Performant Software Hardening under Hardware Support

Ren Ding
Georgia Institute of Technology
Apr 26, 2021
A world of bugs

Vulnerabilities By Year

Microsoft’s February 2020 Patch Tuesday fixes 99 security bugs
This is one of Microsoft’s biggest Patch Tuesday known to date.

Microsoft November 2020 Patch Tuesday arrives with fix for Windows zero-day
The Microsoft November 2020 Patch Tuesday fixes 112 vulnerabilities, 24 of which are remote code execution (RCE) bugs.

Microsoft December 2020 Patch Tuesday fixes 58 vulnerabilities
Fixes for 22 remote code execution vulnerabilities included in this month’s patches.

https://www.cvedetails.com/browse-by-date.php
The stepping-stones to software hardening

- Bug finding
- Runtime Mitigation
- Fault Analysis & Recovery
The stepping-stones to software hardening

**Bug finding**
- FREEDOM [CCS’20]
- HYDRA [SOSP’19]
- DEADLINE [SP’18]
- kAFL [SEC’17]
- Honggfuzz [Google’16]
  ...

**Runtime Mitigation**
- PHMon [SEC’20]
- libmpk [ATC’19]
- uCFI [CCS’18]
- PlatPal [SEC’17]
- HDFI [SP’16]
  ...

**Fault Analysis & Recovery**
- DESEN [NDSS’20]
- CIDER [SP’19]
- REPT [OSDI’18]
- POMP [SEC’17]
  ...


The stepping-stones to software hardening

**Bug finding**
- FREEDOM [CCS’20]
- HYDRA [SOSP’19]
- DEADLINE [SP’18]
- kAFL [SEC’17]
-Honggfuzz [Google’16]

**Runtime Mitigation**
- PHMon [SEC’20]
- libmpk [ATC’19]
- uCFI [CCS’18]
- PlatPal [SEC’17]
- HDFI [SP’16]

**Fault Analysis & Recovery**
- DESEN [NDSS’20]
- CIDER [SP’19]
- REPT [OSDI’18]
- POMP [SEC’17]
Why hardware-based solution?

- **Performance**
  - e.g., Intel VT-x/EPT vs. shadow page tables

- **Compatibility**
  - e.g., Intel PT vs. source instrumentation

- **Reliability**
  - codebase $\propto$ vulnerability

---

Why hardware-based solution?

- **Performance**
  - e.g., Intel VT-x/EPT vs. shadow page tables

- **Compatibility**
  - e.g., Intel PT vs. source instrumentation

- **Reliability**
  - codebase \( \propto \) vulnerability
Why hardware-based solution?

- **Performance**
  - e.g., Intel VT-x/EPT vs. shadow page tables

- **Compatibility**
  - e.g., Intel PT vs. source instrumentation

- **Reliability**
  - codebase $\propto$ vulnerability
# Past research and thesis focus

<table>
<thead>
<tr>
<th>Bug finding</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Support to Improve Fuzzing Performance and Precision</td>
<td>CCS’21</td>
</tr>
<tr>
<td>DORF: State-aware Fuzzing Techniques for Remote Procedure Calls</td>
<td>NDSS’21*</td>
</tr>
<tr>
<td>Runtime mitigation</td>
<td></td>
</tr>
<tr>
<td>Efficient protection of path-sensitive control security</td>
<td>USENIX’17</td>
</tr>
<tr>
<td>Fault analysis</td>
<td></td>
</tr>
<tr>
<td>DESENSITIZATION: Privacy-Aware and Attack-Preserving Crash Report</td>
<td>NDSS’20</td>
</tr>
</tbody>
</table>

* in submission

**Hardware-assisted** software hardening techniques for **performant security**
## Past research and thesis focus

<table>
<thead>
<tr>
<th>Bug finding</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Support to Improve Fuzzing Performance and Precision</td>
<td>CCS’21</td>
</tr>
<tr>
<td>DORF: State-aware Fuzzing Techniques for Remote Procedure Calls</td>
<td>NDSS’21*</td>
</tr>
<tr>
<td>Nested virtualization for performant hypervisor fuzzing</td>
<td>Thesis</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime mitigation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient protection of path-sensitive control security</td>
<td>USENIX’17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fault analysis</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DESENSITIZATION: Privacy-Aware and Attack-Preserving Crash Report</td>
<td>NDSS’20</td>
</tr>
</tbody>
</table>

**Hardware-assisted** software hardening techniques for **performant security**
Efficient Protection of Path-Sensitive Control Security
What is control flow?

- The order of instruction execution
- Only limited sets of valid transitions
What is control hijacking?

