Trustworthy Systems through Information Flow Analysis

Fred Chong
Director, Computer Engineering
Director, Greenscale Center for Energy-Efficient Computing
Professor, Department of Computer Science
A Decade of Information Flow

- Information Flow Analysis can verify and enforce fundamental properties of security and reliability
2004: ISA-level information flow tracking
- Minos (Micro04), Aegis, Taintcheck, Rifle

2005-2006: Systems leveraging IFT
- DACODA (CCS05)
- Temporal Search (ASPLOS06)

2007-2008: IFT follow-on architectures
- SHIFT (ISCA08), Raksha, SIFT

2009-2013: Gate-level information flow
- GLIFT (ASPLOS09, TopPick10)
- Execution Leases (Micro09)

2010-2013: Hardware Description Languages
- Caisson (PLDI11)
- Sapper (ASPLOS14)

2011-2014: Trustworthy Hardware Designs
- *logic (ISCA11)
- SurfNOC (ISCA13, TopPick14)

Future: Statistical information flow analysis
PART 1: INSTRUCTION-LEVEL INFORMATION FLOW TRACKING
The Cost of Malware

- $315 Billion in 2014
- 33 terawatt-hours to view and filter SPAM in 2009
  - Equivalent to 2.4 Million homes
- Remote intrusions -> botnets
Three stages of a remote intrusion

1. Attack Trace
2. Exploit Vector
3. Bogus Control Flow Transfer
Minos [Crandall and Chong; MICRO 2004]

- Tagged architecture that tracks the integrity of every memory word
  - Network data is tainted
  - Control data (return pointers, function pointers, jump targets, etc.) should not be
- Taint tracking with every instruction
- Great for catching worms
  - Uses the $\gamma$ mapping
The Minos Architecture

- Integrity bit kept with every word of L1 cache
- Integrity bits grouped into words in L2 cache
- Integrity bits grouped into pages in VM
How to catch worms...
Only one false positive...
Minos: Lessons Learned

- Minos would be hard to implement
  - Can leverage existing hardware to emulate Minos (SHIFT)
- Control data integrity is powerful
  - Emulated Minos processor running a webserver deployed for 5 years.
    - Attacked 3 times a day
    - Never compromised
  - Minos used as the basis for DACODA and Temporal Search
Speculative Hardware Information Flow Tracking

Uses the Itanium NaT bit
- Fake a NaT error with an invalid address
- Instrument every load and store

Relative Perf. on SPEC (Control Speculation Off)

average 2.81X for byte-level and 2.27X for word-level
DACODA [Crandall et al.; CCS 2005]

• **DA**vis mal**C**OD**e** A**nalyzer

• Example of analysis in the exploit vector ($\varepsilon$)
  - Symbolic execution on the system trace during attacks that Minos catches
  - Discover invariants and attempts to construct signatures

• Used for an empirical analysis of polymorphism and metamorphism
  - Quantify and understand the limits
Worm Polymorphism and Metamorphism

- Viruses: Defender has time to pick apart the attacker’s techniques
  - e.g. Algorithmic scanners, emulation
- Worms: Attacker has time to pick apart the deployed network defense techniques
  - What can defenders do to evaluate the robustness of defenses against attacks that don’t exist yet?
The Epsilon-Gamma-Pi Model
How DACODA Works

• Gives each byte of network data a unique label
• Tracks these through the entire system
• Discovers predicates about how the host under attack interprets the network bytes
### Actual Worms/Attacks Caught by Minos and Analyzed by DACODA

<table>
<thead>
<tr>
<th>Name</th>
<th>OS</th>
<th>Port</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sasser</td>
<td>WinXP</td>
<td>445TCP</td>
<td>Buff.Over.</td>
</tr>
<tr>
<td>Blaster</td>
<td>WinXP</td>
<td>135TCP</td>
<td>Buff.Over.</td>
</tr>
<tr>
<td>RPCSS</td>
<td>WinXP</td>
<td>135TCP</td>
<td>Buff.Over.</td>
</tr>
<tr>
<td>Slammer</td>
<td>Whist.</td>
<td>1434UDP</td>
<td>Buff.Over.</td>
</tr>
<tr>
<td>Code Red II</td>
<td>Whist.</td>
<td>80TCP</td>
<td>Buff.Over.</td>
</tr>
<tr>
<td>Zotob</td>
<td>Win2K</td>
<td>445TCP</td>
<td>Buff.Over.</td>
</tr>
</tbody>
</table>
### Other Attacks Caught by Minos and Analyzed by DACODA

