Testudo:
Heavyweight Security Analysis
via Statistical Sampling

Joseph L. Greathouse

Ilya Wagner           David A. Ramos           Gautam Bhatnagar
Todd Austin           Valeria Bertacco          Seth Pettie

Advanced Computer Architecture Laboratory
University of Michigan

MICRO, Lake Como, Italy
November 11, 2008
Bad Software is Everywhere

- NIST: SW errors cost U.S. $60 billion/year as of 2002
- These errors include security vulnerabilities.

"Security bugs are out there, in fact in web apps they're pretty much a plague." - Zeev Suraski, co-creator of PHP
Bad Software is Everywhere

- NIST: SW errors cost U.S. $60 billion/year as of 2002
- These errors include security vulnerabilities.

"Security bugs are out there, in fact in web apps they're pretty much a plague." - Zeev Suraski, co-creator of PHP
Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - Multiple types of tests, runtime protection
  - Extremely high runtime overheads
Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - Multiple types of tests, runtime protection
  - Extremely high runtime overheads
Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - Multiple types of tests, runtime protection
    - Extremely high runtime overheads
Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  + Multiple types of tests, runtime protection
  - Extremely high runtime overheads
Hardware Dynamic Analysis for Security

- DIFT, Raksha, FlexiTaint, et al.
  - Low/no runtime overhead, runtime protection
  - Limited analysis types, complex HW overhead
Hardware Dynamic Analysis for Security

- DIFT, Raksha, FlexiTaint, et al.
  + Low/no runtime overhead, runtime protection
  - Limited analysis types, complex HW overhead
Hardware Dynamic Analysis for Security

- DIFT, Raksha, FlexiTaint, et al.
  + Low/no runtime overhead, runtime protection
  - Limited analysis types, complex HW overhead
Hardware Dynamic Analysis for Security

- DIFT, Raksha, FlexiTaint, et al.
  + Low/no runtime overhead, runtime protection
  - Limited analysis types, complex HW overhead
Testudo: Distributed Dynamic Analysis

- Split analysis across population of users
  - Low HW cost, low runtime overhead, runtime information from the field
  - Analysis only
Testudo: Distributed Dynamic Analysis

- Split analysis across population of users
  - Low HW cost, low runtime overhead, runtime information from the field
  - Analysis only

Users running at full speed
Testudo: Distributed Dynamic Analysis

- Split analysis across population of users
  - Low HW cost, low runtime overhead, runtime information from the field
  - Analysis only

- Users running at full speed

- Potential problems encountered
Testudo: Distributed Dynamic Analysis

- Split analysis across population of users
  - Low HW cost, low runtime overhead, runtime information from the field
  - Analysis only

Users running at full speed

Developer

Reported before they are exploited
Heavyweight Dynamic Analysis

- Heavyweight analyses use shadow values.
- Shadow values hold meta-information about associated memory values.
- Can be used to detect potential errors without an active exploit.
Example of Heavyweight Analysis

**Code:**

```c
int sample(int a[8]){  
    int x = read_in();  
    int y = x + 1;  
    int z = x * 2;  
    print a[x];  
    if(y>0&&y<8)  
        print a[y];  
    return a[z];  
}
```

**Dataflow:**

- **Key:**
  - □ = has shadow value

**Memory:**

- Memory Location
  - `a[0]`
  - `x`
  - `y`
  - `z`

- Shadow Value
  - ...

...
Example of Heavyweight Analysis

Code:

```c
int sample(int a[8]) {
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if (y > 0 && y < 8)
        print a[y];
    return a[z];
}
```

Memory:

Dataflow:

Key:
- = has shadow value

I/O

x = read_in()
Example of Heavyweight Analysis

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Dataflow:**

Key:
- Red = has shadow value

**Memory:**

- **Memory Location**
  - a[0]
- **Shadow Value**
  - x: 6
  - y: 7
  - z

**I/O**

- x = read_in()

- y = x + 1
Example of Heavyweight Analysis

Code:

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Memory:

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Shadow Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>UT</td>
</tr>
<tr>
<td>y</td>
<td>UT</td>
</tr>
<tr>
<td>z</td>
<td>UT</td>
</tr>
</tbody>
</table>

Dataflow:

Key:
- = has shadow value

I/O

- x = read_in()
- y = x + 1
- z = x * 2
Example of Heavyweight Analysis

Code:
```c
int sample(int a[8]){  
    int x = read_in();  
    int y = x + 1;  
    int z = x * 2;  
    print a[x];  
    if(y>0&&y<8)  
        print a[y];  
    return a[z];  
}
```

