Predictable and Secure Computing Infrastructure for Intelligent Cyber-Physical Systems

2/14/2020
Heechul Yun
Associate Professor, EECS
University of Kansas
Intelligent Cyber-Physical Systems

- Cyber Physical Systems (CPS)
  - Cyber (Computer) + Physical (Plant)
- Real-time
  - Control physical process in real-time
- Safety-critical
  - Can harm people/things
- Intelligent
  - Can function autonomously
Cost, Size, Weight, and Power Constraints

• Maximum performance with minimal resources
  – Cannot afford too many or too power hungry systems

Figure source: Robert Leibinger, “Software Architectures for Advanced Driver Assistance Systems (ADAS),” OSPERT, 2015
Trends in Automotive E/E Systems

Modern System-on-Chip (SoC)

- Integrate multiple cores, GPU, accelerators
- Good performance, cost, size, weight, power
- But...
Challenge: Time Predictability

- Many important hardware resources are *shared*
- Each is locally optimized for average performance
- Software has limited ability to reason and control
- Unpredictable software execution timing
Why Predictable Timing?

- Computing time matters for temporal correctness
  - Logically correct output at wrong time is a fault
Effect of Co-Scheduling

https://youtu.be/Jm6KSDqlqiU
Automotive Industry Challenges

• “Automotive industry is transitioning from μC toward μP based platforms”

• “Interference effects (on μPs) are more severe by orders of magnitude compared to μC platforms”

• “Goal for automotive systems engineering: predictable real-time behavior on high-performance platforms”

(*) Arne Hamann (Bosch), “Industrial Challenges: moving from classical to high-performance real-time systems.” In Waters 2019
Certification Challenges in Aviation

Certification Authorities Software Team (CAST)

Position Paper
CAST-32A

Multi-core Processors

COMPLETED November 2016 (Rev 0)

https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf
CAST-32A: Multicore-Processors

• A position paper by FAA and other certification agencies on multicore

• Discuss interference channels of multi-core that affect software timing

• Suggestion 1: disable all but one core

• Suggestion 2: provide evidence that all interference channels are taken care of ("robust partitioning") \( \rightarrow \) nobody can (yet)
Timing Matters for Security

Measurable timing differences in accessing shared hardware resources can leak secret
Fundamental Challenge: Isolation

• Traditionally about memory isolation
  – Prevent unauthorized access to memory
  – Hardware support: MPU, MMU

• What we need
  – Prevent timing influence between domains
  – Not only for real-time systems
  – But also for security

¹ Q Ge, Y Yarom, T Chothia, G Heiser. "Time Protection: the Missing OS Abstraction". In EuroSys, 2019
My Research

• Build **predictable and secure computing infrastructure** for the next generation of intelligent Cyber Physical Systems (CPS).

Research Sponsors

[NSF logo]  [National Security Agency logo]  [NASA logo]

Equipment Sponsors

[intel logo]  [NVIDIA logo]  [AWS logo]
Agenda

• Part 1. Time predictable software on COTS hardware

• Part 2. Hardware/software collaboration for time predictability and security
What are the (hidden) sources of interference of COTS hardware?


Shared Hardware Resources

- Lots of important hardware resources are shared
- Hardware makes allocation/scheduling decisions without knowing what software wants/does
- Existing partitioning techniques are not sufficient
Cache Partitioning

• Eliminate unwanted cache-line evictions

• Can be done in either SW or HW
  – Software: page coloring
  – Hardware: way partitioning

• Common assumption
  – Cache partitioning $\rightarrow$ performance isolation

• Not necessarily true for non-blocking caches
Non-Blocking Cache

• Can serve cache hits under multiple cache misses
  – Essential for multicore performance

(*) D. Kroft. “Lockup-free instruction fetch/prefetch cache organization,” ISCA’81
Non-Blocking Cache

- Miss Status Holding Registers
  - Track outstanding cache misses.

- Writeback Buffer
  - Holds evicted dirty lines (writebacks).
  - Prevents cache refills from waiting.

