Samurai: Protecting Critical Data in Unsafe Languages

Karthik Pattabiraman, U. Illinois (Urbana Champaign)
Vinod Grover, NVIDIA Corp.
Ben Zorn, Microsoft Research
Motivation: Emerging Platforms

Fault-tolerance necessary for platform dependability

(1) Little or no isolation among components for functionality;
(2) Support for third-party components for extensibility
The Problem: A Dangerous Mix

Danger 1: Flat, uniform address space

Danger 2: Unsafe programming languages

Danger 3: Unrestricted 3rd party code

Result: corrupt data, crashes security risks

Ben Zorn, Microsoft Research
Samurai: Protecting Critical Data in Unsafe Languages
Solution: Prevent or Detect Errors

- **Rewrite in a safe language** (e.g. Java, C#, Cyclone)
  - Considerable investment in programmer time
  - Performance-critical components may still be unsafe

- **Static techniques** (e.g. LINT, SAL/ESP, SAFEcode)
  - Require the entire code to be statically checked
  - May be subject to false-positives (wrong detections)

- **Dynamic techniques** (CCured, CRED, etc.)
  - All or nothing – issues with 3rd party code
  - High performance overheads
Alternative: Tolerate Errors at Runtime

- Failure oblivious computing [Rinard]
  - Data structure healing (unsound)
  - Boundless Memory Blocks
- Rx [Qin et al.] – checkpoint and restart
- DieHard [Berger et al.] – randomize, overprovision, replicate
- Critical memory / Samurai
Observations to Take Away…

- Memory is a weak abstraction
  - We can strengthen it

- Some data is more important
  - Programmers know it…
  - … and program accordingly

- No code stands alone
  - Solutions have to be compatible with / handle external code
Critical Memory

■ Approach
  ▶ Identify **critical program data**
  ▶ Protect it with **isolation & replication**

■ Goals:
  ▶ **Harden** programs from both SW and HW errors
    ■ Unify existing ad hoc solutions
  ▶ Enable **local reasoning** about memory state
    ■ Leverage powerful static analysis tools
  ▶ Allow **selective, incremental hardening** of apps
  ▶ Provide **compatibility** with existing libraries, apps
Outline

- Motivation
- Critical memory
- Samurai implementation
  - Software-only
  - Probabilistic guarantees
- Evaluation
- Conclusion
Critical Memory: Idea

- Identify and mark some data as “critical
  - Type specifier like `const`
- Shadow critical data in parallel address space (critical memory)
- New operations on critical data
  - `cload` – read
  - `cstore` – write

```c
critical int balance;

balance += 100;
if (balance < 0) {
    chargeCredit();
} else {
    // use x, y, etc.
}
```

![Diagram showing critical data and non-critical data]

Data
- `x`, `y`, other non-critical data
- `balance` critical data

Ben Zorn, Microsoft Research

Samurai: Protecting Critical Data in Unsafe Languages
Critical Memory: Example

```c
int buffer[10];
critical int balance;

balance = 100;
buffer[10] += 200;

....
if (balance < 0) {
  ...
}

map_critical(&balance);
temp1 = 100;
cstore(&balance, temp1);
temp = load ((buffer+40));
store((buffer+40), temp+200);
temp2 = cload(&balance);
if (temp2 > 0) {
  ...
}

balance = 100;
buffer[10] += 200;
.....
```

**Normal Mem**

<table>
<thead>
<tr>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>300</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

**Critical Mem**

<table>
<thead>
<tr>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>
Third-party Libraries/Untrusted Code

- Library code does not need to be critical memory aware
  - If library does not update critical data, no changes required
- If library needs to modify critical data
  - Allow normal stores to critical memory in library
  - Explicitly “promote” on return
- Copy-in, copy-out semantics

```c
critical int balance = 100;
...
library_foo(&balance);
promote balance;
...

// arg is not critical int *
void library_foo(int *arg)
{
    *arg = 10000;
    return;
}
```
Samurai: SCM Prototype

- Software critical memory for heap objects
  - Critical objects allocated with crit_malloc, crit_free

- Approach
  - Replication – base copy + 2 shadow copies
  - Redundant metadata
    - Stored with base copy, copy in hash table
    - Checksum, size data for overflow detection
  - Robust allocator as foundation
    - DieHard unreplicated
    - Randomizes locations of shadow copies
Samurai: Protecting Critical Data in Unsafe Languages

Ben Zorn, Microsoft Research

13

Samurai: Software Prototype

- Two replicas
- Shadow pointers in metadata
- Randomized to reduce correlated errors

Critical load checks 2 copies, detects/repairs on mismatch

Regular store causes memory error!

