Bunshin: Compositing Security Mechanisms through Diversification

Meng Xu, Kangjie Lu, Taesoo Kim, Wenke Lee

Georgia Institute of Technology
Memory Corruptions Are Costly...
Heartbleed bug 'will cost millions'

Revoking all SSL certificates leaked by Heartbleed will cost millions of dollars, according to Cloudflare, which provides services to website hosts.
June 08, 2017

InfoSec 2017: Memory-based attacks on printers on the rise, says HP

*Increase in use of printers as an attack vector for hackers: recommended that purchasing decisions include security considerations, not just price.*
Name your phone “Nexus 5X %x.%x”
Battle against Memory Errors

Existing security mechanisms: $W \oplus R$, ASLR, CFI

→ Not hard to bypass
Battle against Memory Errors

Existing security mechanisms: $W \oplus R$, ASLR, CFI
→ Not hard to bypass

Protect all dangerous operation using **sanity checks**:  
→ Auto-applied at compile time

```c
void foo(T *a) {
    *a = 0x1234;
}
```

```c
void foo(T *a) {
    if(!is_valid_address(a)) {
        report_and_abort();
    }
    *a = 0x1234;
}
```
# Battle against Memory Errors

<table>
<thead>
<tr>
<th>Memory Error</th>
<th>Main Causes</th>
<th>Defenses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-bound read/write</td>
<td>Lack of length check</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integer overflow</td>
<td>Softbound AddressSanitizer</td>
</tr>
<tr>
<td></td>
<td>Format string bug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bad type casting</td>
<td></td>
</tr>
<tr>
<td>Use-after-free</td>
<td>Dangling pointer</td>
<td>CETS</td>
</tr>
<tr>
<td></td>
<td>Double free</td>
<td>AddressSanitizer</td>
</tr>
<tr>
<td>Uninitialized read</td>
<td>Lack of initialization</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data structure alignment</td>
<td>MemorySanitizer</td>
</tr>
<tr>
<td></td>
<td>Subword copying</td>
<td></td>
</tr>
<tr>
<td>Undefined behaviors</td>
<td>Divide-by-zero</td>
<td>UndefinedBehaviorSanitizer</td>
</tr>
<tr>
<td></td>
<td>Pointer misalignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Null-pointer dereference</td>
<td></td>
</tr>
</tbody>
</table>
Comprehensive Protection: Goal and Reality

• Accumulated execution slowdown
  • Example: Softbound + CETS → 110% slowdown

• Implementation conflicts
  • Example: AddressSanitizer and MemorySanitizer
Comprehensive Protection with Bunshin

• Accumulated execution slowdown
  • Example: Softbound + CETS → 110% slowdown
    • Bunshin: Reduce to 60% or 40% (depends on the config)

• Implementation conflicts
  • Example: AddressSanitizer and MemorySanitizer
    • Bunshin: Seamlessly enforce conflicting sanitizers
The N-Version Way
The N-Version Way

Input → Virtualization → Variant 1 → Variant 2 → Variant 3 → Synchronize Execution & Consolidate Outputs → Output
The N-Version Way

Input (benign)

Virtualization

Variant 1
Variant 2
Variant 3

Synchronize Execution & Consolidate Outputs

Output (consensus)
The N-Version Way

Input (malicious)

Virtualization

Variant 1

Variant 2

Variant 3

Synchronize Execution & Consolidate Outputs

Output (divergence)
An attacker has to *simultaneously* compromise all variants in order to compromise the whole system.
Similar Ideas

- Two variants placed in disjoint memory partitions
  \([N\text{-Variant Systems}]\)

- Two variants with stacks growing in different directions
  \([Orchestra]\)

- Multiple variants with randomized heap object locations
  \([DieHard]\)

- Multiple versions of the same program
  \([Varan, Mx]\)
Bunshin Overview

• Goal:
  • Reduce slowdown caused by security mechanisms
  • Enable different or even conflicting mechanisms
Challenges for Bunshin

- How to generate these variants?
- What properties they should have?
- How to make them appear as one to outsiders?
- What is a “behavior” and what is a divergence?
- What if the sanitizers introduces new behaviors?
- Multi-threading support?
Variant Generation Intuitions