Dataflow:
- Key: = has shadow value
- I/O
- x = read_in()
- y = x + 1
- z = x * 2

Memory:
- a[0]
  - Memory Location:  
  - Shadow Value: UT
- x
  - Memory Location:  
  - Shadow Value: 6
- y
  - Memory Location:  
  - Shadow Value: 7
- z
  - Memory Location:  
  - Shadow Value: 12
Example of Heavyweight Analysis

Code:
```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Memory:
- Memory Location
- Shadow Value

Dataflow:
- Key: = has shadow value
- I/O
- x = read_in()
- y = x + 1
- z = x * 2

Key:
- UT = has shadow value
- T = has shadow value
Example of Heavyweight Analysis

**Code:**

```c
int sample(int a[8]) {
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if (y > 0 && y < 8) {
        print a[y];
    }
    return a[z];
}
```

**Memory:**

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Shadow Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>UT</td>
</tr>
</tbody>
</table>

**Dataflow:**

- **Key:**
  - Red = has shadow value

- **I/O**
  - x = read_in()
  - y = x + 1
  - z = x * 2
Contributions of Testudo

- **Reduce hardware complexity:** Shadow storage is a small, constant size. No out-of-core changes.

- **Reduce runtime overhead:** Divide work across users to reduce overhead for each individual.

- **Increase analysis quality:** Large user population allows analysis of large, varying state spaces.
Dataflow Sampling Example: 1st User

**Code:**
```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**
- `a[0]`
- `x`
- `y`
- `z`

**Dataflow:**
- **Key:**
  - = shadow value
  - = no shadow value
  - * = globally observed shadow value

**Dataflow Sample:**
- **Input (I/O):**
  - `x = read_in()`
- **Computation:**
  - `y = x + 1`
  - `z = x * 2`
Dataflow Sampling Example: 1\textsuperscript{st} User

Code:

```c
int sample(int a[8]) {
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Memory:

**Key:**
- \(\star\) = shadow value
- = no shadow value
- \(\ast\) = globally observed shadow value

**Dataflow:**

- \(x = \text{read}_\text{in}()\)
- \(y = x + 1\)
- \(z = x * 2\)
- I/O
- Memory:
  - `x`: 6
  - `y`
  - `z`
  - `a[0]`
  - UT

**Memory Location**
- Single shared shadow value
Dataflow Sampling Example: 1st User

**Code:**
```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**
- `a[0]`
- `x` (6)
- `y` (7)
- `z`

**Dataflow:**
- **Key:**
  - Red = shadow value
  - Blue = no shadow value
  - Grey = globally observed shadow value
  - Green = globally observed value

- **I/O**
  - `x = read_in()`
  - `y = x + 1`
  - `z = x * 2`

- **UT**
  - No Replacement
Dataflow Sampling Example: 1st User

Code:
```
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Memory:
```
<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Single shared shadow value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>7</td>
</tr>
<tr>
<td>z</td>
<td>12</td>
</tr>
</tbody>
</table>
```

Dataflow:
- **Key:**
  - Red = shadow value
  - Blue = no shadow value
  - Yellow = globally observed shadow value

I/O
- x = read_in()
- y = x + 1
- z = x * 2

UT
- No Replacement
Dataflow Sampling Example: 1\textsuperscript{st} User

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    //print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Dataflow:**

- **Key:**
  - \(\text{red}\) = shadow value
  - \(\text{blue}\) = no shadow value
  - \(\text{asterisk}\) = globally observed shadow value

**Memory:**

- **Memory Location**
  - `a[0]`
  - `x`
  - `y`
  - `z`

- **Single shared shadow value**
  - `UT`
Dataflow Sampling Example: 1st User

Code:
```c
int sample(int a[8]) {
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8) print a[y];
    return a[z];
}
```

Memory:
```
Memory Location
Single shared shadow value
```

Dataflow:
```
I/O
x = read_in()

Key:  
= shadow value  
= no shadow value  
= globally observed shadow value
```

I/O
x = read_in()

y = x + 1
z = x * 2

Memory Location:
```
a[0]  
x  
y  
z  
```

Key:
```
= shadow value  
= no shadow value  
= globally observed shadow value
```

x = 6
y = 7
z = 12

UT
Dataflow Sampling Example: 2\textsuperscript{nd} User

Code:

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Dataflow:

Key:
- = shadow value
- = no shadow value
* = globally observed shadow value

Memory:

- `a[0]`: Single shared shadow value
- `x`, `y`, `z`
Dataflow Sampling Example: 2nd User

Code:

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Memory:

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Single shared shadow value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

Dataflow:

Key:
- = shadow value
- = no shadow value
* = globally observed shadow value

I/O: x = read_in()

y = x + 1
z = x * 2
Dataflow Sampling Example: 2\textsuperscript{nd} User

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Dataflow:**

- **Key:**
  - Red = shadow value
  - Blue = no shadow value
  - Green = globally observed shadow value

- **I/O:**
  - `x = read_in()`

- **Expressions:**
  - `y = x + 1`
  - `z = x * 2`

- **Memory:**
  - `a[0]`
  - `x = 6`
  - `y`
  - `z`

- **Options:**
  - UT

- **Annotations:**
  - Single shared shadow value
Dataflow Sampling Example: 2nd User

Code:
```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Dataflow:

<table>
<thead>
<tr>
<th>I/O</th>
<th>x=read_in()</th>
<th>y = x + 1</th>
<th>z = x * 2</th>
</tr>
</thead>
</table>

Key:
- Red = shadow value
- Blue = no shadow value
- * = globally observed shadow value

Memory:
- `a[0]` (Single shared shadow value)
- `x` = 6
- `y` = 7
- `z`

UT
Dataflow Sampling Example: 2\textsuperscript{nd} User

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**

- **Memory Location**
  - `a[0]`

- **Single shared shadow value**
  - `x` = 6
  - `y` = 7
  - `z` = 12

**Dataflow:**

- **Key:**
  - Red = shadow value
  - Blue = no shadow value
  - Green = globally observed shadow value

- **I/O**
  - `x = read_in()`

- **x = read_in()**
  - `y = x + 1`
  - `z = x * 2`

- **UT**
Dataflow Sampling Example: 2nd User

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**

- **Memory Location:**
  - a[0]
  - x
  - y
  - z

- **Single shared shadow value:** T

**Dataflow:**

- **Key:**
  - Red = shadow value
  - Blue = no shadow value
  - Green = globally observed shadow value

- **I/O:**
  - x = read_in()

- **x = x + 1**

- **z = x * 2**
Dataflow Sampling Example: 2\textsuperscript{nd} User

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**

- `a[0]`<br>
- `x`<br>
- `y`<br>
- `z`<br>

**Dataflow:**

- `x = read_in()`<br>
- `y = x + 1`<br>
- `z = x * 2`<br>

**Key:**

- Red = shadow value
- Blue = no shadow value
- Green = globally observed shadow value

**I/O:**

- 6<br>
- 7<br>
- 12
Dataflow Sampling Example: 3rd User

Code:

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Memory:

- `a[0]`: Single
- `x`, `y`, `z`: Shared shadow value

Dataflow:

- Key:
  - Red: shadow value
  - Blue: no shadow value
  - *: globally observed shadow value

- `x` is read from I/O, and `y = x + 1`, `z = x * 2` are computed.
- Results are stored in `a[x]`, `a[y]`, and `a[z]` as needed.
Dataflow Sampling Example: 3rd User

**Code:**

```c
int sample(int a[8])
{
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**

- `a[0]` (Single)
- `x` (Primary)
- `y`
- `z`

**Dataflow:**

- **Key:**
  - Red = shadow value
  - Blue = no shadow value
  - Green = globally observed shadow value

- **I/O**
  - `x = read_in()`
  - `y = x + 1`
  - `z = x * 2`

- **UT**
Dataflow Sampling Example: 3rd User

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**

**Key:**
- **=** shadow value
- **=** no shadow value
- **=** globally observed shadow value

**Dataflow:**

- **I/O**
- **x** = read_in()
- **y = x + 1**
- **z = x * 2**

**Memory Location**:

- **a[0]**
- **x**
- **y**
- **z**

**Single shared shadow value**

**UT**

**No Replacement**
Dataflow Sampling Example: 3rd User

Code:
```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Memory:

Dataflow:

Key:
- = shadow value
- = no shadow value
* = globally observed shadow value
Dataflow Sampling Example: 3rd User

Code:
```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

Dataflow:
- **Key:**
  - Red = shadow value
  - Blue = no shadow value
  - Asterisk = globally observed shadow value

Memory:
- **Memory Location**
  - a[0]
  - x
  - y
  - z

- **Single shared shadow value**
  - UT
Dataflow Sampling Example: 3<sup>rd</sup> User

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Single shared shadow value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>7</td>
</tr>
<tr>
<td>z</td>
<td>12</td>
</tr>
</tbody>
</table>

**Dataflow:**

Key:
- **Red** = shadow value
- **Blue** = no shadow value
- **Green** = globally observed shadow value

- **I/O**
- **x = read_in()**
- **y = x + 1**
- **z = x * 2**
- **UT**
Dataflow Sampling Example: 3rd User

**Code:**