• Metadata protected with checksums/backup
• Protection is only probabilistic

Update

Base Object

Replica 1

Replica 2

Vote

Heap

Metadata

shadow pointer 1

shadow pointer 2

Critical store writes to all copies
Samurai Experimental Results

- **Implementation**
  - Automated Phoenix pass to instrument loads and stores
  - Runtime library for critical data allocation/de-allocation (C++)

- Protected critical data in 5 applications (mostly SPEC2k)
  - Chose data that is crucial for end-to-end correctness of program
  - Evaluation of performance overhead by instrumentation
  - Fault-injections into critical and non-critical data (for propagation)

- Protected critical data in libraries
  - **STL List Class**: Backbone of list structure (link pointers)
  - **Memory allocator**: Heap meta-data (object size + free list)
Performance Overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>Samurai</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpr</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>crafty</td>
<td>1.08</td>
<td>1.08</td>
</tr>
<tr>
<td>parser</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>rayshade</td>
<td>1.08</td>
<td>1.08</td>
</tr>
<tr>
<td>gzip</td>
<td>2.73</td>
<td>2.73</td>
</tr>
</tbody>
</table>

Samurai: Protecting Critical Data in Unsafe Languages

Ben Zorn, Microsoft Research
Fault Injection into Critical Memory (vpr)

with Samurai

without Samurai

Fault Period (\# of accesses)

Ben Zorn, Microsoft Research

Samurai: Protecting Critical Data in Unsafe Languages
Samurai: STL Class + WebServer

- **STL List Class**
  - Modified memory allocator for class
  - Modified member functions `insert`, `erase`
  - Modified custom iterators for list objects
  - Added a new call-back function for direct modifications to list data

- **Webserver**
  - Used STL list class for maintaining client connection information
  - Made list critical – one thread/connection
  - Evaluated across multiple threads and connections
  - Max performance overhead = 9 %
Samurai: Protecting Allocator Metadata

Performance Overheads

Average = 10%

- espresso
- cfrac
- p2c
- Lindsay
- Boxed-Sim
- Mudlle
- Average

Kingsley  Samurai
Related Work

- Robust Data Structures
  - Design [Bright et al.], [Kant et al.]
  - Specification [Dempsky et al.]

- Memory safety in unsafe languages
  - Static and dynamic methods (numerous, already mentioned)
  - Software fault isolation [Wahbe et al.]

- Recovery
  - Failure oblivious computing [Rinard et al.]
  - Rx [Qin et al.]
  - Sprite’s recovery box [Baker and Sullivan]
  - SafeDrive extension recovery [Zhou et al.]
Summary

- **Critical Memory: abstract memory model**
  - Protect critical data in applications
  - Define special operations: critical load/store
  - Inter-operation with untrusted/trusted libraries
    - Compatible with existing code
    - Enables local reasoning

- **Samurai: heap-based software prototype**
  - Uses replication and forward error-correction
  - Tested on both applications and libraries
    - Performance overheads of less than 10% in many cases, including practical libraries
Questions?

Samurai: Protecting Critical Data in Unsafe Programming Languages

Karthik Pattabiraman, U. Illinois, U.C.
Vinod Grover, NVIDIA Corp.
Ben Zorn, Microsoft Research
Hardware Trends (1) Reliability

- Hardware transient faults are increasing
  - Even type-safe programs can be subverted in presence of HW errors
    - Academic demonstrations in Java, OCaml
  - Soft error workshop (SELSE) conclusions
    - Intel, AMD now more carefully measuring
    - “Not practical to protect everything”
    - Faults need to be handled at all levels from HW up the software stack
  - Measurement is difficult
    - How to determine soft HW error vs. software error?
    - Early measurement papers appearing
Hardware Trends (2) Multicore

- DRAM prices dropping
  - 2Gb, Dual Channel PC 6400 DDR2 800 MHz $85

- Multicore CPUs
  - **Quad-core** Intel Core 2 Quad, AMD Quad-core Opteron
  - **Eight core** Intel by 2008?

