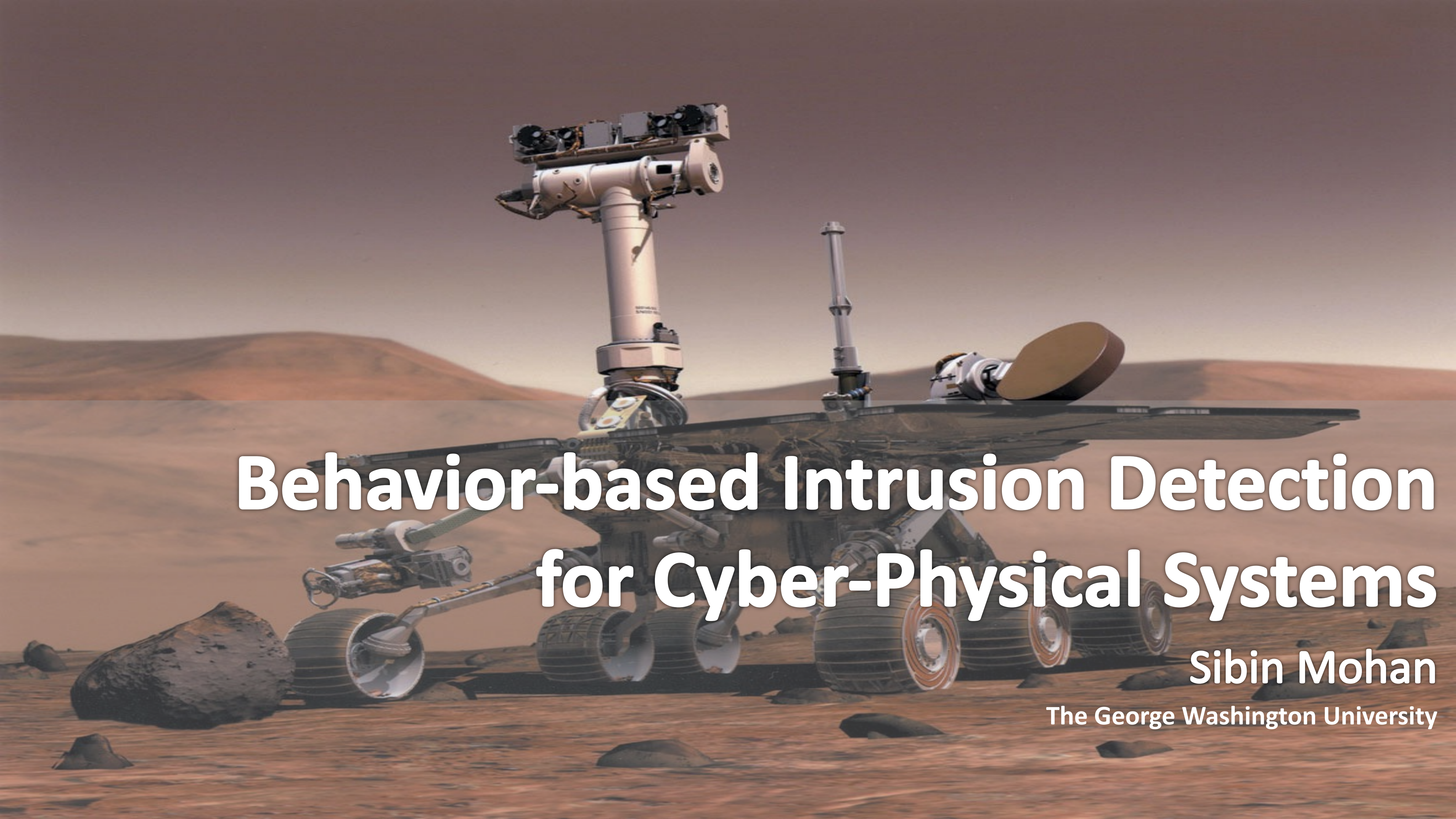
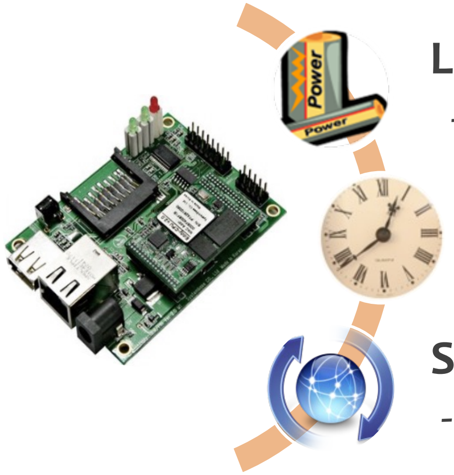


Behavior-based Intrusion Detection for Cyber-Physical Systems

Sibin Mohan

The George Washington University

Challenges in CPS Security



Limited Resources
- Computational power, energy, cost

Timing Requirement
- Safety, reliability, deadlines

System Upgrade
- Verifiability

○ Limitations in Existing Approaches

Resource Overhead

Fine-grained monitoring

(Instructions, memory access, function/system calls)

High Upgrade Cost

Formal verification for every new update

Increasing Complexity

Capabilities and access control

Adaptability

Domain-specific, unforeseen vulnerabilities

Behavior-based Intrusion Detection for CPS

Behavior Monitoring

- Non-intrusive observation of system “behavior”

Protection/Isolation

- Trusted hardware component

System Recovery

- Loss-less control guarantees

Outline

- Background: Real-Time Systems & Simplex
- Cyber-Physical Systems Behavior
- SecureCore: Multicore-based Intrusion Detection
- Control Flow Monitoring
- Anomaly Detection using Kernel-memory Behavior
- Execution Contexts Learned from System Call Distributions
- Current and Future Work
- Conclusion

Background | Real-Time Systems

- A real-time system is defined as,

  “*a system that requires both, logical correctness as well as* ***temporal*** *correctness*”
- Temporal correctness defined as a constraint: **deadline**
- Deadlines determine usefulness of results
  - deadline passes → usefulness drops.

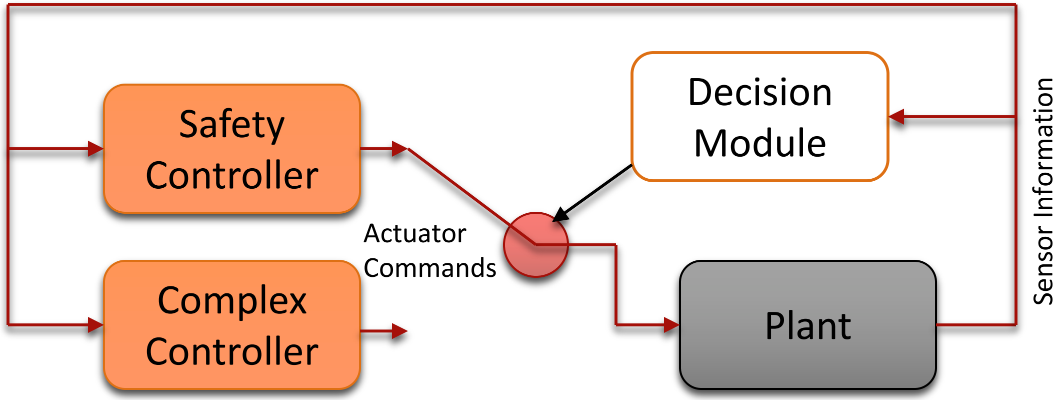
- *E.g.*: Anti-lock Braking System (ABS) in modern automobiles
  - must *function correctly in* ***milliseconds*** *time-frame*
  - even 1 *second might be too late*

  (e.g.: a car traveling at 60 mph has travelled 88 ft. in 1s!)

- Some assumptions in Real-Time Systems:
  - no dynamically loaded code/function pointers/etc.
  - **periodic** programs (“tasks”) that execute **independent** of each other
  - relatively **simple** operating systems
  - limited processing power/memory/network bandwidth/etc.

Background | Simplex Architecture Overview

- Use simplicity to control complexity
- **Simplex** allows use of untrusted, yet high performance/complex subsystem
  - in a safety-critical control system
  - Used successfully in avionics, pacemakers, *etc.*
- High Level-architecture:



Detect problems if complex controller violates safety of plant

- System-level Simplex*: hardware/software partitioning of system

* "The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety" by Bak et al. RTAS 2009.

Outline

- Background: Real-Time Systems & Simplex
- **Cyber-Physical Systems Behavior**
- SecureCore: Multicore-based Intrusion Detection
- Control Flow Monitoring
- Anomaly Detection using Kernel-memory Behavior
- Execution Contexts Learned from System Call Distributions
- Current and Future Work
- Conclusion

Question: can you use the innate properties of real-time systems to detect problems/anomalous behavior?

* **[ICCPS 2010]** “Time-Based Intrusion Detection in Cyber-Physical Systems” by C. Zimmer, B. Bhatt, F. Mueller and S. Mohan.

* **[HiCONS 2013]** “S3A: Secure System Simplex Architecture for Enhanced Security and Robustness of Cyber-Physical Systems” by S. Mohan et al. in HiCONS 2013.