- An exploitable vulnerability
- Subvert control transfer to unexpected targets
  - Code injection attack
  - ROP/return-to-libc attack
Control flow integrity (CFI)

- Lightweight
- Runtime enforcement
- Pre-computed valid sets
  - Points-to analysis
- Limitations
  - Over-approximation for soundness
Motivating example

- Parse request
- Assign <handler> function pointer
  - if ADMIN: priv
  - else: unpriv
- Strip request arguments
- Handle request

```c
void dispatch() {
  void (*handler)(struct request *) = 0;
  struct request req;

  while(1) {
    parse_request(&req);
    if (req.auth_user == ADMIN) {
      handler = priv;
    } else {
      handler = unpriv;
      // NOTE: buffer overflow
      strip_args(req.args);
    }
    handler(&req);
  }
}
```
Motivating example

```c
void dispatch() {
    void (*handler)(struct request *) = 0;
    struct request req;

    while(1) {
        parse_request(&req);
        if (req.auth_user == ADMIN) {
            handler = priv;
        } else {
            handler = unpriv;
            // NOTE: buffer overflow
            strip_args(req.args);
        }
        handler(&req);
    }
}
```
Limitation of traditional CFI

- Pre-computed valid transfer sets through static analysis
- Lacks dynamic information

```c
void dispatch() {
    void (*handler)(struct request *) = 0;
    struct request req;

    while (1) {
        parse_request(&req);
        if (req.auth_user == ADMIN) {
            handler = priv;
        } else {
            handler = unpriv;
            // NOTE: buffer overflow
            strip_args(req.args);
        }
        handler(&req);
    }
}
```
PITTYPAT: path-sensitive CFI

- At each control transfer, verify points-to set based on a single execution path

```c
void dispatch() {
  void (*handler)(struct request *) = 0;
  struct request req;

  while(1) {
    parse_request(&req);
    if (req.auth_user == ADMIN) {
      handler = priv;
    } else {
      handler = unpriv;
      // NOTE: buffer overflow
      strip_args(req.args);
    }
    handler(&req);
  }
}
```
Challenges

- Collecting executed path information efficiently
- Trace information cannot be tampered with
- Compute runtime points-to relations efficiently and precisely
Our solution: Intel Processor Trace

- **Low-overhead** commodity hardware
  - Compressed packets to save bandwidth
  - PIN/DynamoRIO/QEMU?

- Trace sharing **protected**
  - CR3 filtering
  - HW -> OS -> user-space
  - Source instrumentation?
Our solution: incremental points-to

- **Input:**
  - LLVM IR of target program
  - Mapping between IR and binary
  - Runtime execution trace

- **Output:** points-to relations on a single execution path
PITTYPAT: system overview

- **Monitor module:**
  - Kernel-space driver for Intel PT

- **Analyzer module:**
  - User-space analysis to update points-to relations

- **Optimization for efficiency:**
  - Parallel design
  - Avoid decoding exact conditional branches
  - Only analyze control-relevant functions/instructions
Forward edge points-to size

- The majority of indirect callsites allow only a single valid address
- Not field-sensitive
Performance overhead

Overhead (%)

Benchmarks

Geo. Mean

Overhead (%)

Perlbench
bzip2
gcc
mcf
milc
namd
isabmk
soplex
povray
hmmer
sjeng
libquantum
h264ref
lmb
astar
sphinx3
nginx

12.73
Discussion

- Non-control data corruption cannot be detected
- Not reasoning about field sensitiveness
- Performance vs. accuracy
Hardware Support to Improve Fuzzing Performance and Precision
Fuzzing – a searching problem

- Input space -> program states

- Search **comprehensively**
  - Seed selection
  - Input mutation
  - Feedback collection

- Search **efficiently**
  - More iterations under limited resources
Coverage-guided fuzzing

- Code coverage ≈ program states
- Trace-encoded bitmap
  - basic blocks/edges
- Coverage collection
  - Source-based
  - Binary-only

```
New input I' → Program

"Interesting" input I

mutate

Y

New cov.?
```
Source-based tracing

- When source code is available
- Source instrumentation
  - GCC, Clang

1 # [Basic Block]:
2 # saving register context
3 mov %rdx, (%rsp)
4 mov %rcx, 0x8(%rsp)
5 mov %rax, 0x10(%rsp)
6 # bitmap update
7 mov $0x40a5, %rcx
8 callq __afl_maybe_log
9 # restoring register context
10 mov 0x10(%rsp), %rax
11 mov 0x8(%rsp), %rcx
12 mov (%rsp), %rdx

(a) afl-gcc

1 # preparing 8 spare registers
2 push %rbp
3 push %r15
4 push %r14
5 ...
6 mov %rax, %r14
7 # [Basic Block]: bitmap update
8 movslq %fs:(%rbx), %rax
9 mov 0xc8845(%rip), %rcx
10 xor $0xca59, %rax
11 addb $0x1, (%rcx,%rax,1)
12 movl $0x652c, %fs:(%rbx)