• Scope of protection required → Sanitizers selected

<table>
<thead>
<tr>
<th>Memory Error</th>
<th>Defenses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-bound read/write</td>
<td>Softbound, AddressSanitizer</td>
</tr>
<tr>
<td>Use-after-free</td>
<td>CETS, AddressSanitizer</td>
</tr>
<tr>
<td>Uninitialized read</td>
<td>MemorySanitizer</td>
</tr>
<tr>
<td>Undefined behaviors</td>
<td>UndefinedBehaviorSanitizer</td>
</tr>
</tbody>
</table>

• Instrumented checks by each sanitizer

```c
void foo(T *a) {
    if(!is_valid_address(a) {
        report_and_abort();
    }
    *a = 0x1234;
}

void bar(T *b) {
    if(!is_valid_address(b) {
        report_and_abort();
    }
    *b = 0x5678;
}
```
Variant Generation Principles

- Check distribution
- Sanitizer distribution
Check Distribution

Input

Partition 1
Partition 2
Partition 3
Program

Output

Input

Virtualization

Partition 1
Partition 2
Partition 3

Variant 1
Variant 2
Variant 3

Synchronize Execution & Consolidate Outputs

Output
Sanitizer Distribution

Program

Virtualization

Variant 1

Variant 2

Variant 3

Synchronize Execution & Consolidate Outputs

Input

Output
Cost Profiling

• Calculate the slowdown caused by the sanity checks

```c
void foo(T *a) {
    timing_start();
    if(!is_valid_address(a) {
        report_and_abort();
    }
    *a = 0x1234;
    timing_end();
}
```
Cost Distribution

• Equally distribute overhead to variants so that they execute at the same speed

<table>
<thead>
<tr>
<th>Variant</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo</td>
<td>17%</td>
</tr>
<tr>
<td>Bar</td>
<td>28%</td>
</tr>
<tr>
<td>Baz</td>
<td>35%</td>
</tr>
<tr>
<td>Qux</td>
<td>20%</td>
</tr>
</tbody>
</table>

Variant 1 (52% overhead)

<table>
<thead>
<tr>
<th>Variant</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo</td>
<td>17%</td>
</tr>
<tr>
<td>Baz</td>
<td>35%</td>
</tr>
<tr>
<td>Bar</td>
<td>28%</td>
</tr>
<tr>
<td>Qux</td>
<td>20%</td>
</tr>
</tbody>
</table>

Variant 2 (48% overhead)
Variant Generation Process

Source code

Security mechanisms
(e.g., ASan, MSan, UBSan)

Variant generator

Costs profiling

Overhead distribution

Variant compiling

Variants

full
w/ MSan
w/ ASan
w/ UBSan

selective
Variant Sync Considerations

- What is a behavior and what is a divergence?
  - System call (both order and arguments)

- How to hook it?
  - By patching the system call table with a kernel module

- What if different sanitizers introduce different system calls?
  - Sync only when a program is in its `main` function
  - Do not check system calls for memory management
System Call Synchronization

Partition 1

Userspace
Leader

Partition 2
Follower 1

Partition 3
Follower 2

Kernel

Sync slot
Syscall number
Arguments
Execution result
System Call Synchronization

1. Leader enters syscall

**Kernel**

**Userspace**

- Leader
- Partition 1
- Partition 2
- Partition 3
- Follower 1
- Follower 2

**Syscall number**

**Arguments**

**Execution result**

**sync slot**
System Call Synchronization

Userspace

Kernel

Partition 1
Leader

Partition 2
Follower 1

Partition 3
Follower 2

② Followers enter syscall

Syscall number
Arguments
Execution result

sync slot
System Call Synchronization

Userspace

<table>
<thead>
<tr>
<th>Partition 1</th>
<th>Leader</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partition 2</th>
<th>Follower 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partition 3</th>
<th>Follower 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kernel

- Syscall number
- Arguments
- Execution result
- sync slot

③ Kernel execute the syscall only once
System Call Synchronization

Userspace

Kernel

Leaders

Partition 1
Leader

Partition 2
Follower 1

Partition 3
Follower 2

Followers

Sync slot

Syscall number
Arguments
Execution result

④ Leader fetches syscall result

④ Followers fetch syscall result
Strict and Selective Lockstep

**Userspace**
- Leader

**Kernel**
- Leader writes at the next available slot
- Followers read at their own speed

**Sync ring buffer**
- Partition 1
- Partition 2
- Partition 3
- Follower 1
- Follower 2
Strict and Selective Lockstep

Userspace
- Leader
- Partition 1
- Partition 2
- Partition 3
- Follower 1
- Follower 2

Kernel
- sync ring buffer
- Always strictly synchronized for “write” related system calls
Strict and Selective Lockstep

Selective-locksteps mitigates *address leaks*

Address leak involves a "write" system call and with ASLR enabled, such leak attempt will be captured.