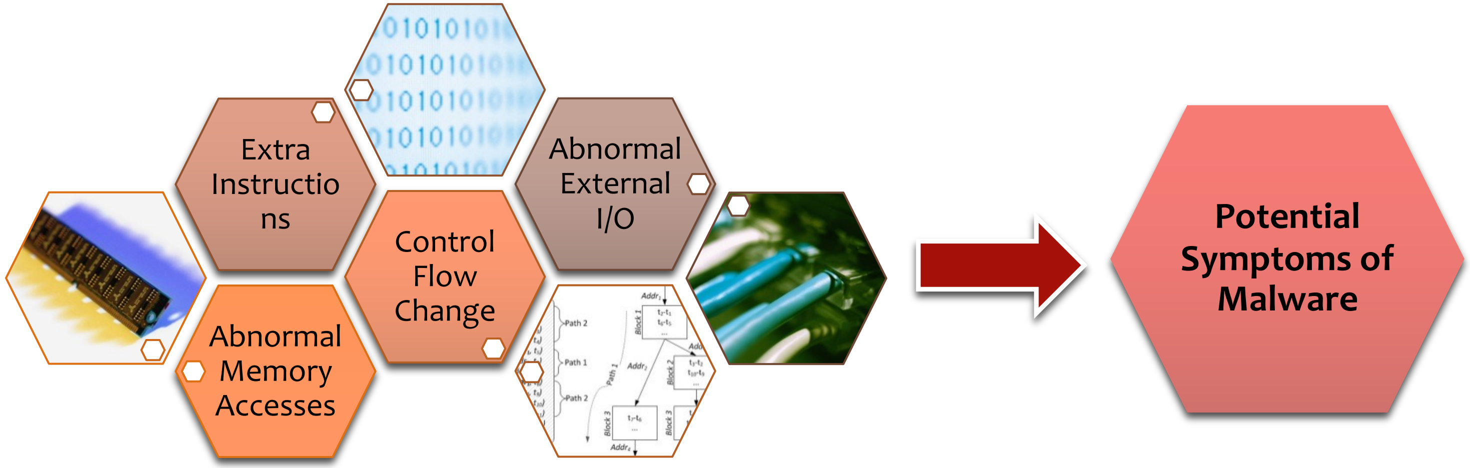
* **[RTAS 2013]** “SecureCore: A Multicore based Intrusion Detection Architecture for Real time Embedded Systems” by M. K. Yoon, S. Mohan, J. Choi, J. E. Kim and L. Sha in RTAS 2013.

* **[CPSNA 2013]** “On-chip control flow integrity check for real time embedded systems” by F. A. T. Abad, J. V. D. Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso and S. Mohan in CPSNA 2013.

* **[DAC 2015]** “Memory Heat Map: Anomaly Detection in Real-Time Embedded Systems using Memory Behavior” by M. K. Yoon, J. Choi, L. Sha and S. Mohan in DAC 2015 [accepted].

CPS Behavior

- CPS are predictable by design
  - Finite set of operational modes, periodic jobs, etc.
- In particular, because of their **real-time** nature their run-time behavior is
  - Fairly *predictable* and *deterministic*
- E.g.: execution time, memory access profile, I/O flows, OS resource usage, power consumption, etc.
- **Deviations from expected behavior** → **suspicious** (more evident than in general purpose systems)

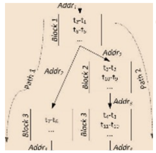


- Combine with **trusted hardware module** to increase robustness

CPS Behavior (contd.)

- Malicious activity consumes resources (e.g. CPU cycles, memory, network, etc.)
- Compromised systems behave differently

- High-level methods to obtain Behavioral Profiles:



Compile-time Analysis

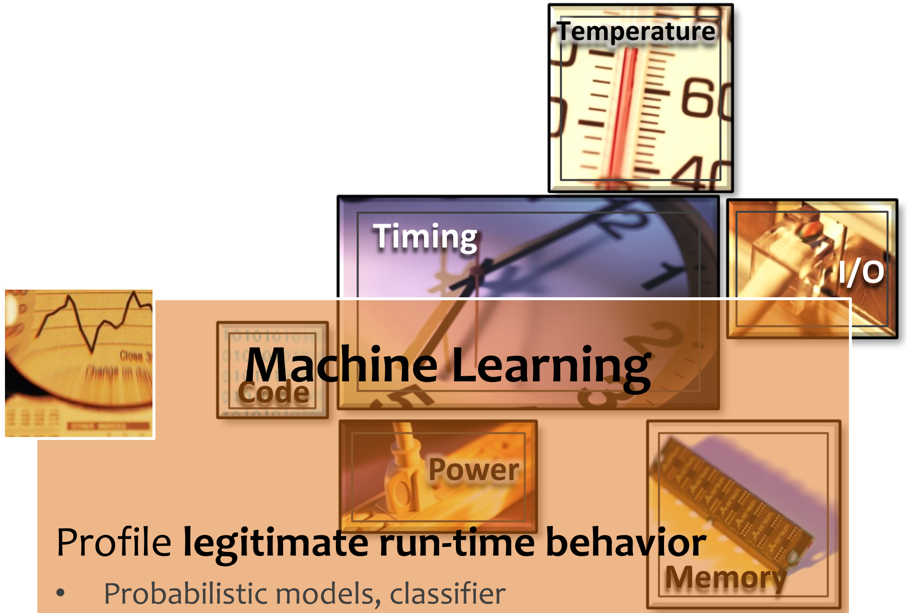
Policy extraction from **source code analysis**

- Exact models, policy checker
- Ex: legitimate control flow of application

Precise

- But, cannot capture behavioral variations

Harder to apply to complex systems



Machine Learning

Profile **legitimate run-time behavior**

- Probabilistic models, classifier
- Ex: legitimate execution time, memory access pattern

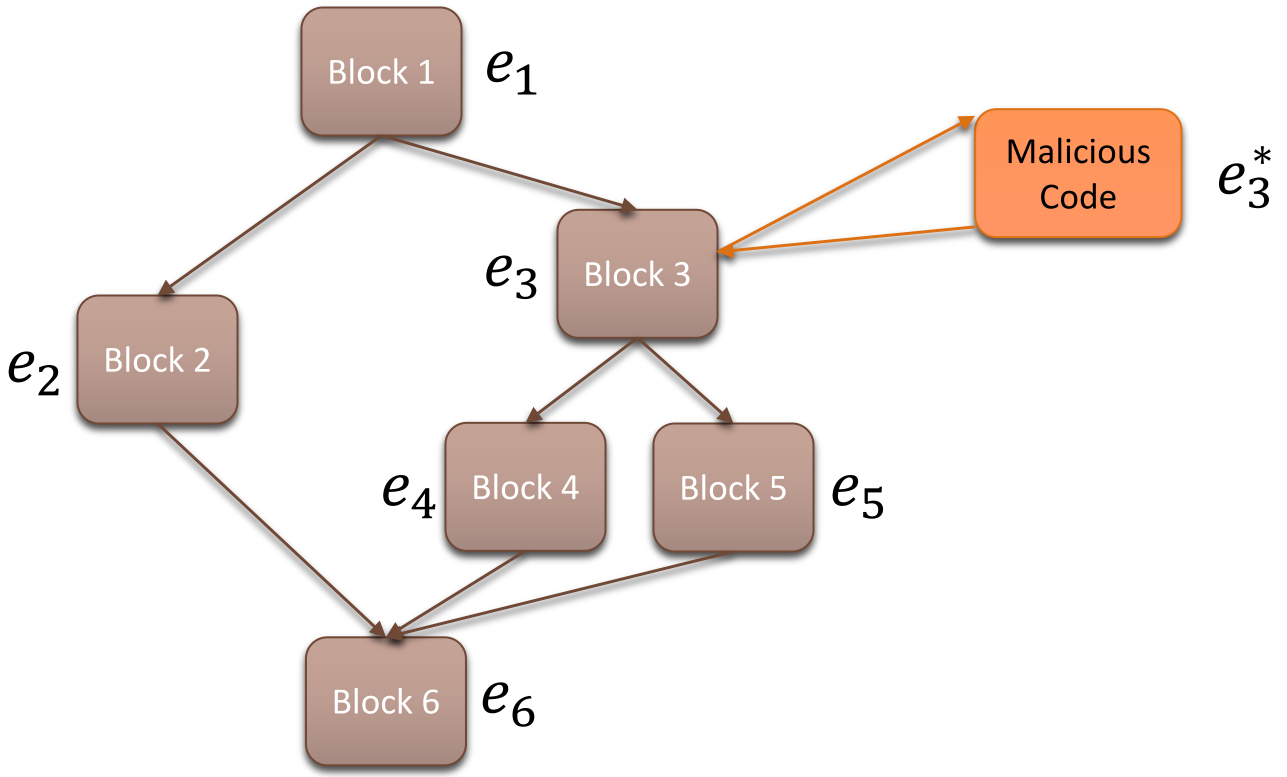
Legitimate variation can also be captured

- System effects, input sets

False alarms can occur

Behavioral Signal | Execution Time

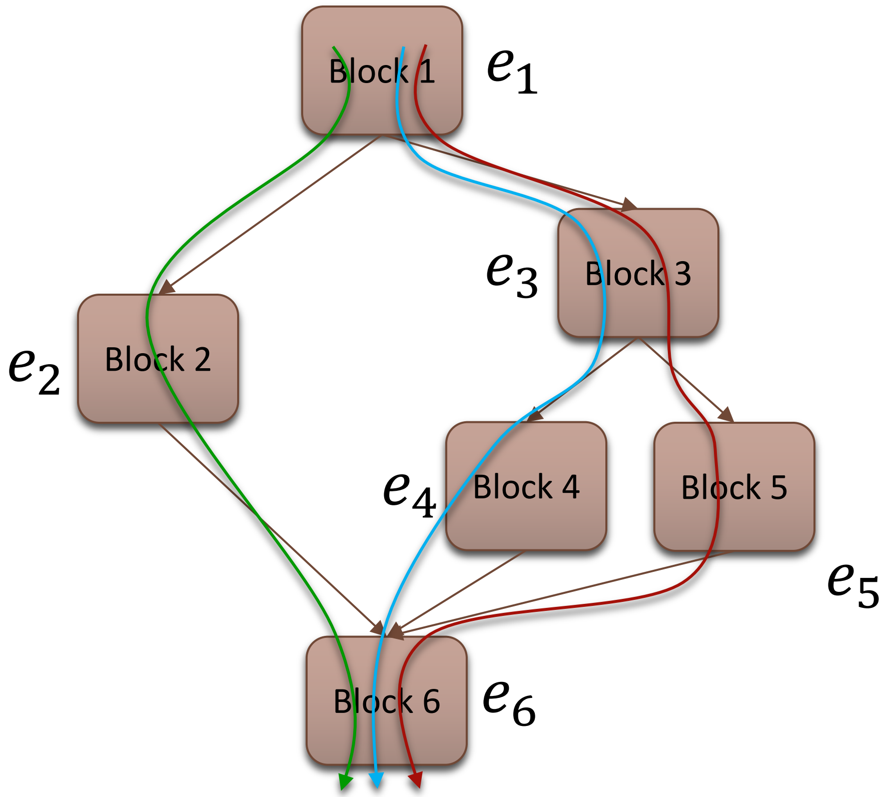
- Say, we are interested in the **deterministic timing profiles** of real-time CPS
- Consider simple control flow:

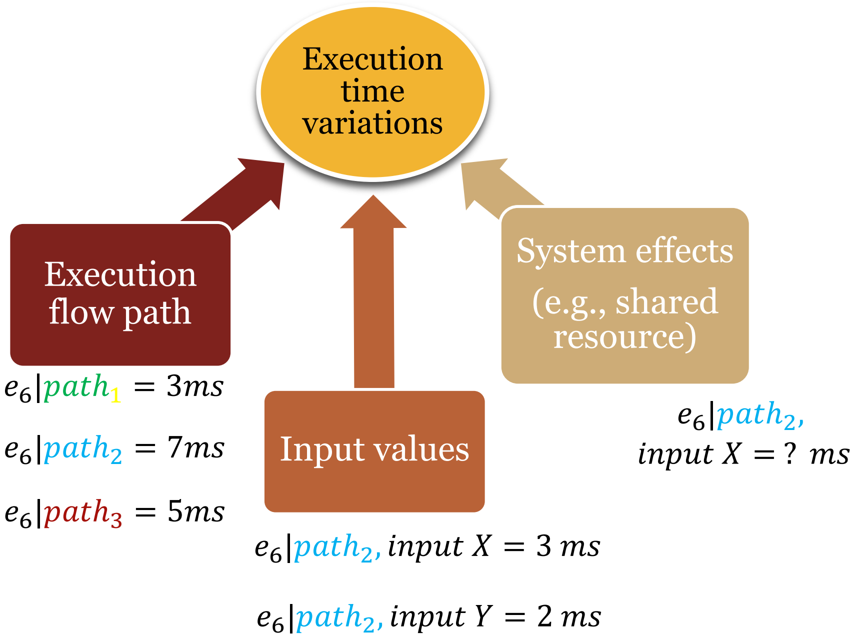


$$e_3^* \neq e_3$$

Behavioral Signal | Variations in Execution Time

- Reasons for variations in CPS Execution Time




Execution time variations
Execution flow path
Input values
System effects (e.g., shared resource)

$e_6|path_1 = 3ms$

$e_6|path_2 = 7ms$

$e_6|path_3 = 5ms$

$e_6|path_2, input\ X = 3\ ms$

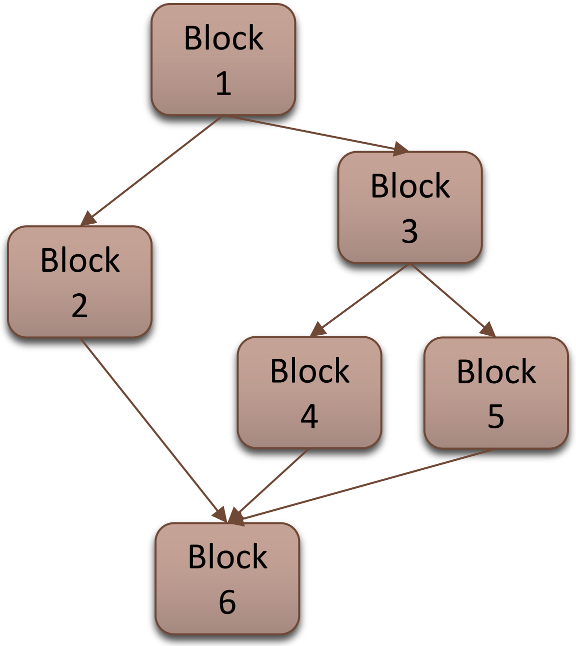
$e_6|path_2, input\ Y = 2\ ms$

$e_6|path_2, input\ X = ?\ ms$

- Will address them all of these types of variations

Timing Profiles

- We profile the application at the **basic block** level
- By narrowing estimation domain (basic block), we get
  - Lesser variations
  - Better accuracy
- Block boundary → check point to detect unexpected flow deviations



Challenges:

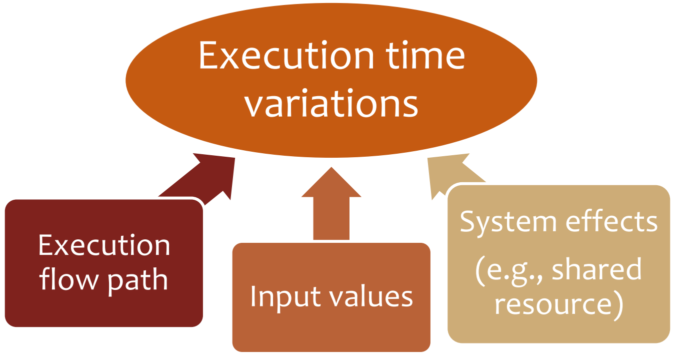
- Execution times can vary for even a single block
  - Due to execution path variations, input sets, system effects, etc.
- *What is a good estimation on execution times?*
  - min, max, variance, mean, etc. → **not representative**; cannot capture variations

Statistical/learning-based Behavioral Profiles

- Obtain behavioral timing profiles for complex code
  - Variability due to control flow, input set variations, etc.

Statistical learning-based profiling/detection

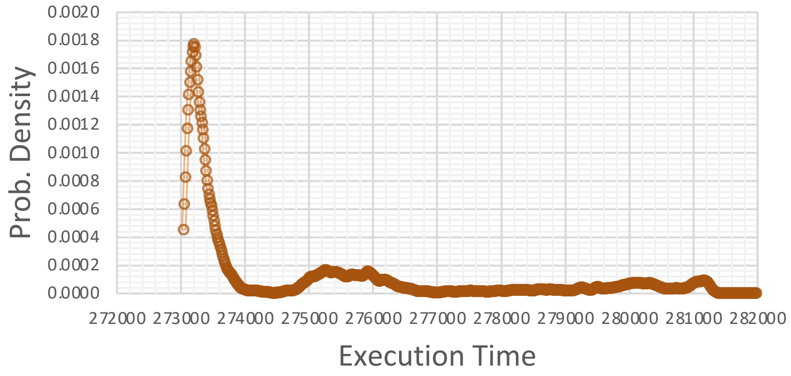
- Profile execution times
  - Even legitimate variations
- Detect abnormal execution time probabilistically


Execution time variations

Execution flow path

Input values

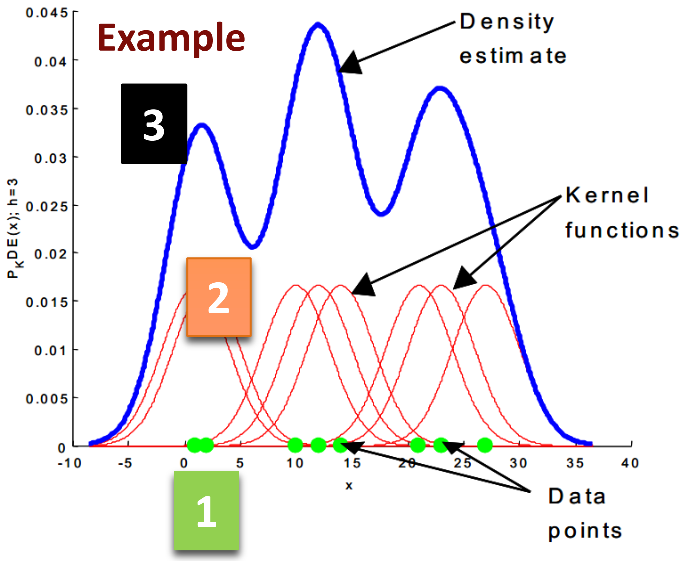
System effects (e.g., shared resource)



- We use **probability density functions** (**PDF**s) for this purpose

Kernel Density Estimation (KDE)

- Non-parametric Probability Density Function Estimation

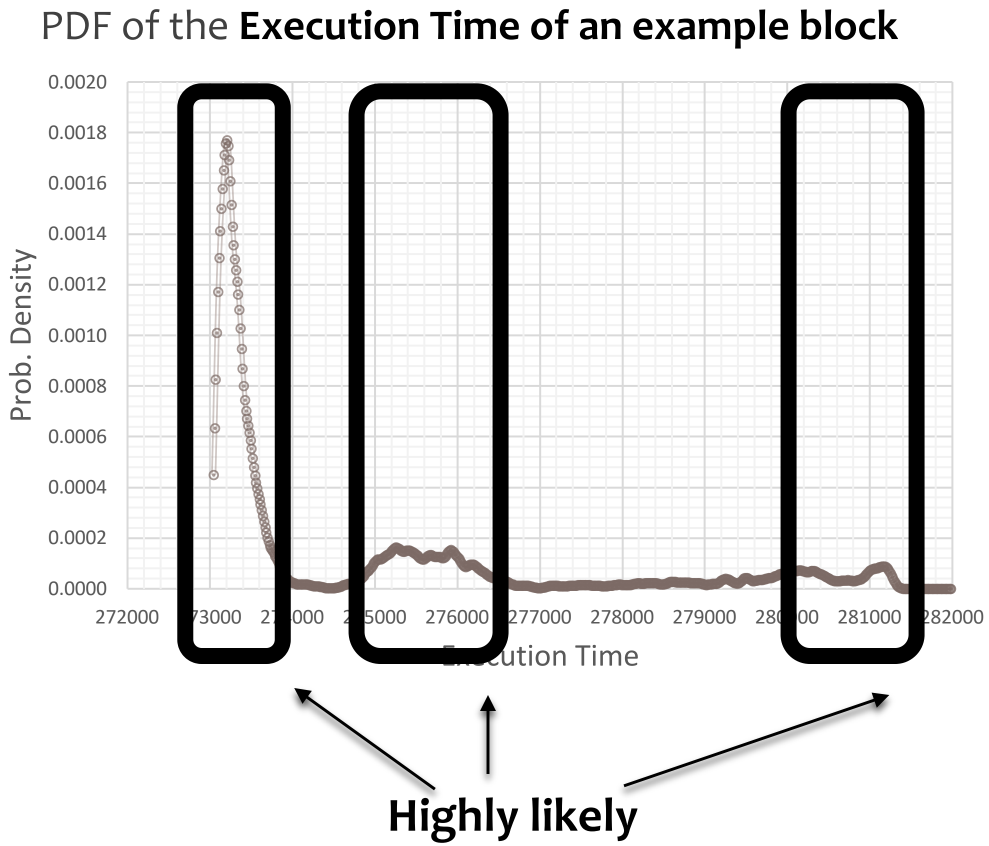


[Figure is from CSCE 666 “Pattern Analysis” by Ricardo Gutierrez-Osuna at Texas A&M]

1. Given samples of exec times
2. Draw scaled distribution at each sample point
3. Sum them up

$$\underbrace{\hat{f}_h(e|e^{(1)},\ldots,e^{(m)})}_{\textbf{Estimated pdf}} = \frac{1}{m}\sum_{i=1}^{m} \overset{\textbf{Kernel}\text{ function}}{K}_{\underset{\substack{\text{Bandwidth}\\\text{(Smoothing constant)}}}{h}}(e-e^{(i)})$$

Intrusion Detection Using PDFs



How much **deviation** should we consider malicious?

