages

- High overhead (≥3x)
- Does not work with threaded or self-modifying binaries
- Cannot protect OS
- Poor coverage: control-based, low-level attacks

The Case for HW Support for DIFT

❑ The basic idea [Suh'04, Crandall'04, Chen'05]

- Extend HW state to include taint bits
- Extend HW instructions to check & propagate taint bits

☑ Hardware Advantages

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

☒ Pitfalls to avoid

- Protect only against low-level attacks
- Fix security policies in HW
  - False positives & false negatives in real-world software
  - Cannot adapt to protect against future attacks
- Rely on OS mechanisms to handle security issues

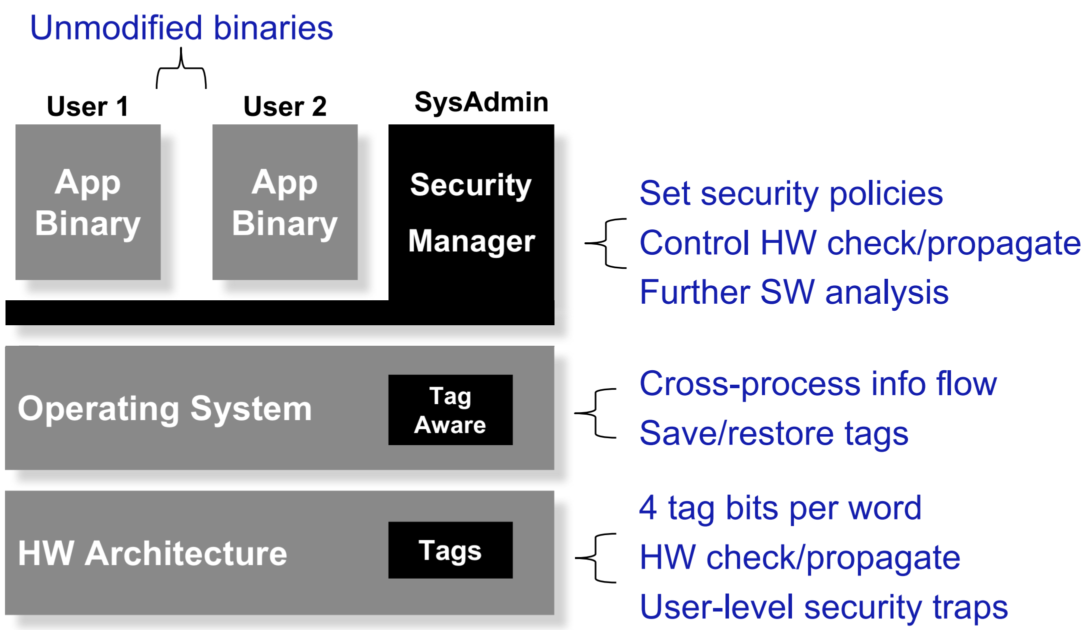
Outline

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

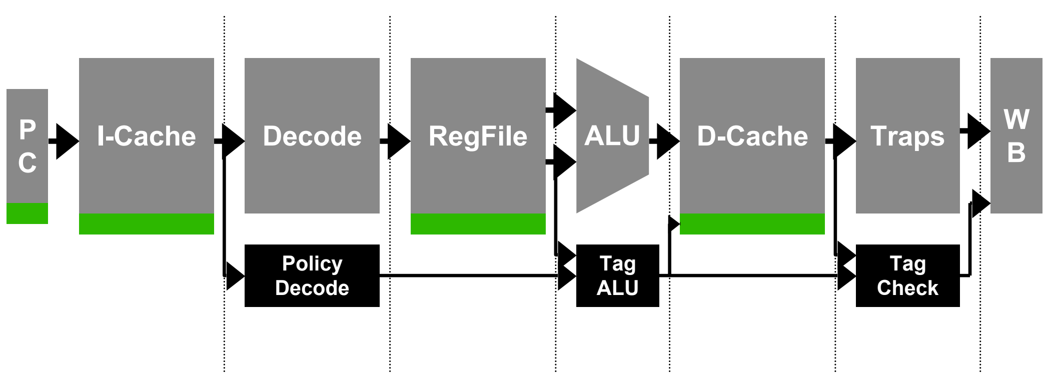
Raksha Philosophy

- ❑ 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 Overview & Features



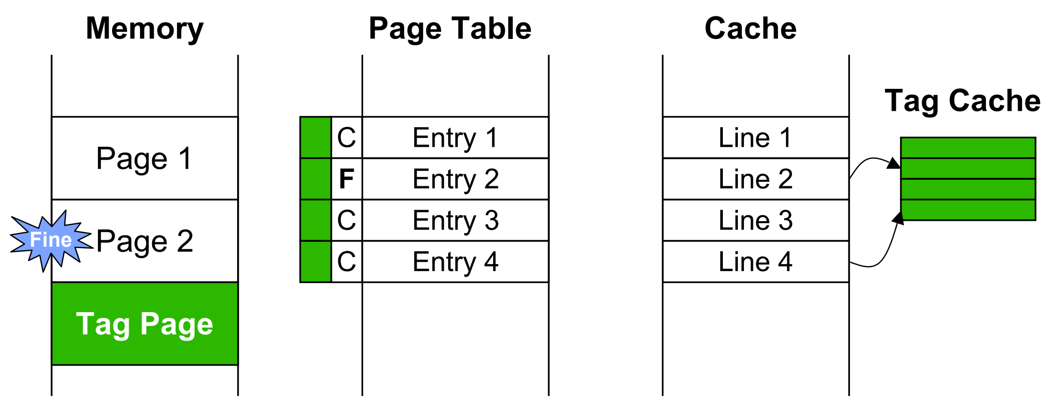
Raksha Architecture



- ❑ 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 Storage

- ❑ Simple approach: +4 bits/word in registers, caches, memory
  - 12.5% storage overhead