Reduce sync. overhead by 3% - 5%
Multi-threading Support

Before fork

Original Execution group

After fork

New Execution group

Leader

New ring buffer

Follower 1

Follower 2
Multi-threading Support

Works if there is no interleaving between threads
Multi-threading Support

Userspace

Kernel

Leader

Follower 1

Follower 2

Total order of lock acquisition and releases
Multi-threading Support

Works under weak determinism (data race-free programs)

Implementation specific (pthread APIs only)

Total order of lock acquisition and releases
Evaluate Bunshin

- Robustness and Security
- Efficiency and Scalability
- Protection Distribution Case Studies
## Robustness

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Single/Multi-thread</th>
<th>Feature</th>
<th>Pass ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC CPU2006</td>
<td>Single</td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>SPLASH-2x</td>
<td>Multi</td>
<td>CPU Intensive</td>
<td>✔️</td>
</tr>
<tr>
<td>PARSEC</td>
<td>Multi</td>
<td></td>
<td>✗ 6 out of 13</td>
</tr>
<tr>
<td>lighttpd</td>
<td>Single</td>
<td>I/O Intensive</td>
<td>✔️</td>
</tr>
<tr>
<td>nginx</td>
<td>Multi</td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>python, php</td>
<td>Single</td>
<td>Interpreter</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Security

• RIPE Benchmark

<table>
<thead>
<tr>
<th>Config</th>
<th>Succeed</th>
<th>Probabilistic</th>
<th>Failed</th>
<th>Not possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>114</td>
<td>16</td>
<td>720</td>
<td>2990</td>
</tr>
<tr>
<td>AddressSanitizer</td>
<td>8</td>
<td>0</td>
<td>842</td>
<td>2990</td>
</tr>
<tr>
<td>Bunshin</td>
<td>8</td>
<td>0</td>
<td>842</td>
<td>2990</td>
</tr>
</tbody>
</table>

• Real-world CVEs

<table>
<thead>
<tr>
<th>Config</th>
<th>CVE</th>
<th>Exploits</th>
<th>Sanitizer</th>
<th>Detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx-1.4.0</td>
<td>2013-2028</td>
<td>Blind ROP</td>
<td>AddressSanitizer</td>
<td>✓</td>
</tr>
<tr>
<td>cpython-2.7.10</td>
<td>2016-5636</td>
<td>Integer overflow</td>
<td>AddressSanitizer</td>
<td>✓</td>
</tr>
<tr>
<td>php-5.6.6</td>
<td>2015-4602</td>
<td>Type confusion</td>
<td>AddressSanitizer</td>
<td>✓</td>
</tr>
<tr>
<td>openssl-1.0.1a</td>
<td>2014-0160</td>
<td>Heartbleed</td>
<td>AddressSanitizer</td>
<td>✓</td>
</tr>
<tr>
<td>httpd-2.4.10</td>
<td>2014-3581</td>
<td>Null dereference</td>
<td>UndefinedBehaviorSanitizer</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Items</th>
<th>Strict-Lockstep</th>
<th>Selective-Lockstep</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPEC CPU2006</strong> (19 Programs)</td>
<td>Max</td>
<td>17.5%</td>
<td>14.7%</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>1.6%</td>
<td>1.0%</td>
</tr>
<tr>
<td></td>
<td>Ave</td>
<td>8.6%</td>
<td>5.6%</td>
</tr>
<tr>
<td><strong>SPLASH-2X / PARSEC</strong> (19 Programs)</td>
<td>Max</td>
<td>21.4%</td>
<td>18.9%</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>10.7%</td>
<td>6.6%</td>
</tr>
<tr>
<td></td>
<td>Ave</td>
<td>16.6%</td>
<td>14.5%</td>
</tr>
<tr>
<td><strong>lighttpd</strong> 1MB File Request</td>
<td>Ave</td>
<td>1.44%</td>
<td>1.21%</td>
</tr>
<tr>
<td><strong>nginx</strong> 1MB File Request</td>
<td>Ave</td>
<td>1.71%</td>
<td>1.41%</td>
</tr>
</tbody>
</table>
Performance Highlights