Threshold test

$Prob(e^*) < \theta \longrightarrow$ **Malicious**

$Prob(e^*) \geq \theta \longrightarrow$ **Legitimate**

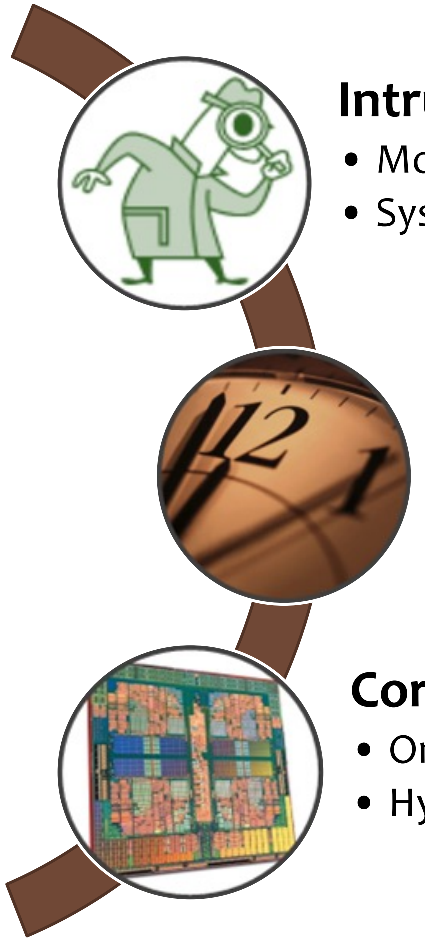
Multiple regions: different inputs or persistent system effects

Outline

- Background: Real-Time Systems & Simplex
- Cyber-Physical Systems Behavior
- **SecureCore: Multicore-based Intrusion Detection**
- Control Flow Monitoring
- Anomaly Detection using Kernel-memory Behavior
- Execution Contexts Learned from System Call Distributions
- Current and Future Work
- Conclusion

SecureCore

- ○ Can use **redundancy in multiple cores** to improve security of CPS
- ○ Multicore-based **on-chip hardware** for monitoring behavior of tasks
- ○ Directly obtain information from processor → don’t rely on monitored task
- ○ Analyze more complex control flows and behavioral variations
- ○ Uses **statistical/machine learning approach** to creating behavior profiles



Intrusion detection, not prevention

- Monitor the most critical components
- System recovery upon detection

Behavior monitoring

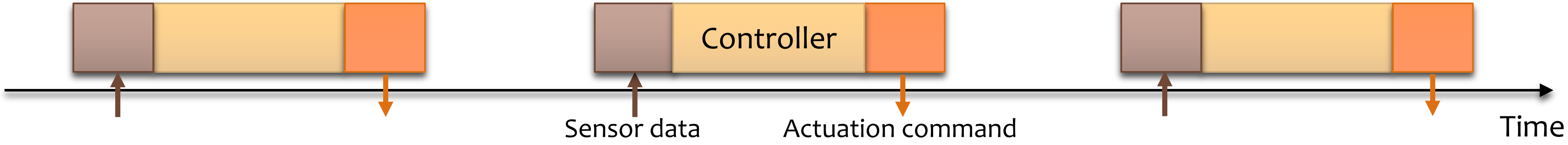
- Predictable behaviors of real-time apps
- Profile behavior pattern by machine learning

Core-to-core monitoring on multicore

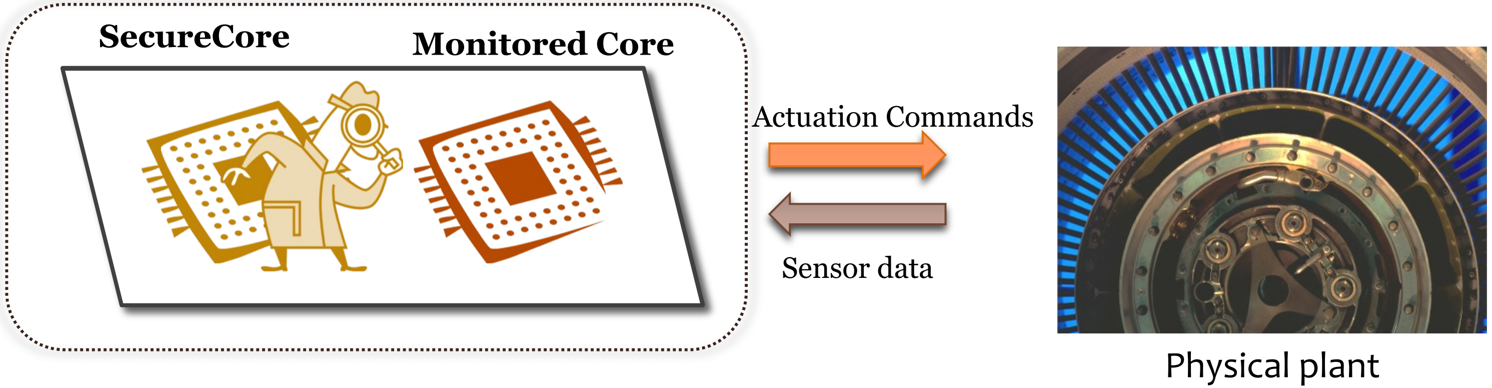
- On-chip HW for OS/APP state inspections
- Hypervisor-/HW-based core protection/isolation

System/App Model

- Periodic Controller task with real-time requirements



- A multicore-based real-time control system



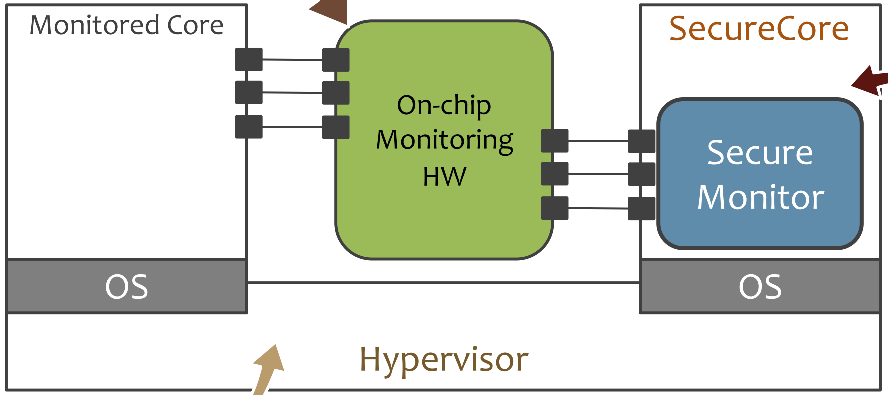
SecureCore Architecture

SecureCore High-Level Design

- A dedicated core for inspecting behavior of other core(s)

On-chip Monitoring HW Unit

- Observes the state of monitored cores, I/O activities, physical states, etc.
- Invisible to all but SecureCore, non-intrusive