- Cache internal structures (MSHRs, writeback buffer) can have huge timing impacts if they are exhausted


Cache Blocking

• Happens if MSHRs/WBBuffer become full
  – The cache is **locked up**
  – Subsequent accesses---including **cache hits**---to the cache **stall** and have to wait until the pending misses are completed
  – A pending miss costs **100’s of CPU cycles** to complete (access to DRAM is slow)
  – We will see the impact of this in later experiments
Threat Model

- Attacker’s goal: increase the victim’s task execution time
- The attacker is on different core/memory/cache partition
- The attacker can only execute non-privileged code.
Cache DoS Attacks

- Denial-of-Service (DoS) attacks targeting internal hardware structures of a shared cache.
  - Block the cache $\rightarrow$ delay the victim’s execution time

```c
for (i = 0; i < mem_size; i += LINE_SIZE)
{
    sum += ptr[i];
}
```

Read Attacker (target MSHRs)

```c
for (i = 0; i < mem_size; i += LINE_SIZE)
{
    ptr[i] = 0xff;
}
```

Write Attacker (target WBBuffer)
Effects of Cache DoS Attacks

- Observed worst-case: >300X (times) slowdown
  - On popular in-order multicore processors
  - Due to contention in cache internal buffers
Effects on EEMBC and SD-VBS

- Cache DoS attacks are effective on real-world benchmarks
- LLC sensitive SD-VBS benchmarks are more susceptible
Summary

• Cache internal hardware structures (MSHRs, WriteBack buffer) are viable DoS attack vectors in multicore platforms.

• Traditional cache partitioning may not be effective to defend against cache DoS attacks targeting these internal hardware structures.
How to improve time predictability of software on COTS hardware?


Parallel Real-Time Tasks

- Many emerging workloads in AI, vision, robotics are parallel real-time tasks

---

DNN based real-time control

Effect of parallelization on DNN control task

![Diagram showing the effect of parallelization on DNN control task](image)

 Observations

• Constructive sharing (Good)
  – Between threads of a single parallel task

• Destructive sharing (Bad)
  – Between threads of different tasks

• Goal: analyzable and efficient parallel real-time task scheduling framework for multicore
  • By avoiding destructive sharing
Gang Scheduling

• Schedule all threads of a parallel task *only if* enough cores are available

• Gang-FTP (fixed task priority)
  – Highest priority task $\tau_i$ on the first $h_i$ available cores (if exist) among the active (ready) tasks.

$$\tau_i = < h_i , C_i , T_i >$$
RT-Gang

- Restricted from Gang-FTP: One gang at a time
  - Eliminate inter-task interference by construction
  - Allow best-effort co-scheduling with bandwidth limit
  - Cons: limited real-time task utilization

RTG-Sync

• Statically group RT tasks as a “virtual gang”
  – All members are synchronously released and scheduled
  – Improve real-time task schedulability
RTG-Sync

- Gang scheduler + bandwidth limiter + cache partitioning, implemented in Linux kernel
- Middleware for virtual gang management
Schedulability Analysis

- RTG-Sync achieves better analytic schedulability
  – Compared to Gang-FTP, Threaded (Linux baseline)
Case Study: DeepPicar*

- A low cost, small scale replication of NVIDIA’s DAVE-2
- Uses the exact same DNN
- Runs on a quad-core SoC in real-time

Experiment Setup

• NVIDIA Jetson TX2 (4 ARM cores)
• Parallel taskset
  – DeepPicar DNN control tasks, Parboil, IsolBench

<table>
<thead>
<tr>
<th>Task</th>
<th>WCET (ms)</th>
<th>Period (ms)</th>
<th># of Threads</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_{BWT}^{RT}$</td>
<td>50.0</td>
<td>100.0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$\tau_{DNN}^{RT} - 1$</td>
<td>8.2</td>
<td>50.0</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>$\tau_{DNN}^{RT} - 2$</td>
<td>8.2</td>
<td>50.0</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>$\tau_{eutep}^{BE}$</td>
<td>$\infty$</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>$\tau_{lbm}^{BE}$</td>
<td>$\infty$</td>
<td>N/A</td>
<td>2</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Execution Time Distribution

![Graph showing CDF of DNN Job Duration with μ values for RT-Gang (μ = 8.5), RTG-Sync (μ = 9.0), and Linux (μ = 11.3).]
Execution Trace

Annotated outputs of *KernelShark* tool
Effect of RT-Gang/RTG-Sync

https://youtu.be/pk0j063cUAs
Summary

• Parallel real-time task scheduling
  – Hard to analyze on COTS multicore
  – Due to interference in shared memory hierarchy

• RT-Gang and RTG-Sync
  – *Analyzable* and *efficient* parallel real-time gang scheduling framework, implemented in Linux
  – Avoid and bound interference by design
  – Simple and achieves better analytic schedulability
Agenda

• Part 1. Time predictable software on COTS hardware

• Part 2. Hardware/software collaboration for time predictability and security
How to improve time predictability of high-performance processors?