<table>
<thead>
<tr>
<th>Name</th>
<th>OS</th>
<th>Port</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc.statd</td>
<td>Linux</td>
<td>111 &amp; 918TCP</td>
<td>Form.Str.</td>
</tr>
<tr>
<td>innd</td>
<td>Linux</td>
<td>119TCP</td>
<td>Buff.Over.</td>
</tr>
<tr>
<td>Scalper</td>
<td>OBSD</td>
<td>80TCP</td>
<td>Int.Over.</td>
</tr>
<tr>
<td>ntpd</td>
<td>FBSD</td>
<td>123TCP</td>
<td>Buff.Over.</td>
</tr>
<tr>
<td>Turkey</td>
<td>FBSD</td>
<td>21TCP</td>
<td>OffByOne</td>
</tr>
</tbody>
</table>
### Single Contiguous Byte Strings

<table>
<thead>
<tr>
<th>Name</th>
<th>Longest String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sasser</td>
<td>36</td>
</tr>
<tr>
<td>Blaster</td>
<td>92</td>
</tr>
<tr>
<td>Work.</td>
<td>23</td>
</tr>
<tr>
<td>RPCSS</td>
<td>18</td>
</tr>
<tr>
<td>Slammer</td>
<td>1</td>
</tr>
<tr>
<td>CR11</td>
<td>17</td>
</tr>
<tr>
<td>Zotob</td>
<td>36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Longest String</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAuth</td>
<td>4</td>
</tr>
<tr>
<td>rpc.statd</td>
<td>16</td>
</tr>
<tr>
<td>innd</td>
<td>27</td>
</tr>
<tr>
<td>Scalper</td>
<td>32</td>
</tr>
<tr>
<td>ntpd</td>
<td>8</td>
</tr>
<tr>
<td>Turkey</td>
<td>21</td>
</tr>
</tbody>
</table>
Single Contiguous Signatures

- [Newsome et al.; IEEE S&P 2005] came to the same conclusion as we did and proposed sets of smaller byte strings called tokens.
DACODA: Lessons Learned

- Signatures are short and hard to make robust
- We can learn a lot about emerging malware threats by studying existing malware samples and their interactions with the systems they run on
- Need to develop techniques for behavioral analysis
Temporal Search

[Crandall et al.; ASPLOS 2006]

- Automated discovery of timebomb attacks
  - Example of analysis in the $\pi$ stage
- Prototype of behavior-based analysis
  - Proposed a framework for a problem space nobody has looked at before
  - Implemented parts of it
  - Identified the remaining challenges
    - By testing real worms with timebombs on our prototype
Basic Idea

• Find timers
  ▪ Run the PIT at different rates of *perceived time*
    • System performance stays the same
    • Correlate between PIT and memory writes

• Symbolic execution
  ▪ e.g. with DACODA

• Weakest precondition calculation

• Trick: *invert time predicates to make uncommon events common*
Temporal Search Results

- Discovered some predicates on day, hour, minute, etc. on a real time trace
- Unable to discover predicates
  - Sober.X, Kama Sutra
Temporal Search: Lessons Learned

• Manual analysis is tricky and time-consuming
  ▪ Temporal Search can dramatically improve response time

• A practical tool would need more comprehensive symbolic analysis, eg:
  ▪ Visual Basic strings
  ▪ While loops

• Behavioral analysis is arbitrarily difficult, but a few techniques for future use:
  ▪ Correlation to an important signal, such as the system timer
  ▪ Predicate inversion to convert uncommon to common case
PART 2: GATE-LEVEL INFORMATION FLOW TRACKING
• **Gate-Level Information Flow Tracking**

• Track every signal
  - All flows – implicit, explicit, and timing – must travel through signals on the chip (excluding physical attacks)