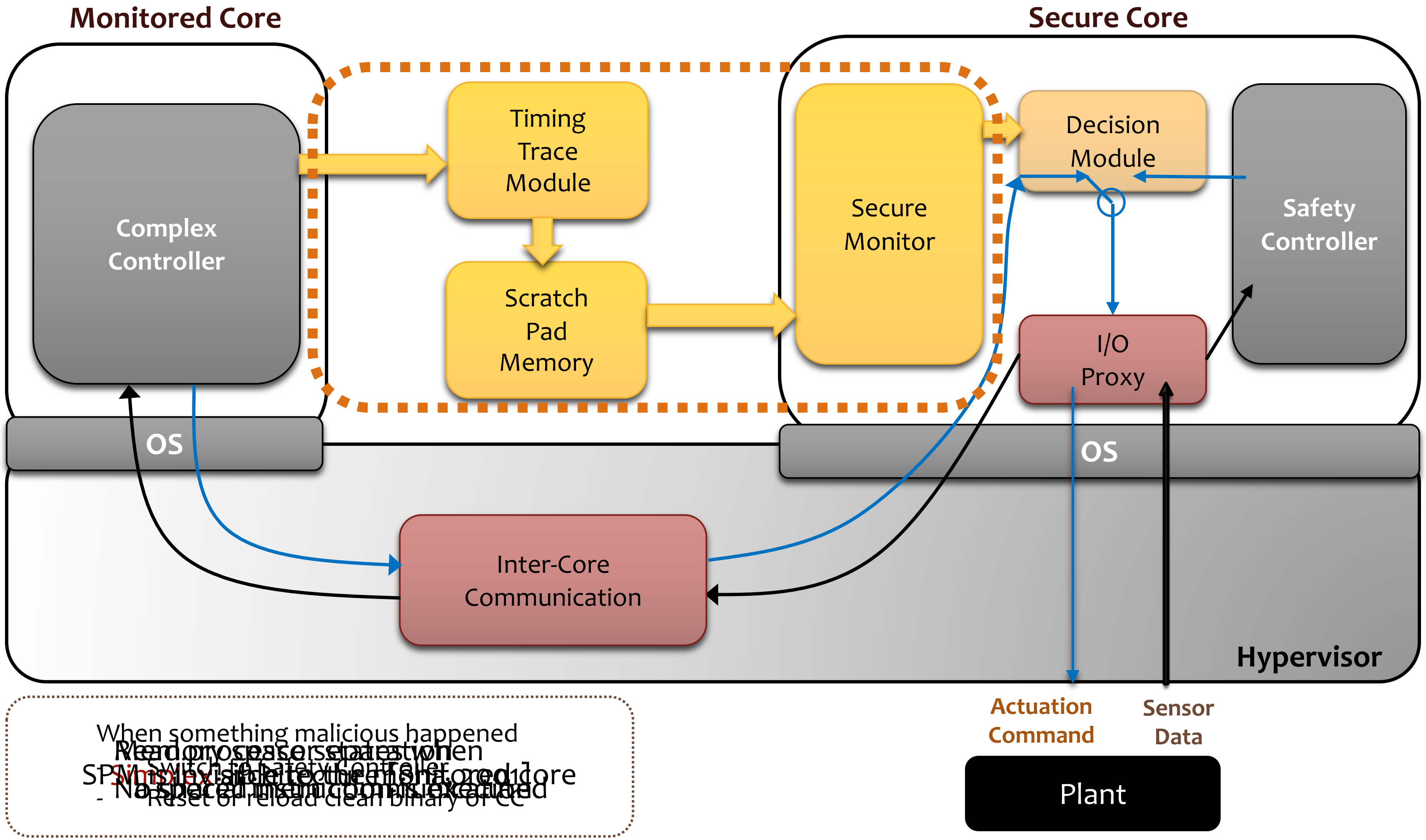
Secure Monitor

- Software process that performs monitoring and detection using observed behavior

Hypervisor-based SecureCore Protection

- Resource virtualization: memory space separation, I/O device consolidation
- Additional HW-based protection (e.g., ARM TrustZone)

SecureCore Architecture Low-level Design



When something malicious happened

- Switch to Safety Controller
- Reset or reload clean binary of CC

Read processor state when

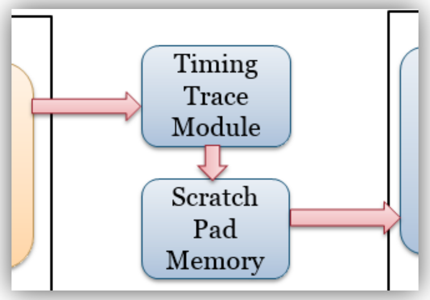
No special instructions executed

Memory space separation

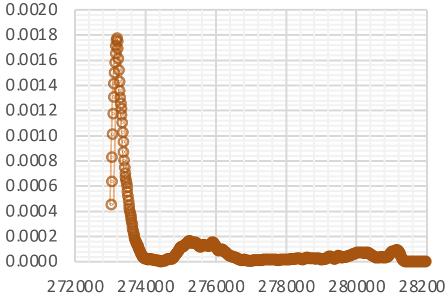
SPM is visible to the monitored core

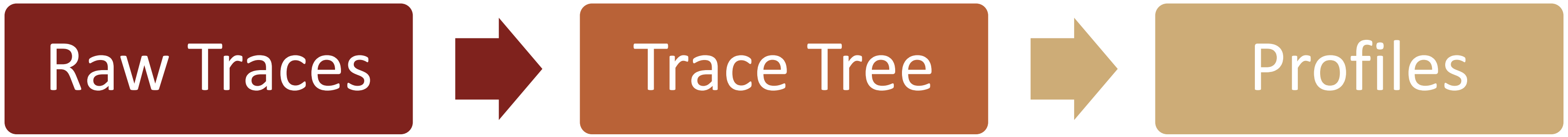
Simplex

How to Obtain the Timing Profiles



Statistical Learning



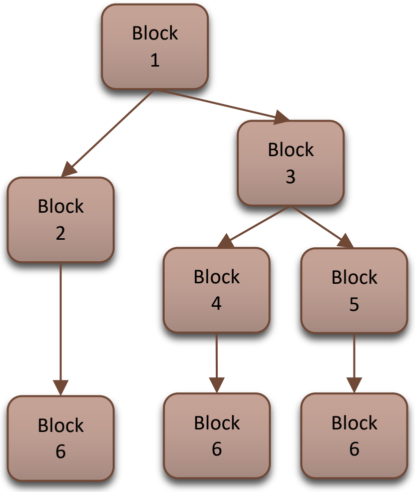


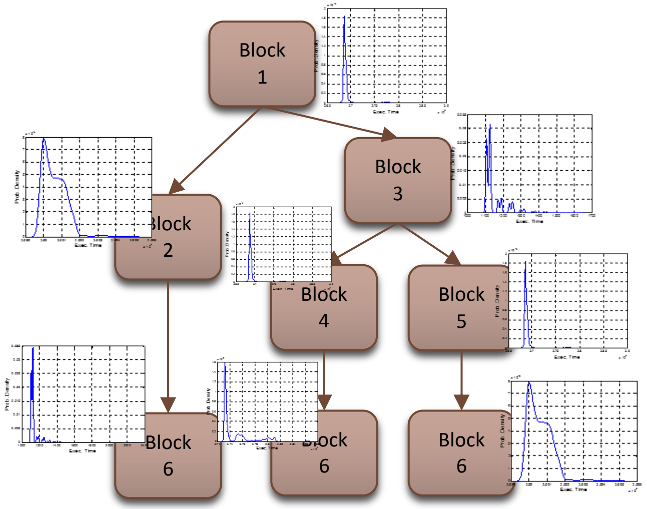
Raw Traces

Trace Tree

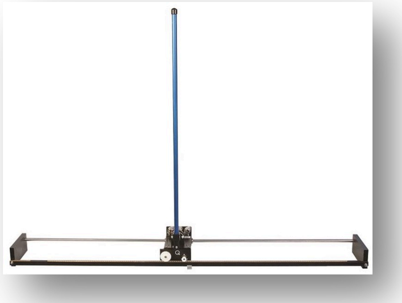
Profiles

$(Addr_1, t_1)$
$(Addr_2, t_2)$
$(Addr_3, t_3)$
$(Addr_4, t_4)$
$(Addr_1, t_5)$
$(Addr_2, t_6)$
$(Addr_5, t_7)$
$(Addr_6, t_8)$
$(Addr_7, t_9)$
$(Addr_1, t_{10})$
$(Addr_2, t_{11})$
…

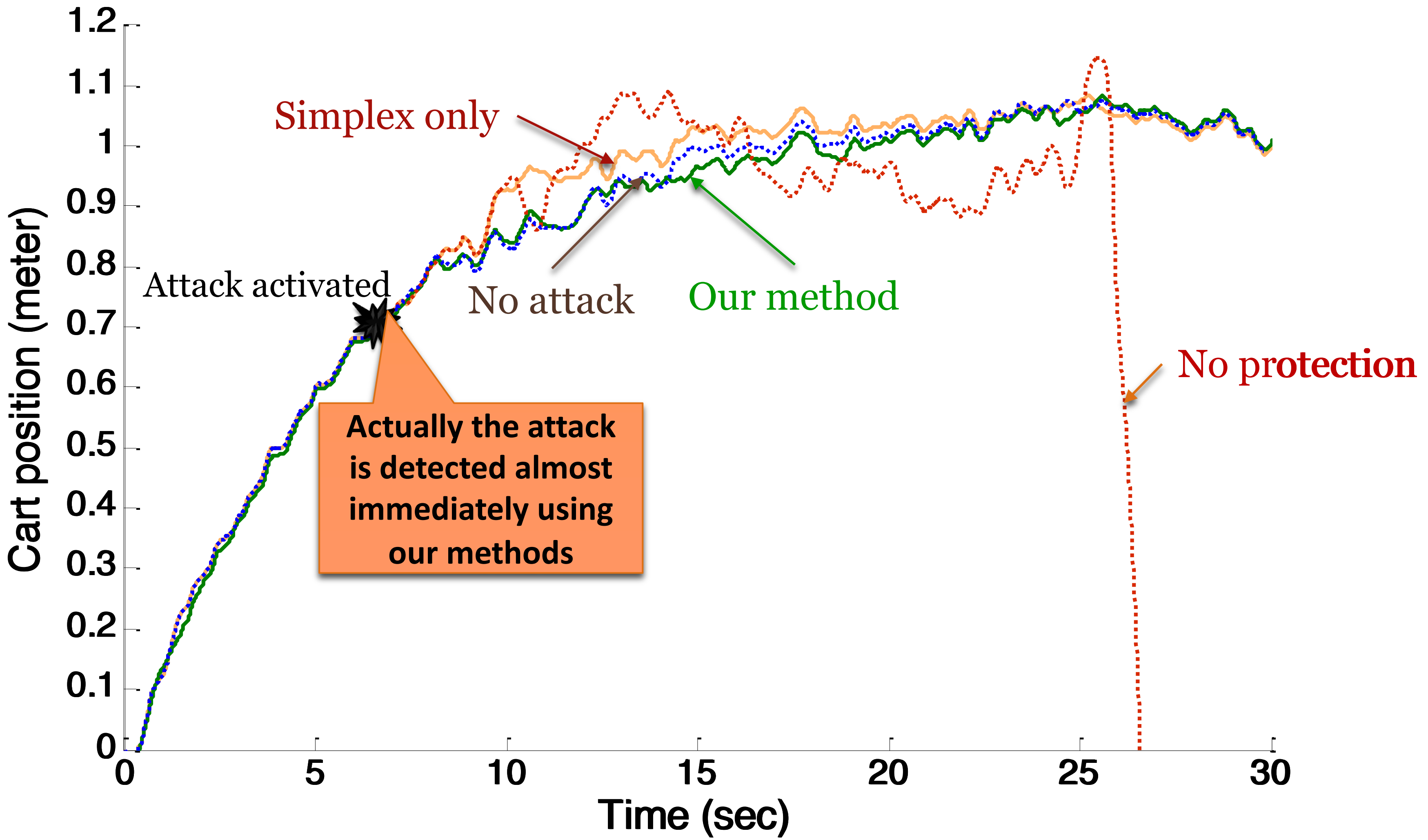




Early Detection for Inverted Pendulum (IP)