RISC-V SoC Testbed

- Full-featured RISC-V cores (in-order/out-of-order) + hardware DNN accelerator on Amazon FPGA
  - Run Linux, YOLO v3 object detection

Source: NVIDIA, “The Nvidia Deep Learning Accelerator”
YOLOv3 Object Detection
YOLOv3 Performance

- **Rocket**: 4 Rocket cores (baseline)
- **NVDLA+Rocket**: baseline + NVDLA
- **Xeon**: E5-2658 v3 (24 cores/48 threads)
- **Titan Xp**: Pascal arch, 3840 CUDA cores
One of the best ways to get started is to dive right in with object detection using YOLOv3 on NVDLA with RISC-V and FireSim in the cloud.

...  

 curing clone [https://github.com/CSL-KU/firesim-nvdla](https://github.com/CSL-KU/firesim-nvdla)
Bandwidth Regulation Unit (BRU)

- Regulate per-core/group memory bandwidth
- Drop-in addition to existing processor design
Bandwidth Regulation Unit (BRU)

• Access regulation
  – Regulate cache misses

• Writeback regulation
  – Regulate cache write-back

• Group regulation
  – Multiple cores can be regulated as a group
Dual-core BOOM with BRU

- BOOM: high-performance out-of-order RISC-V core
- Cadence synthesis result at 7nm node
- Less than 2% impact on max. frequency
- Less than 0.2% space overhead

**Dual-Core BOOM Chip Area Breakdown**

<table>
<thead>
<tr>
<th>Modules</th>
<th>Area ($\mu m^2$)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRU</td>
<td>4,669</td>
<td>0.19%</td>
</tr>
<tr>
<td>Boom Core $\times 2$</td>
<td>2,309,681</td>
<td>92.41%</td>
</tr>
<tr>
<td>Others (System Bus, Manager, etc.)</td>
<td>184,950</td>
<td>7.40%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2,499,300</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>
Effects of BRU

w/o BRU

(a) disparity

(b) localization

(c) svm

w/ BRU regulation (@320MB/s budget, 100ns period)

- BRU = MemGuard in hardware + alpha
Summary

• BRU
  – A synthesizable hardware IP that regulates memory traffic at the source (cores)
  – Demonstrates the feasibility of fast AND predictable processors

• Future work
  – Accelerator regulation support
  – More software/hardware co-design
How to improve security of high-performance processors?

Speculative Execution Attacks

- Attacks exploiting microarchitectural **side-effects** of executing speculative (transient) instructions
- Recover secrets by measuring **timing differences** caused by the side-effects

<table>
<thead>
<tr>
<th>Attack</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variant 1 (Spectre) [16]</td>
<td>Bounds Check Bypass</td>
</tr>
<tr>
<td>Variant 1.1 [15]</td>
<td>Bounds Check Bypass Store</td>
</tr>
<tr>
<td>Variant 1.2 [15]</td>
<td>Read-only Protection Bypass</td>
</tr>
<tr>
<td>Variant 2 (Spectre) [16]</td>
<td>Branch Target Injection</td>
</tr>
<tr>
<td>Variant 3 (Meltdown) [18]</td>
<td>Supervisor Protection Bypass</td>
</tr>
<tr>
<td>Variant 3a [12]</td>
<td>System Register Bypass</td>
</tr>
<tr>
<td>Lazy FP [24]</td>
<td>FPU Register Bypass</td>
</tr>
<tr>
<td>Variant 4 [9]</td>
<td>Speculative Store Bypass</td>
</tr>
<tr>
<td>ret2spec [20]</td>
<td>Return Stack Buffer</td>
</tr>
<tr>
<td>L1 Terminal Fault [11, 26]</td>
<td>Virtual Translation Bypass</td>
</tr>
</tbody>
</table>
Spectre Attack (Variant 1)

```c
if(x < array1_length){
    val = array1[x];
    tmp = array2[val*512];
}

.........
```