(b) afl-clang
The cost of source instrumentation

Size (MB)

Overhead (%)

baseline instrumented (afl-clang)
Binary-only tracing

- COTS binary, legacy software
- Dynamic binary instrumentation
  - e.g., QEMU, PIN, DynamoRIO, Unicorn
- Static binary rewriting
  - e.g., DynInst, RetroWrite
- Hardware features:
  - e.g., Intel PT, Intel LBR
The cost of binary-relevant schemes

- Dynamic binary instrumentation
  - Performance

- Static binary rewriting
  - Compatibility

- Hardware features
  - Usability
Our solution: SNAP

- **Transparent** support of fuzzing
  - Source-based vs. binary-only

- **Efficient** hardware-based tracing
  - Existing information in CPU pipeline
  - Idle hardware resources

- **Rich** feedback information
  - Micro-architectural program states
The hardware perspective

- RISC-V BOOM core
- Trace Decision Logic
- Branch Update Queue
- Last Branch Queue

1. Fetch Buffer
2. Allocate new entries
3. Target address, prediction result

LDQ (Load Queue)
STQ (Store Queue)
BUQ (Branch Update Queue)
LBQ (Last Branch Queue)
Micro-architectural Optimization

- Memory request aggregation
- Opportunistic bitmap update

![Diagram showing theFetch Stage and Memory Stage with components like Fetch Controller, Branch Predictor, Fetch Buffer, L1 I-Cache, L1 D-Cache, LDQ, STQ, and BUQ. The diagram includes steps for allocating new entries, opportunistic bitmap update, and aggregation.](image)
The OS perspective

- Configuration interface
  - CSR & system calls
- Memory sharing
  - Kernel device driver
- Process management
  - task_struct
Near-zero tracing overhead

- Tracing overhead: 3.14%
- Memory request aggregation rate: 13.47%
- Cache thrashing problem: 1.63%
Improved fuzzing metrics

- **Fuzzing throughput**
  - 228x faster than AFL-QEMU
  - 41% faster than AFL-gcc

- **Runtime coverage**
  - AFL-QEMU covers 23% paths
  - AFL-gcc covers 85% paths
Improved fuzzing metrics

- Fuzzing throughput
  - 228x faster than AFL-QEMU
  - 41% faster than AFL-gcc

- Runtime coverage
  - AFL-QEMU covers 23% paths
  - AFL-gcc covers 85% paths
SNAP is practical

- Area overhead: 4.82%
- Power overhead: 6.53%
- Compatible to most existing fuzzers
Discussion

- Experimental setup with low clock frequency
- No support for kernel coverage, yet
- Not suitable for tracking dynamic code generation
Nested Virtualization for Performant Hypervisor Fuzzing
Cloud services – the new battlefield

- $111 billion revenue in 2020
- 36.5% yearly growth

Overview of virtualization

- Intel VT-x / AMD-v
- Type-I hypervisor
  - e.g., Xen, VMware ESXi, Hyper-V
- Type-II hypervisor
  - e.g., KVM/QEMU, VirtualBox, VMware Workstation
Threat model

- VM integrity violated
  - Kernel- & user-space

- Privileged operations trapped by VMM
  - e.g., MMIO/PIO, hypercall
Dissecting the attack surface

- CPU virtualization
  - VM control structure (VMCS)
  - VMX root mode

- Memory virtualization
  - Extended page table (EPT)

- I/O virtualization
  - QEMU device emulation
Dissecting the attack surface

- CPU virtualization
  - VM control structure (VMCS)
  - VMX root mode

- Memory virtualization
  - Extended page table (EPT)

- I/O virtualization
  - QEMU device emulation
Dissecting the attack surface

- CPU virtualization
  - VM control structure (VMCS)
  - root mode

- Memory virtualization
  - Extended page table (EPT)

- I/O virtualization
  - QEMU device emulation
Dissecting the attack surface

- CPU virtualization
  - VM control structure (VMCS)
  - root mode

- Memory virtualization
  - Extended page table (EPT)

- I/O virtualization
  - QEMU device emulation
Fuzzing non-user application

- Syzkaller [Google’17]
  - Coverage feedback enabled
  - Running multiple inputs at once

- HYPER-CUBE [NDSS’20]
  - Coverage feedback disabled
  - Customized OS
Challenges

- **Determinism**
  - Clean state per execution

- **Throughput**
  - Fast execution speed

- **Compatibility**
  - One solution for all
Nested virtualization to rescue
Nested virtualization to rescue...?