Inverted Pendulum



Actually the attack is detected almost immediately using our methods

Outline

- Background: Real-Time Systems & Simplex
- Cyber-Physical Systems Behavior
- SecureCore: Multicore-based Intrusion Detection
- **Control Flow Monitoring**
- Anomaly Detection using Kernel-memory Behavior
- Execution Contexts Learned from System Call Distributions
- Current and Future Work
- Conclusion

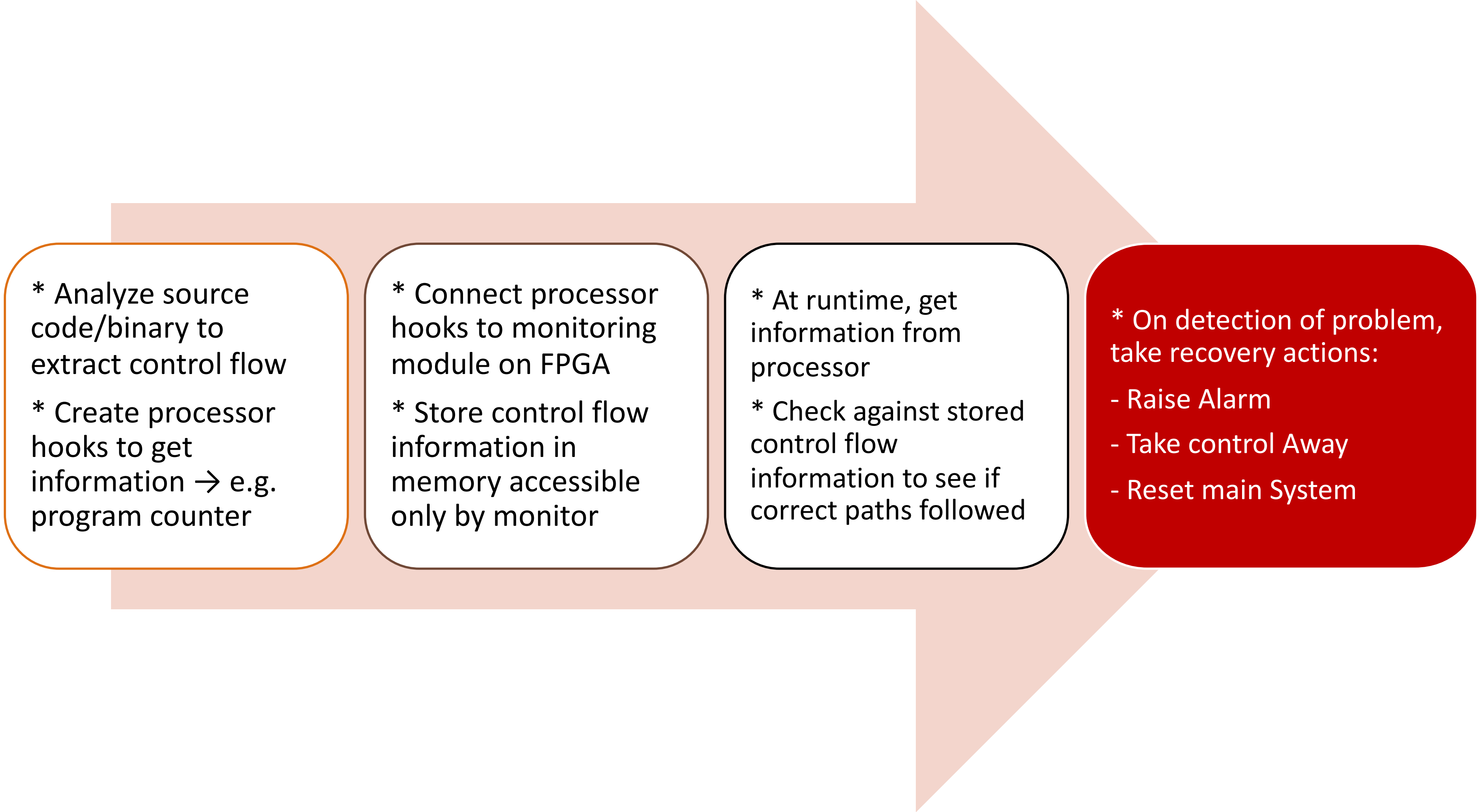
CFM: Control Flow Monitor

- Timing is just one of the behavioral ‘*signals*’ that can be monitored
  - Smart adversary can insert code that matches timing behavior closely

- The **behavior** of the **control flow** in real-time systems is deterministic
  - Not just for the real-time tasks, but also for OS components like scheduler

- **CFM**
  - Profile the control flow for components on the main processor
  - Track the flow of control at runtime
  - **Tracking module implemented as simple monitoring module on FPGA logic**
  - Get critical information directly from processor hooks

CFM Sequence of Events



* Analyze source code/binary to extract control flow

* Create processor hooks to get information → e.g. program counter

* Connect processor hooks to monitoring module on FPGA

* Store control flow information in memory accessible only by monitor

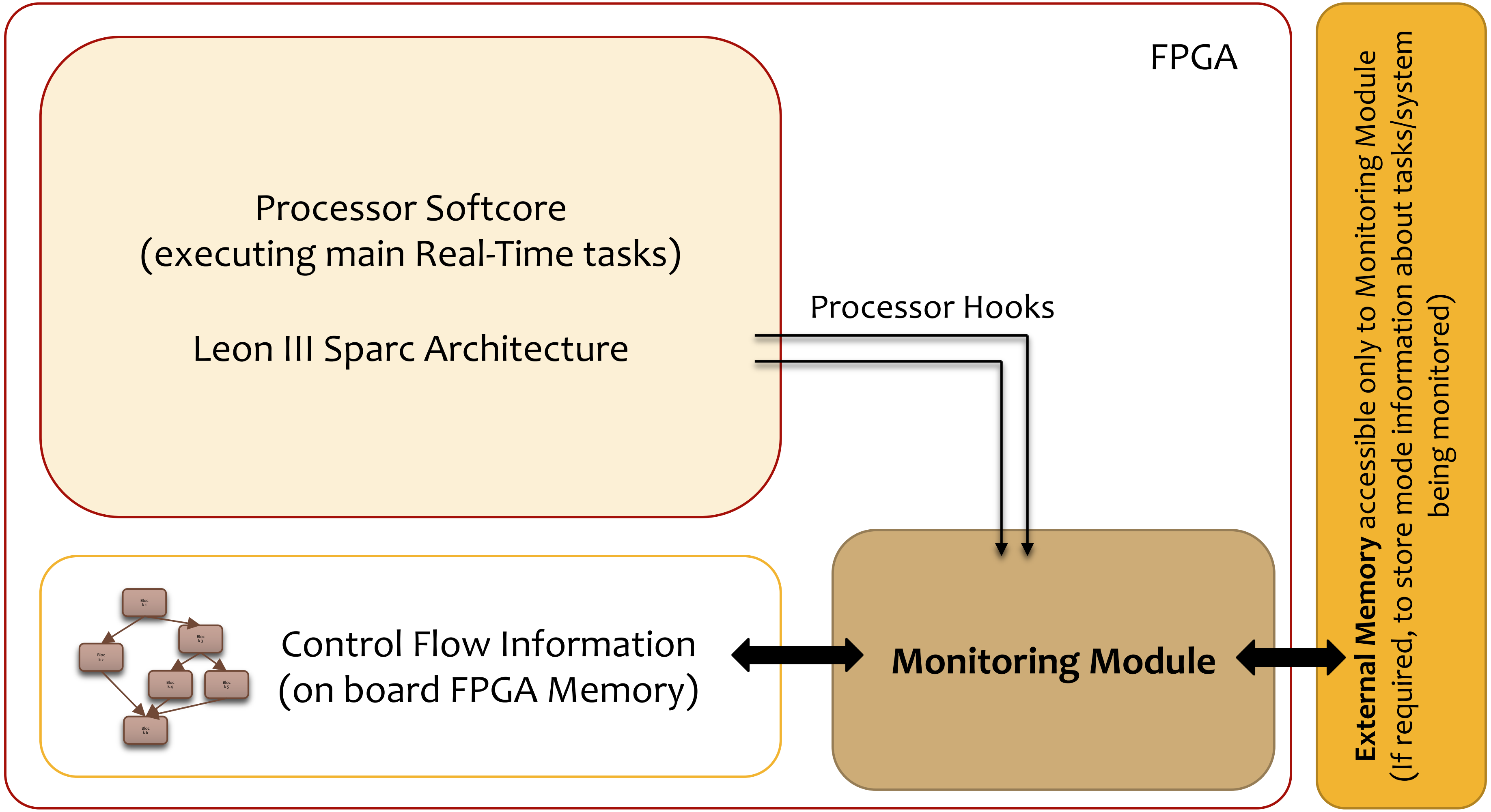
* At runtime, get information from processor

* Check against stored control flow information to see if correct paths followed

* On detection of problem, take recovery actions:

- Raise Alarm
- Take control Away
- Reset main System

CFM High Level Architecture



FPGA

Processor Softcore
(executing main Real-Time tasks)

Leon III Sparc Architecture

Processor Hooks

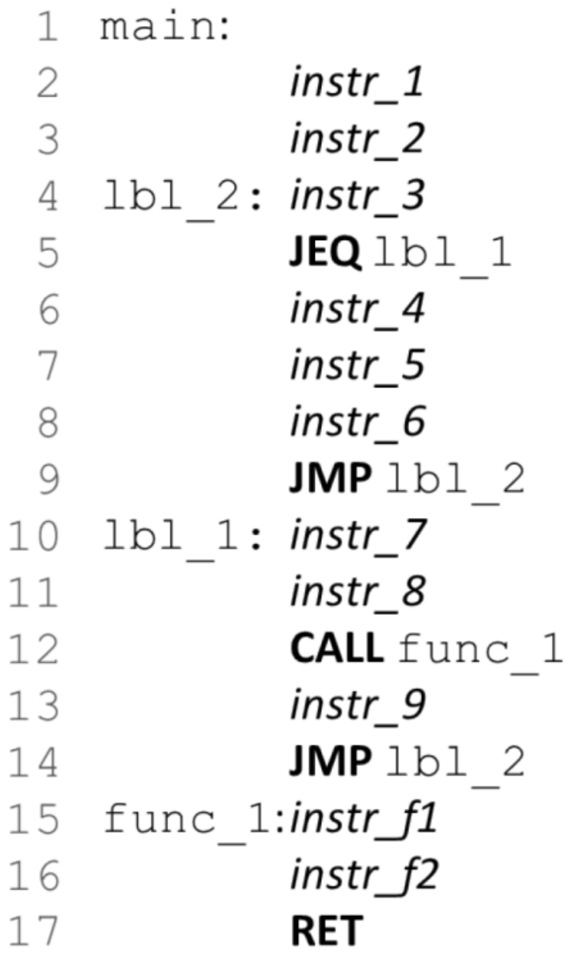
Control Flow Information
(on board FPGA Memory)