High-assurance Systems demand strong security guarantees
Hardware Vulnerabilities

- Covert channels
- Unspecified behavior
- Hardware bugs

Security needs to be enforced at hardware level
The Price of Assurance

- Evaluation Assurance Levels (EAL 1—7)
  - Evaluation of process, not end artifact

- RedHat Linux: EAL 4+
  - $30-$40 per LOC

- Integrity RTOS: EAL 6+
  - $10,000 per LOC

... and increasing. Many approaches.
Analysis Technique: GLIFT

AND

Shadow AND
Precise Information Flow: AND Gate

Use both inputs and input labels

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Required: Precise Information Flow

- Conventional OR-ing of labels monotonic
Analysis Technique: GLIFT
Avoiding Implicit Dataflow

**Traditional Conditional Branch Architecture**

- PC
- Instr Mem
- Reg File
- R1
- R2
- Jump target
- Is jump?

Through decode

**Predicates-Only Architecture**

- PC
- Instr Mem
- Reg File
- PR1
- R2
- Old value
- Jump target
- Is jump?

Through decode
Architectural Features

• No Implicit Flows: Conditionals are implemented through predicates (tags are set regardless of predicate values), Loops are all set to exact bounds (no variable looping), Memory operations are all fixed to loop iteration counts

• Execution Lease – a HW space-time sandbox [Micro09]
Execution Lease Architecture

- **Intr Mem**
  - Through decode

- **Predicates**
  - Old value

- **Reg File**
  - R1
  - R2

- **Data Memory**

- **Lease Unit**
  - Timer
  - PC
  - Memory

- **PC**
  - 0
  - 1
  - Jump target

- **Data Memory**
  - High
  - Low

- **Timer expired?**

- **Restore PC**
Execution Lease Architecture

- **Execution Lease Architecture**

  - **Instr Mem**
  - **Predicates**
  - **Reg File**
  - **Data Memory**
  - **Timer**
  - **PC**
  - **Memory**

  - **PC**
  - **Instr Mem**
  - **Predicates**
  - **Reg File**
  - **Data Memory**

  - **timer expired?**
  - **restore PC**

  - **through decode**
  - **jump target**
  - **old value**
  - **high low**

  - **Lease Unit**

The diagram illustrates the flow of execution in an architecture using a lease unit to manage execution time.
Execution Lease Architecture

Registers become untainted with trusted loads
Processor Design

- 5-stage pipeline
- Most of the common MIPS ISA
  - FPU
  - Multiplication / Division
- Cache and Memory
- Coded in Verilog (and our HDLs)
- Synthesized using Synopsis Design Compiler (90nm)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Instruction List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additive Arithmetic</td>
<td>add, addu, addiu, sub, subu</td>
</tr>
<tr>
<td>Binary Arithmetic</td>
<td>and, andi, or, ori, xor, xori, nor, slt, sltu, sra, srav, srl, srlv</td>
</tr>
<tr>
<td>Multiplicative Arithmetic</td>
<td>mult, mutlu, div</td>
</tr>
<tr>
<td>FPU instructions</td>
<td>add.s, sub.s, mul.s, div.s, neg.s, abs.s, mov.s cvt.s.w, cvt.w.s, le.s, lt.s, ge.s, gt.s</td>
</tr>
<tr>
<td>Branch</td>
<td>beq, bgt, ble, bne, bltz, bgez, beql, bnel, blel, bltzl, bc1t</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal, jalr</td>
</tr>
<tr>
<td>Memory Operation</td>
<td>lb, lbu, lhu, lw, sb, sh, sw; lw1, lw1r, swl, swr; swc1, lwc1</td>
</tr>
<tr>
<td>Others</td>
<td>slti, sltiu, lui, mflo, mfhi, mtc1, mfc1</td>
</tr>
<tr>
<td>Security Related</td>
<td>set-tag*, set-timer</td>
</tr>
</tbody>
</table>
Evaluation

- Functional Evaluation
  - Running benchmarks programs
    - mcf, specrand, bzip2 from SPEC CPU 2006
    - sha, rijndael, FFT from MiBench
  - Security Evaluation
    - Proof of non-interference
    - Micro-kernel + different security scenarios
## Overheads