- Assume $x$ is under the attacker’s control
- Attacker trains the branch predictor to predict the branch is in-bound
Spectre Attack (Variant 1)

```java
if (x < array1_length) {
    val = array1[x];
    tmp = array2[val*512];
}
```

1. [ACCESS]

- Speculative execution of the first line accesses the secret (val)
Spectre Attack (Variant 1)

```java
if(x < array1_length){
    val = array1[x];
    tmp = array2[val*512];
}
```

2. [TRANSMIT]

• Speculative execution of the second, secret dependent load **transmits** the secret to a microarchitectural state (e.g., cache)
Spectre Attack (Variant 1)

```java
if(x < array1_length){
    val = array1[x];
    tmp = array2[val*512];
}
```

- Attacker receives the secret by measuring access timing differences (cache hit vs. miss) among the elements in the probe array
  - Various timing channels exist (e.g., cache)
Existing Software Mitigation

```c
if(x < array1_length){
    __mm_lfence();
    val = array1[x];
    tmp = array2[val*512];
}
```

- Manually stop speculation
  - By inserting `__lfence` instructions [Intel, 2018]
  - Or by introducing additional data dependencies [Carruth, 2018]
  - Error prone, high programming complexity, performance overhead
Existing Hardware Mitigation

- Hide speculative execution
  - By buffering speculative results into additional "shadow" hardware structures
  - High complexity, high overhead (performance, space)

InvisiSpec [Yan et al., MICRO’18]
SafeSpec [Khasawneh et al., DAC’19]
SpectreGuard

• Data-centric software/hardware co-design
  – Software tells hardware what data (not code) needs protection
  – Hardware selectively protects the identified data from Spectre attacks

• Key observations
  – Not all data is secret
  – Not all speculative loads leak secret

Obs. 1: Not All Data Is Secret

• Non-sensitive data
  – Most program code, data
  – Optimize for performance

• Sensitive (secret) data
  – Cryptographic keys, passwords, ...
  – Optimize for security

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attacker’s controlled data</td>
</tr>
<tr>
<td>AES encryption table</td>
</tr>
<tr>
<td>Other public information</td>
</tr>
<tr>
<td>RSA private key</td>
</tr>
<tr>
<td>Bank account information</td>
</tr>
<tr>
<td>Other secret data</td>
</tr>
</tbody>
</table>
Obs. 2: Not All Speculative Loads Leak Secret

```java
if(x < array1_length){
    val = array1[x];
    tmp = array2[val*512];
}
..........  
1. [ACCESS]  
2. [TRANSMIT]
```

- The first load does **NOT** leak secret
- The second, **secret dependent load** leaks the secret
- Delay the secret dependent load until *after* the branch is resolved
SpectreGuard Approach

- Step 1: Software tells OS what data is secret
- Step 2: OS updates the page table entries
- Step 3: Load of the secret data is identified by MMU
- Step 4: secret data forwarding is delayed until safe
Results of SPEC2006 Benchmarks

- Good protection at low controllable overhead
- SpectreGuard enables targeted security and performance trade-offs
Summary

• Speculative execution attacks
  – Affect all high-performance out-of-order processors
  – Existing software mitigation suffers high programming complexity/overhead
  – Hardware only mitigation is costly

• SpectreGuard
  – A data-centric software/hardware collaborative defense mechanism
  – Low programming effort (identifying secret data, not vulnerable code)
  – Low hardware cost (no additional "shadow" structure)
  – Effective, targeted defense against Spectre attacks

https://github.com/CSL-KU/SpectreGuard
Conclusion

• Smart scheduling and OS support can provide time predictability on COTS hardware

• Small changes in COTS hardware can provide both time predictability and high performance

• Our research develops fundamental computing infrastructure technologies to enable predictable and secure computing for intelligent CPS
Our Vision

- Holistic, cross-layer: from chips to applications
Recent Publications


Full List: [http://www.ittc.ku.edu/~heechul/pub.html](http://www.ittc.ku.edu/~heechul/pub.html)
Thank You!

Acknowledgement:
This research has been supported by NSA Science of Security initiative contract #H98230-18-D-0009 and NSF CNS 1718880, 1815959.