Monitoring Module

External Memory accessible only to Monitoring Module
(If required, to store mode information about tasks/system being monitored)

CFM Implementation

- Automated tool to extract control flow information from single task binary
- Example



```
main:
          instr_1
          instr_2
lbl_2:    instr_3
          JEQ lbl_1
          instr_4
          instr_5
          instr_6
          JMP lbl_2
lbl_1:    instr_7
          instr_8
          CALL func_1
          instr_9
          JMP lbl_2
func_1:   instr_f1
          instr_f2
          RET
```

Implementation (contd.) & Evaluation

- Implemented using a Leon3 softcore processor on Xilinx Virtex-5 FPGA
- Remaining fabric on FPGA → monitoring module with hooks into Leon3 pipeline
  - Program counter (PC) & Instruction Register (IR)
- Application: code for PID controller for temperature control in an industrial unit
  - Generated 240 separate execution blocks in the CFG

- Attacks:
  1. Code replacement attack by loading a modified binary → different jump destination for one block
  2. Return address overwritten on the stack using buffer overflows

- Both attacks detected almost immediately – i.e. within a few instructions (before next block executes)

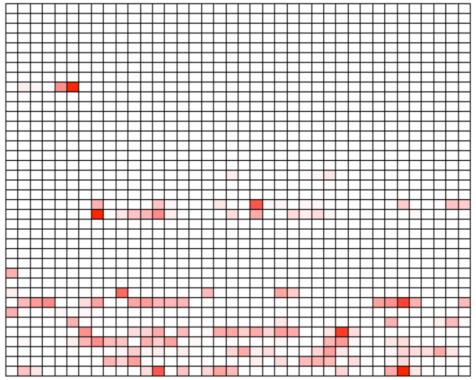
Outline

- Background: Real-Time Systems & Simplex
- Cyber-Physical Systems Behavior
- SecureCore: Multicore-based Intrusion Detection
- Control Flow Monitoring
- **Anomaly Detection using Kernel-memory Behavior**
- Execution Contexts Learned from System Call Distributions
- Current and Future Work
- Conclusion

Tracking Memory Behavior

- Multiple ways to track memory
  - Exact sequence of memory addresses → too much overhead [storage/computation]
  - Monitor the amount of memory traffic [e.g. bandwidth] → abstracts away details

- We introduce the **memory heat map** (**MHM**)
  - *Composition of different activities in a certain memory region*
  - Provides necessary details
  - Concise data structure



- We use this to profile memory behavior for the operating system **kernel**

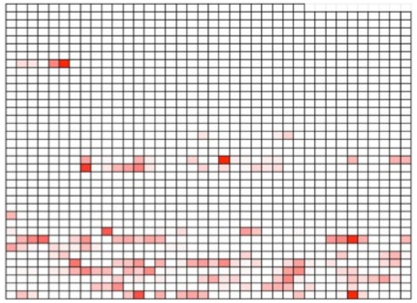
Why Kernel Memory Behavior

- Good indicator of system-wide behavior
  - Every application has to use kernel services
- Can also detect certain anomalies
  - e.g. unexpected start/end of applications or
  - Suspicious use of kernel services
- Simpler H/W design
  - Kernel memory location is **fixed** and **known**
  - No need to deal with address translation and paging

- Monitor kernel **instructions [.text]** section
  - Inspect which parts of kernel have executed

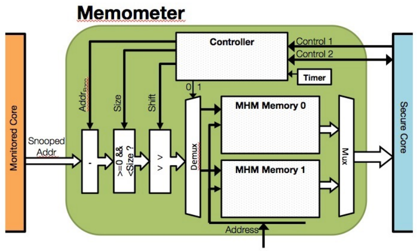
Key Approaches

- Memory Heat Map (MHM)
  - # of accesses to memory regions during a time interval
  - Depends only on the size of the monitored region

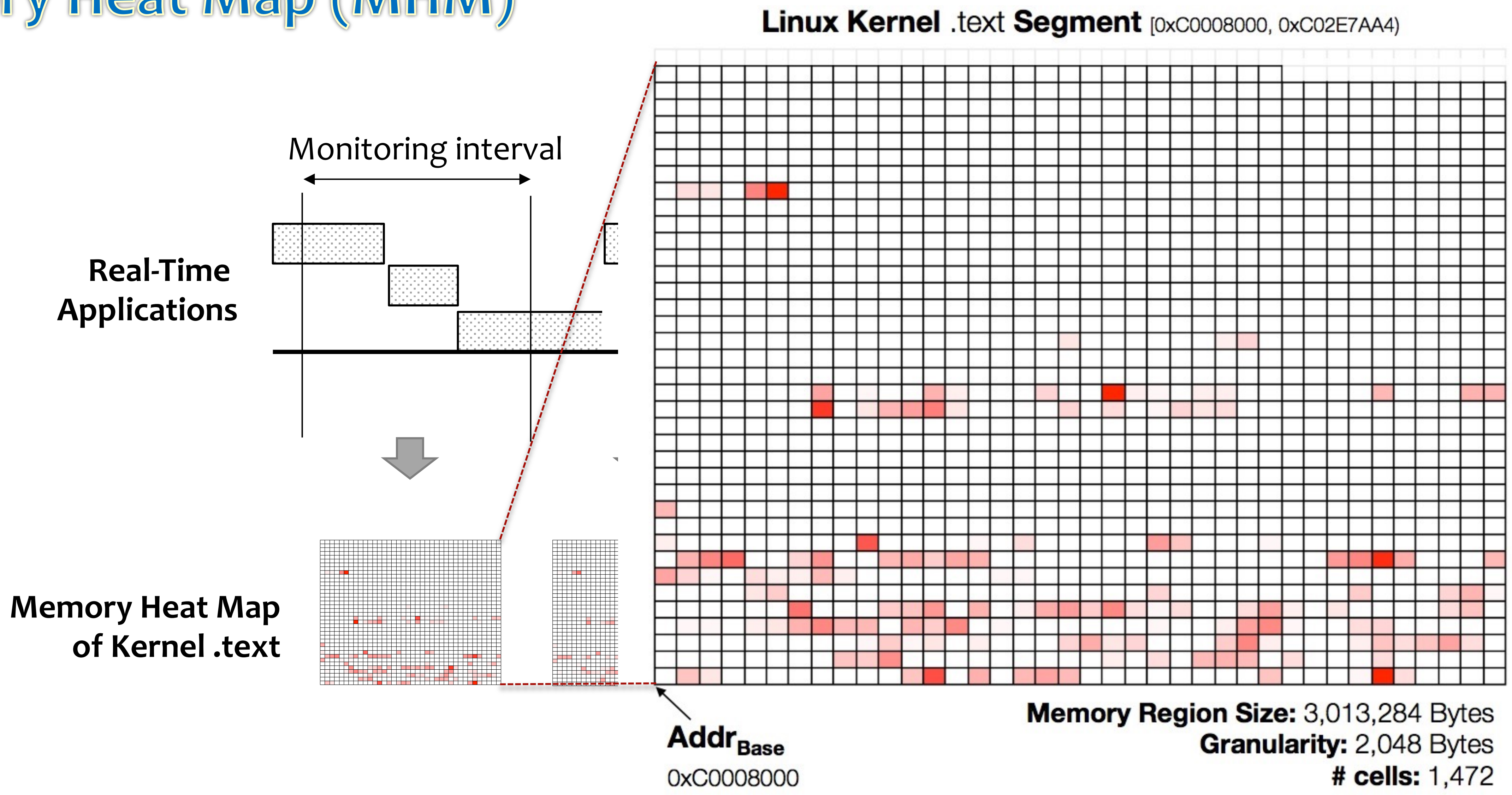


- Image recognition technique
  - Dimensionality reduction
  - Normal behavior learning and anomalous behavior detection

- On-chip SecureCore-based hardware module (**Memometer**)
  - Real-time memory access monitoring

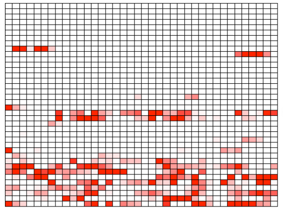


Memory Heat Map (MHM)


Linux Kernel .text Segment [0xC0008000, 0xC02E7AA4)
Monitoring interval
Real-Time Applications
Memory Heat Map of Kernel .text
Addr$_{Base}$
0xC0008000
Memory Region Size: 3,013,284 Bytes
Granularity: 2,048 Bytes
# cells: 1,472

Learning from Memory Heat Maps

- Goal
  - 1) Find the **legitimate behavior patterns** from the normal MHMs
  - 2) Given a new observation (MHM), analyze the **statistical similarity** to the patterns



Is this normal or not?