<table>
<thead>
<tr>
<th>Processor</th>
<th>Area (µm²)</th>
<th>Delay (ns)</th>
<th>Power (mW)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Processor</td>
<td>560,687</td>
<td>5.57</td>
<td>2.79</td>
<td></td>
</tr>
<tr>
<td>GLIFT</td>
<td>4,277,412</td>
<td>7.6X</td>
<td>4X</td>
<td>5X</td>
</tr>
<tr>
<td>Caisson</td>
<td>1,171,448</td>
<td>2X</td>
<td>6.36</td>
<td>1.14X</td>
</tr>
<tr>
<td>Sapper</td>
<td>582,333</td>
<td>1.04X</td>
<td>5.57</td>
<td>1X</td>
</tr>
</tbody>
</table>
Caisson [PLDI’11]

- HDL with static enforcement of GLIFT properties
  - Avoid overhead of shadow logic
- Extension to Verilog with static type systems and hierarchical state machine modeling
- Type system prevents illegal data flow

\[
x := y; \quad \rightarrow \quad \begin{array}{ll}
x &: H \quad & y &: L \\
x &: L \quad & y &: H
\end{array}
\]
• Problem: untrusted children can’t transition back to trusted parent
• Solution: child states can “fall” back to parent
## Static vs Dynamic Techniques

<table>
<thead>
<tr>
<th>Processor</th>
<th>Area (µm²)</th>
<th>Delay (ns)</th>
<th>Power (mW)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Processor</td>
<td>560,687</td>
<td>5.57</td>
<td>2.79</td>
<td></td>
</tr>
<tr>
<td>GLIFT</td>
<td>4,277,412</td>
<td>7.6X</td>
<td>4X</td>
<td>5X</td>
</tr>
<tr>
<td>Caisson</td>
<td>1,171,448</td>
<td>2X</td>
<td>1.14X</td>
<td>2X</td>
</tr>
<tr>
<td>Sapper</td>
<td>582,333</td>
<td>1.04X</td>
<td>5.57</td>
<td>2.93</td>
</tr>
</tbody>
</table>

- Registers and memory are statically typed
  - Commonly requires duplication of registers and memory (eg Hi and Lo)
Register and Memory Duplication

- Generally need multiple copies of data for different security levels

```verilog
reg a_h, b_h, c_h : H;
reg a_l, b_l, c_l : L;

if (mode)
    a_h = b_h + c_h;
else
    a_l = b_l + c_l;
```
Sapper [ASPLOS’14]

• Dynamic tracking of security tags
• No type system
• Low-overhead dynamic tracking and enforcement
  ▪ Automatically generated by compiler
From static to dynamic

- In Caisson:

  ```
  reg a_h, b_h, c_h : H;
  reg a_l, b_l, c_l : L;
  if (mode)
      a_h = b_h + c_h;
  else
      a_l = b_l + c_l;
  ```

- In Sapper:

  ```
  reg a, b, c : T;
  a = b + c;
  ```
Security Tags

• Data in the design are associated with security tags (as dynamic types that will persist after fabrication)
• These tags are tracked on the fly and checked against policy violations
Violation Handling

- Designers can provide user-defined actions for violation handling

\[
\text{command otherwise secure action} \quad \text{if (derived condition)} \\
\text{command; else secure action;}
\]
Manipulating Tags

\[ \text{setTag}(r, E) \]

- The security level of any data can only be changed under a context whose level is not higher than the data’s
- When data is downgraded (e.g., changed from High to Low) the data is automatically zeroed instantly to avoid leakage.
## Summary

<table>
<thead>
<tr>
<th>Processor</th>
<th>Area (µm²)</th>
<th>Delay (ns)</th>
<th>Power (mW)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Processor</td>
<td>560,687</td>
<td>5.57</td>
<td>2.79</td>
<td></td>
</tr>
<tr>
<td>GLIFT</td>
<td>4,277,412</td>
<td>21.62</td>
<td>4X</td>
<td>5X</td>
</tr>
<tr>
<td>Caisson</td>
<td>1,171,448</td>
<td>6.36</td>
<td>1.14X</td>
<td>2X</td>
</tr>
<tr>
<td>Sapper</td>
<td>582,333</td>
<td>5.57</td>
<td>1X</td>
<td>1.05X</td>
</tr>
</tbody>
</table>
Case Study: Provable non-interference in a Network on Chip
Packets from certain domains should not affect the delivery of packets of other domains.