```c
int sample(int a[8]){
    int x = read_in();
    int y = x + 1;
    int z = x * 2;
    print a[x];
    if(y>0&&y<8)
        print a[y];
    return a[z];
}
```

**Memory:**

- **Key:**
  - Red = shadow value
  - Blue = no shadow value
  - Green = globally observed shadow value

**Dataflow:**

- `x = read_in()`: Shadow value
- `y = x + 1`: Globally observed shadow value
- `z = x * 2`: Shadow value
- `print a[x]`: Shadow value
- `if(y>0&&y<8)
  print a[y];`: Shadow value
- `return a[z]`: Shadow value

**I/O:**

- **Memory Location:**
  - `a[0]`: Blue
  - `x`: Red
  - `y`: Blue
  - `z`: Red

- **Single shared shadow value:**
  - `UT`
A Pipeline for Testudo

- Extend registers with single shadow bit.
- Also extend pipeline buffers.
- Common in hardware DIFT
A Pipeline for Testudo

- Compute new shadow bits
- Generally hard-coded in some way
- Common in hardware DIFT
A Pipeline for Testudo

- Contains addresses of shadowed variables
- Drop some values when this small cache is full
- Replacement scheme *not* deterministic
A Pipeline for Testudo

- Different actions & opcodes lead to different handler code
- Replaces or augments EX-stage modifications
- Allows for complicated testing and propagation
Experimental Framework

- Insecure programs (with exploits), including:
  - TIFF image engine
  - Eggdrop IRC bot
  - Lynx web browser
  - PDF library
  - Simulated SQL injection

- CACTI v5.0 used for cache estimation.
How many runs will I need?

To see all of a program’s dataflows with high statistical confidence.

Some need few samples with tiny sample caches

Others need a larger cache and/or more runs for good results
Does it scale to complex analyses?

If each shadow operation uses 1000 instructions:

Each execution sees few shadow values

Fewer shadow values reduce overhead

![Graph showing average % overhead for pdf and sql_injct with 1024-entry Sample Cache.]

![Graph showing average % overhead for telnet server benchmark with 1024 entry and 512 entry.]

17.3% vs. 0.3%
Does it scale to complex analyses?

If each shadow operation uses 1000 instructions:

- Each execution sees few shadow values
- Fewer shadow values reduce overhead
  - Tradeoff: need more executions

Graphs showing:
- pdf vs sql_injct for 1024-entry Sample Cache
- Average % Overhead for 1024 entry telnet server benchmark vs 512 entry benchmark
  - 1024 entry benchmark with 17.3% overhead
  - 512 entry benchmark with 0.3% overhead
  - 3500 executions for 1024 entry benchmark
  - 169,000 executions for 512 entry benchmark
How much will it cost?
- ~0 change in clock period of modern CPU
- No overhead outside of the CPU core
- Very low hardware overhead in CPU core

Sample Cache Access Time

<table>
<thead>
<tr>
<th>Entry Size</th>
<th>2.5GHz AMD Phenom</th>
<th>1.4GHz UltraSPARC T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512 entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024 entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024 entry</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sample Cache Area Overhead

<table>
<thead>
<tr>
<th>Entry Size</th>
<th>AMD Phenom</th>
<th>UltraSPARC T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512 entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024 entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024 entry</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusions from Testudo

- Simplified hardware design for dynamic analysis
- Reduced runtime overhead for heavyweight security analysis
- Increased heavyweight dynamic analysis quality

Future Directions

- Adapting Testudo hardware for multicore
- What is the best cache replacement method?
- Can we skip the hardware additions?
Thank you
BACKUP SLIDES

- Picture rights:
  - One of the following Testudo pictures has been removed, but I don’t remember which one.

- Testudo picture 1
  http://www.flickr.com/photos/frield/1254958611/

- Testudo picture 2
  http://www.flickr.com/photos/manel/154985772/

- ’The Art of Debugging …’ and ’The IDA Pro Book’:
  http://nostarch.com/

- Fuzzing for Software Security Testing and Quality Assurance

- Secure Programming with Static Analysis
  copyright Addison-Wesley
Systems for Detecting Security Errors

- **Eyeballs**
  - Disassembly, debugging, fuzz testing, whitehat/grayhat hackers
  - Time-consuming, difficult

- **Static Analysis**
  - Analyze source, formal reasoning methods, compile-time checks
  - Intractable, requires expert input, no system state
Testudo—Distributed Dynamic Debugging

Current Heavyweight Analysis Systems

Testudo: Heavyweight Sampling Analysis