- No proper hardware support
  - Root vs. non-root mode
  - One VMCS in use
The multiplexing design

- L0 serves the intermediate layer
- Complex VM states 😊
- Performance cost 😊
  - Execution overhead – exit multiplication
  - Snapshot overhead – dirty memory multiplication
  - The interleaving effect
Snapshot and recovery

- CPU reset
- Memory reset
- Device reset

![Diagram of snapshot and recovery process]

- Snapshot or restore hypercall
- Reflect L2 VMExit
- Trap L2 VMEntry
- Forward to userland
- Continue L2 VM
- Handle request
Snapshot and recovery

- CPU reset
- Memory reset
- Device reset
Snapshot and recovery

- CPU reset
- Memory reset
- Device reset
Snapshot and recovery

- CPU reset
- Memory reset
- Device reset
Minimal OS

- L1 & L2 OS
  - L1 minimized, but still supports virtualization
  - L2 customized, as fuzzing executor

- Mitigate overhead from both exits and dirty memory
Implementation details

- Snapshot mechanism
  - QEMU v5.0.0
- L0 VM exit handling
  - Linux v5.0
- L2 customized OS
  - crashOS

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of code (LoC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snapshot &amp; restore mechanism (QEMU)</td>
<td>1,317 lines in C</td>
</tr>
<tr>
<td>VM exit trapping and handling (KVM)</td>
<td>97 lines in C</td>
</tr>
<tr>
<td>Fuzzing executor (L2 kernel)</td>
<td>1,146 lines in C</td>
</tr>
<tr>
<td>Fuzzing coordinator (fuzzer)</td>
<td>1,386 lines in Python</td>
</tr>
</tbody>
</table>
Evaluation

Q1. How fast can we improve the fuzzing throughput from baseline?
Q2. How does variation of OS affect overhead?
Q3. Can we find real-world vulnerabilities in existing hypervisors?
Improving fuzzing throughput

- 72x faster than rebooting with Linux kernel
- 9x faster than HYPERCUBE
- OS complexity $\propto 1/\text{throughput}$
Anatomy of overhead

- Execution overhead
  - Exit multiplication

- Snapshot overhead
  - Dirty memory multiplication
Exit Multiplication

- 8.00% exits from Linux defconfig
- 16.20% exits from Linux miniconfig
Dirty memory multiplication

- 27.37% pages from Linux defconfig
- 40.62% pages from Linux miniconfig
Performance cost from dirty memory

Throughput (EXEC/S) vs. Memory Dirty Pages (#)

- **Snapshot**
- **Reboot**

The graph shows a decrease in throughput as the number of memory dirty pages increases.
Performance cost from VM exits
Testing real-world hypervisors

- QEMU & VirtualBox
- Found 14 zero-day bugs

<table>
<thead>
<tr>
<th>CVE</th>
<th>Device</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2020-13361</td>
<td>ES1370 audio</td>
<td>Heap OOB access</td>
</tr>
<tr>
<td>CVE-2020-13362</td>
<td>MegaRAID SAS storage manager</td>
<td>Heap OOB access</td>
</tr>
<tr>
<td>CVE-2020-13659</td>
<td>MegaRAID SAS storage manager</td>
<td>Null pointer dereference</td>
</tr>
<tr>
<td>CVE-2020-13754</td>
<td>Message Signalled Interrupt (MSI-X)</td>
<td>Heap OOB access</td>
</tr>
<tr>
<td>CVE-2020-13791</td>
<td>ATI VGA display</td>
<td>Heap OOB access</td>
</tr>
<tr>
<td>CVE-2020-13800</td>
<td>ATI VGA display</td>
<td>Infinite recursion</td>
</tr>
</tbody>
</table>
Discussion

- Fuzzing precision also matters
- Testing more hypervisor targets
- Adopting advanced bug finding techniques
Conclusion

A hardware-assisted software hardening solution with careful design is beneficial to
- Performance
- Compatibility
- Reliability

This dissertation demonstrates this idea with
- PITTYPAT: an efficient runtime enforcement for path-sensitive control-flow security
- SNAP: a customized hardware platform to enhance the performance of coverage-guided fuzzing
- HYPERSET: a nested virtualization framework for performant hypervisor fuzzing
Acknowledgement

- Georgia Tech
  - Taesoo Kim
  - Wenke Lee
  - Alessandro Orso
  - Brendan Saltaformaggio
  - Chenxiong Qian
  - Wen Xu
  - Yonghae Kim
  - Fan Sang
  - Gururaj Saileshwar
  - Hanqing Zhao

- Oregon State University
  - Yeongjin Jang

- Penn State University
  - Hong Hu

- UC Riverside
  - Chengyu Song

- Galois, Inc
  - William Harris

- CUHK
  - Wei Meng

- My Family
  - Hong Ding
  - Yan Wang
  - Qingqing Tan
  - Udon & Sushi
Thank you!

rding@gatech.edu