- **Low** overhead (5% - 16%) for standard benchmarks
- **Negligible** overhead (<= 2%) for server programs
- Extra cost of ensuring weak determinism is **8%**
- Selective-lockstep saves around **3%** overhead
Scalability - Number of Variants

Ave | Max | Min
---|---|---
0.5 | 0.5 | 0.5
11.2 | 17.2 | 6.6
11.2 | 10.5 | 6.6
17.2 | 20.9 | 11.4
37.6 | 37.6 | 11.4

Number of variants | 2 | 4 | 6 | 8
Scalability - Number of Variants

The number of variants Bunshin can support with a reasonable overhead depends on machine configurations and program characteristics.
Scalability - System Load

- Sync Overhead (%)
  - Number of variants: 2%, 50%, 99%
  - Values: 0.2, 0.8, 1.9

- Average (Ave)
  - Values: 6.4, 4.8, 6.6

- Maximum (Max)
  - Values: 9.7, 9.7, 13

- Minimum (Min)
  - Values: 0.2, 0.8, 1.9

Number of variants: 2%, 50%, 99%
Bunshin works well in all levels of system load (i.e., Bunshin does not require exclusive cores)
Check Distribution - ASan

Overhead (%)

<table>
<thead>
<tr>
<th></th>
<th>Whole</th>
<th>V1</th>
<th>V2</th>
<th>Bunshin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole</td>
<td>57.4</td>
<td>63</td>
<td>65.6</td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td>63</td>
<td>65.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V2</td>
<td>65.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V3</td>
<td>34.8</td>
<td>34.9</td>
<td>37.2</td>
<td>43.1</td>
</tr>
<tr>
<td>Bunshin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sanitizer Distribution - UBSan

Overhead (%)

- Whole: 228
- V1: 124
- V2: 125
- Bunshin: 129

Overhead (%)

- Whole: 228
- V1: 77.2
- V2: 78.7
- V3: 88
- Bunshin: 94.5
Deviation from Optimal - ASan

![Graph showing deviation from optimal values.]
Deviation from Optimal - UBSan

<table>
<thead>
<tr>
<th></th>
<th>Whole</th>
<th>V1</th>
<th>V2</th>
<th>Bunshin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead (%)</td>
<td>228</td>
<td>124</td>
<td>125</td>
<td>129</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td></td>
<td></td>
<td></td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Whole</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>Bunshin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead (%)</td>
<td>228</td>
<td>77.2</td>
<td>78.7</td>
<td>88</td>
<td>94.5</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td></td>
<td>76</td>
<td></td>
<td></td>
<td>76</td>
</tr>
</tbody>
</table>
Reasons for Deviation from Optimal

- Synchronization overhead
- Inaccuracy in profiling
- Suboptimal distribution
- Non-distributable overhead
Unifying LLVM Sanitizers

<table>
<thead>
<tr>
<th>Tool</th>
<th>gobmk</th>
<th>povray</th>
<th>h264ref</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASan</td>
<td>116</td>
<td>158</td>
<td>112</td>
<td>148</td>
</tr>
<tr>
<td>MSan</td>
<td>141</td>
<td>165</td>
<td>191</td>
<td>172</td>
</tr>
<tr>
<td>UBSan</td>
<td>205</td>
<td>246</td>
<td>207</td>
<td>177</td>
</tr>
<tr>
<td>Bunshin</td>
<td>248</td>
<td>248</td>
<td>208</td>
<td>177</td>
</tr>
</tbody>
</table>

Overhead (%)
Unifying LLVM Sanitizers

With an average of 5% more slowdown, Bunshin can seamlessly unify all three LLVM sanitizers.
Limitations and Future Work

- Finer-grained check distribution
- Sanitizer integration
- Record-and-replay
Conclusion

• It is feasible to achieve both comprehensive protection and high throughput with an N-version system

• Bunshin is effective in reducing slowdown caused by sanitizers
  • 107% → 47.1% for ASan, 228% → 94.5% for UBSan

• Bunshin can seamlessly unify three LLVM sanitizers with 5% extra slowdown

https://github.com/sslab-gatech/bunshin

(Source code will be released soon)