  - Used in our current prototype
- ❑ Multi-granularity tag storage scheme [Suh'04]
  - Exploit tag similarity to reduce storage overhead
  - Page-level tags ⇒ cache line-level tags ⇒ word-level tags



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
    - Same HW policies for both RISC & CISC ISAs

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`

`load  r2 ← M[r1+offset]`

Propagate Enables

1. Propagate only from source register
   Tag(**r2**) ←Tag(**r1**)

2. Propagate only from source address
   Tag(**r2**) ← Tag(**M[r1+offset]**)

3. Propagate only from both sources
   OR mode: Tag(**r2**) ←Tag(**r1**) **|** Tag(**M[r1+offset]**)
   AND mode: Tag(**r2**) ←Tag(**r1**) **&** Tag(**M[r1+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

User	Limited instructions; limited address ranges; VM
Kernel	Access to all instructions & address ranges; VM/PM

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

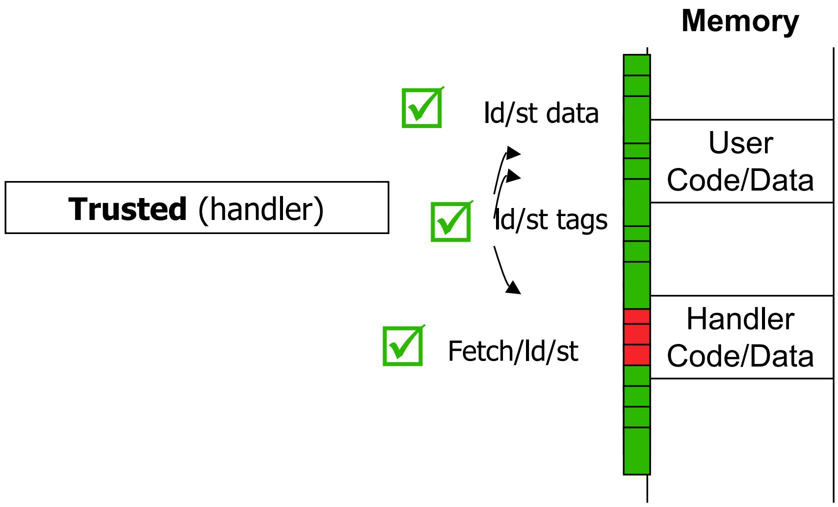
	Untrusted	**Trusted**
User **Kernel**	Tags are transparent	Direct access to tag bits & tag instructions

- 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?