- **Challenge:**
  How should we use all this hardware?
Samurai: Applications/Critical Data

- **vpr**: Does place and route on FPGAs from netlist
  - Made routing-resource graph critical

- **crafty**: Plays a game of chess with the user
  - Made cache of previously-seen board positions critical

- **gzip**: Compress/Decompresses a file
  - Made Huffman decoding table critical

- **parser**: Checks syntactic correctness of English sentences
  - Made the dictionary data structures critical

- **rayshade**: Renders a scene file
  - Made the list of objects to be rendered critical
Samurai: Performance Measurement

- Measured baseline cost without Samurai \((T_1)\)

- Instrumented all loads and stores in program
  - Measured cost of checking all loads, stores
    - *without Samurai* \((T_2)\)
    - *with Samurai* \((T_3)\)
  - Found fraction of critical loads and stores \((n)\)
  - Estimated cost of checking critical loads and stores \(= n \times (T_2 - T_1)\)

- Measured Samurai overhead as:
  \[ T_1 + n \times (T_2 - T_1) + (T_3 - T_2) \]
## Execution Characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>critical data</th>
<th>critical loads</th>
<th>critical stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpr</td>
<td>37%</td>
<td>0.009 %</td>
<td>0.000043 %</td>
</tr>
<tr>
<td>crafty</td>
<td>100%</td>
<td>0.25%</td>
<td>0.60 %</td>
</tr>
<tr>
<td>parser</td>
<td>0.2%</td>
<td>0.01%</td>
<td>0.00013 %</td>
</tr>
<tr>
<td>rayshade</td>
<td>0.7%</td>
<td>1.91%</td>
<td>0.0000004 %</td>
</tr>
<tr>
<td>gzip</td>
<td>100%</td>
<td>12.8%</td>
<td>0.28 %</td>
</tr>
</tbody>
</table>

4/22/2008 Microsoft Research Talk
Samurai: Fault Injection Methodology

- Injections into critical data
  - Corrupted objects on Samurai heap, one at a time
  - Injected more faults into more populated heap regions (Weighted fault-injection policy)
  - Outcome: success, failure, false-positive

- Injections into non-critical data
  - Measure propagation to critical data
  - Corrupted results of random store instructions
  - Compared memory traces of verified stores
  - Outcomes: control error, data error, pointer error
Fault Injection into Non-Critical Data

<table>
<thead>
<tr>
<th>App</th>
<th>Number of Trials</th>
<th>Control Errors</th>
<th>Data Errors</th>
<th>Pointer Errors</th>
<th>Assertion Violations</th>
<th>Total Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpr</td>
<td>550 (199)</td>
<td>0</td>
<td>203 (0)</td>
<td>1 (0)</td>
<td>2 (2)</td>
<td>203 (0)</td>
</tr>
<tr>
<td>crafty</td>
<td>55 (18)</td>
<td>12 (7)</td>
<td>9 (3)</td>
<td>4 (3)</td>
<td>0</td>
<td>25 (13)</td>
</tr>
<tr>
<td>parser</td>
<td>500 (380)</td>
<td>0</td>
<td>3 (1)</td>
<td>0</td>
<td>0</td>
<td>3 (1)</td>
</tr>
<tr>
<td>rayshade</td>
<td>500 (68)</td>
<td>0</td>
<td>5 (1)</td>
<td>0</td>
<td>1 (1)</td>
<td>5 (1)</td>
</tr>
<tr>
<td>gzip</td>
<td>500 (239)</td>
<td>0</td>
<td>1 (1)</td>
<td>2 (2)</td>
<td>157 (157)</td>
<td>3 (3)</td>
</tr>
</tbody>
</table>