- Fault Containment
- Preventing timing channels (side-channel attacks) [NOCS 2012]
SurfNoC Schedule

• By careful scheduling, we can reduce latency overhead
• 16-node half mesh network
• Waves (or pipelines) across routers makes sure that multiple domains make progress each cycle.
• Packet is allowed to traverse a channel only in certain time slots according to its domain.
• Wait time between hopping dimensions depends on number of domains.
• Latency overhead depends on
  ▪ Number of domains
  ▪ Number of dimensions
• Example: Three domains
  ▪ Black and grey (0.25 of BW)
  ▪ White (0.5 of BW)
Careful Router Design

- Router Pipeline
  - BW/RC: Buffer write and route computation
  - VA: Virtual channel allocation
  - SA: Switch allocation
  - ST: Switch traversal
  - LT: Link traversal
- Each stage verified for non-interference using GLIFT analysis
Zero-load latency

Number of Domains

Zero-load latency (cycles)

- baseline-small,64
- baseline-fast,64
- tdma,64
- surf,64
Throughput vs. offered load

- baseline-small
- baseline-fast
- tdma
- surf

One domain throughput

One domain offered load (flits/cycles)

2 domains

16 domains
Implementation

• Synthesizable Verilog implementation
• Verified the non-interference property using GLIFT analysis technique.
• For N domains
  ▪ Area
    • Buffers → Linear
    • Crossbar → Linear
  ▪ Power
    • Buffers → Linear
    • Crossbar → Linear
PART 3: STATISTICAL INFORMATION FLOW ANALYSIS
Mutual Information

$H(X)$

$H(Y)$

$H(X|Y)$

$I(X;Y)$

$H(Y|X)$

$H(X,Y)$

$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \left( \frac{p(x,y)}{p(x)p(y)} \right)$
Example: SRAM Remanence

For data collected on several MSP430 microprocessors, observing each cell 1,000 times,

➔ Around 87% of cells (distributed randomly) always started up in the same state (“strongly biased”)
➔ The remaining 13% sometimes started in the 1-state, sometimes in the 0-state (“weakly biased”)
➔ Write a secret millions of times
➔ The bias of “weakly biased” cells changes!

Measuring bias both before and after the secret is written can leak up to ~13% of the written bits per chip.
SRAM Attack

With a basic attack model on data from three chips, up to one third of the bits can be predicted correctly (but with 7% of the predictions incorrect).

With a little image processing:
**Example: Bodytrack benchmark**

- **Input:** Person walking in video
- **Output:** Body position model
- **High MI** corresponds to taking a step
- **Low MI** corresponds to feet together
Applications

- Compute MI between a function’s output and program output
- Security
  - Invest countermeasures (e.g., fuzzing) in high MI computations
- Reliability / approximate computing
  - Invest energy in more precise computation in high MI computations
- Build a program analysis tool

(Credit: Michael Carbin)
Recap

- 2004: ISA-level information flow tracking
  - Minos (Micro04), Aegis, Taintcheck, Rifle
- 2005-2006: Systems leveraging IFT
  - DACODA (CCS05)
  - Temporal Search (ASPLOS06)
- 2007-2008: IFT follow-on architectures
  - SHIFT (ISCA08), Raksha, SIFT
- 2009-2013: Gate-level information flow
  - GLIFT (ASPLOS09, TopPick10)
  - Execution Leases (Micro09)
- 2010-2013: Hardware Description Languages
  - Caisson (PLDI11)
  - Sapper (ASPLOS14)
- 2011-2014: Trustworthy Hardware Designs
  - *logic (ISCA11)
  - SurfNOC (ISCA13, TopPick14)
- Future: Statistical information flow analysis