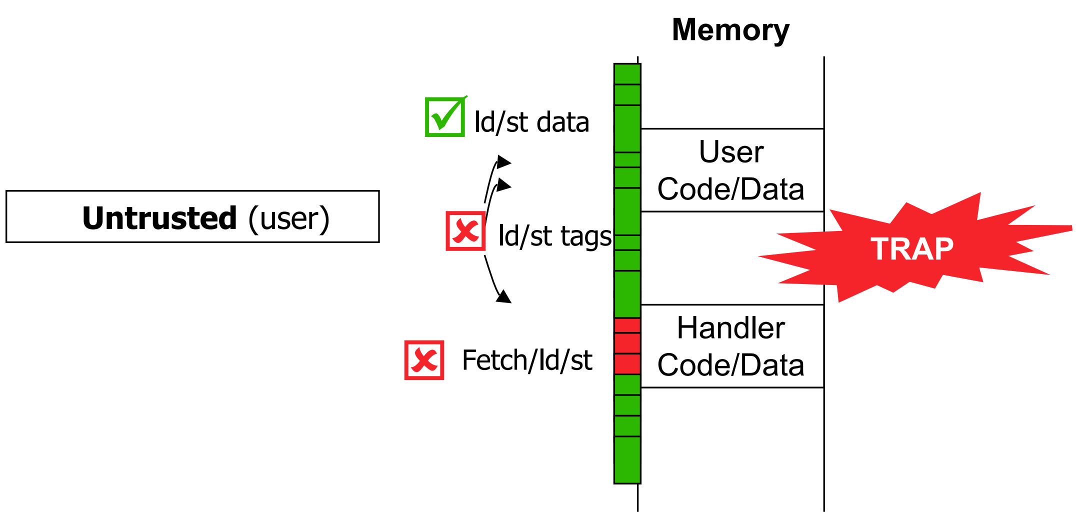
- Use one tag bit to support a sandboxing policy
- Handler data & code accessible <u>only</u> in trusted mode



Protecting the Trap Handler

❑ Can malicious user code overwrite handler?

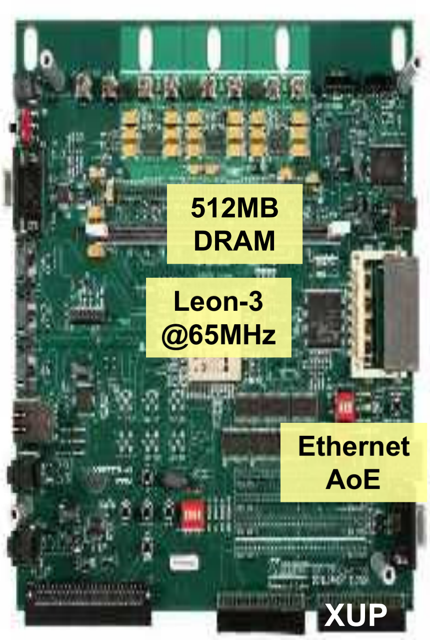
- Use one tag bit to support a sandboxing policy
- Handler data & code accessible <u>only</u> in trusted mode



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
  - Mapped to FPGA board
- ❑ SW: Gentoo-based Linux distribution
  - Based on 2.6 kernel (modified to be tag aware)
  - Set HW policies using preloaded shared libraries
  - ≥11,000 packages (GNU toolchain, apache …)

Prototype Statistics



- ❑ Overhead over original
  - Logic: 7%
  - Storage: 12.5%
  - Clock frequency: none
- ❑ Application performance
  - Check/propagate tags ⇒ 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 user-level traps
  - Bzip2: +280% with OS traps

Security Experiments

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

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

Security Experiments

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

- ❑ Protection against low-level memory corruptions
  - Both control & non-control data attacks

Security Experiments

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

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

Security Experiments

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

- Protection is independent of programming language
  - Catch suspicious behavior, regardless of language choice

HW Policies for Security Experiments

- Concurrent protection using 4 policies
- Memory corruption (LL attacks)
  - Propagate on arithmetic, load/store, logical
  - Check on tainted pointer/PC use
  - Trap handler untaints data validated by user code
- String tainting (LL & HL attacks)
  - Propagate on arithmetic, load/store, logical
  - No checks
- System call interposition (HL attacks)
  - No propagation
  - Check on system call in untrusted mode
  - Trap handler invokes proper SW analysis (e.g. SQL parsing)
- 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's 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?
  - Go to http://raksha.stanford.edu
  - Raksha port to Xilinx XUP board
    - $300 for academics
    - $1500 for industry
  - Full RTL + Linux distribution

Tag Granularity

- Raksha HW maintains per word tag bits
  - 1 tag bit per word per policy
  - Sufficient for most security analyses
- What if SW wants byte or bit granularity for some data?
  - Maintain finer-grain tags in SW
  - Implement sandboxing policy for corresponding data
    - Switch to SW handler when data accessed
  - Handlers provides storage and functionality for fine-grain tags
- Acceptable performance if not common case…