- Idea & Intuition
  - Treat each MHM as an **image**
    - *Normal memory behavior can be grouped into a finite number of similar image groups*
  - Then, use an *image recognition technique* and *clustering*

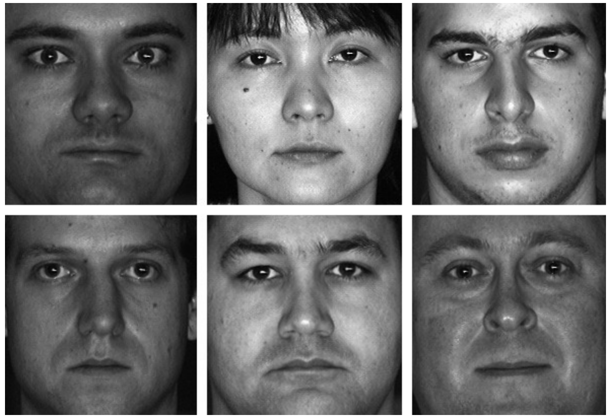
Eigenface | Dimensionality Reduction

- An image recognition technique
- Based on **PCA** (Principal Component Analysis)
  - Transform data to a low-dimensional coordinate system
  - They best describe the distribution of original data
    - The first principal component has the largest variance and so on

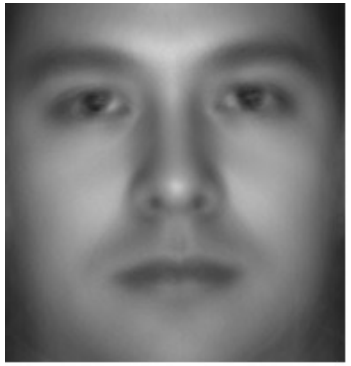
- Eigenface = a basic image
  - Learn (extract) a set of Eigenfaces from the original images using PCA
  - # of eigenfaces << # of original images
  - They can be linearly composed to reconstruct the original images with a minimal approximation error

Eigenface

- Face recognition technique



Original faces



Average face

Eigenfaces

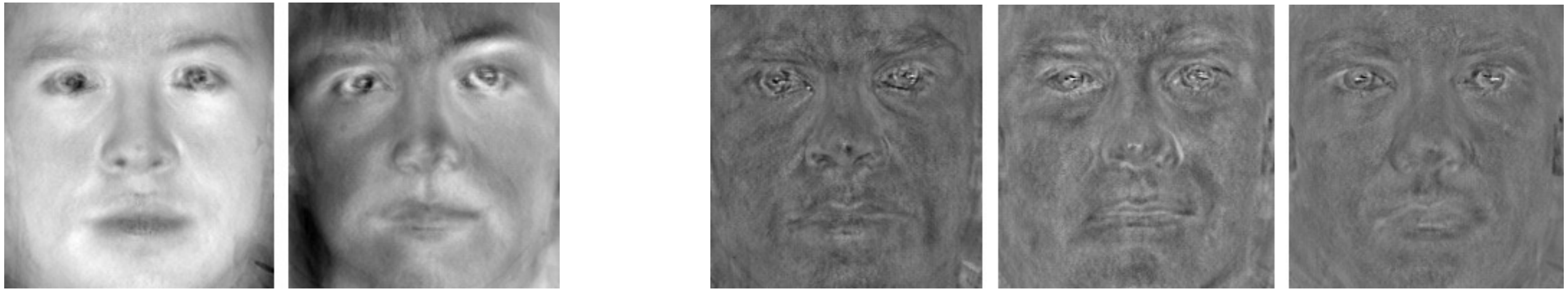
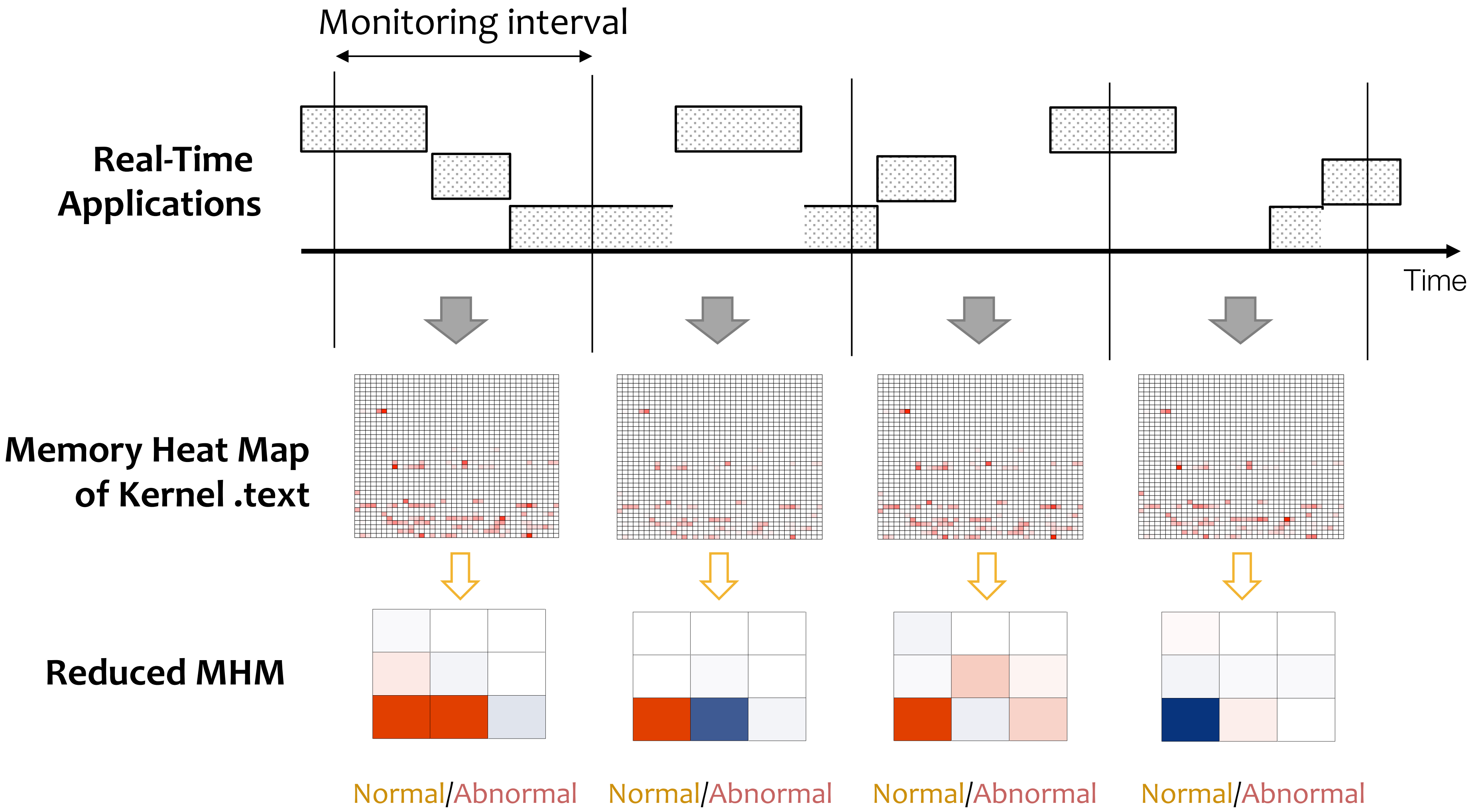
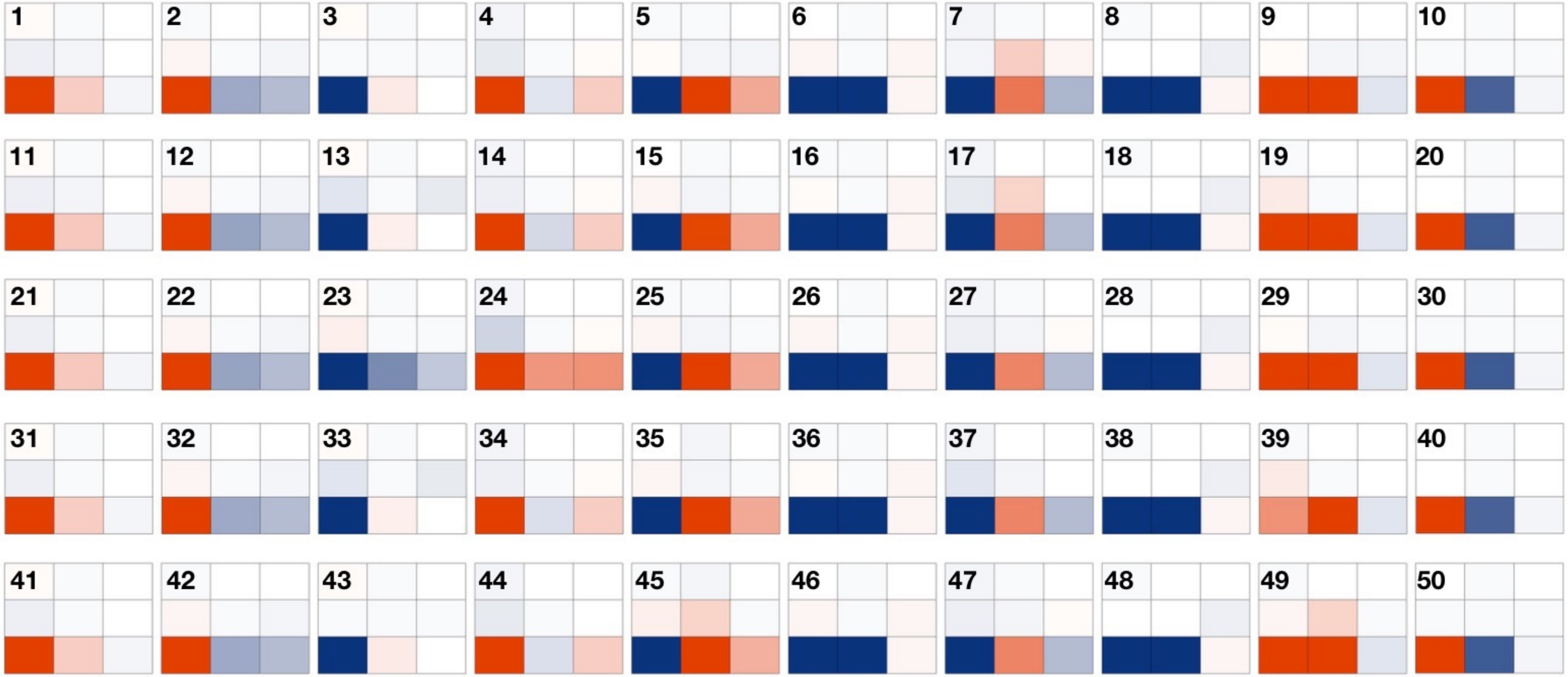


Image source: http://www.scholarpedia.org/article/Eigenfaces

Dimensionality Reduction for MHMs



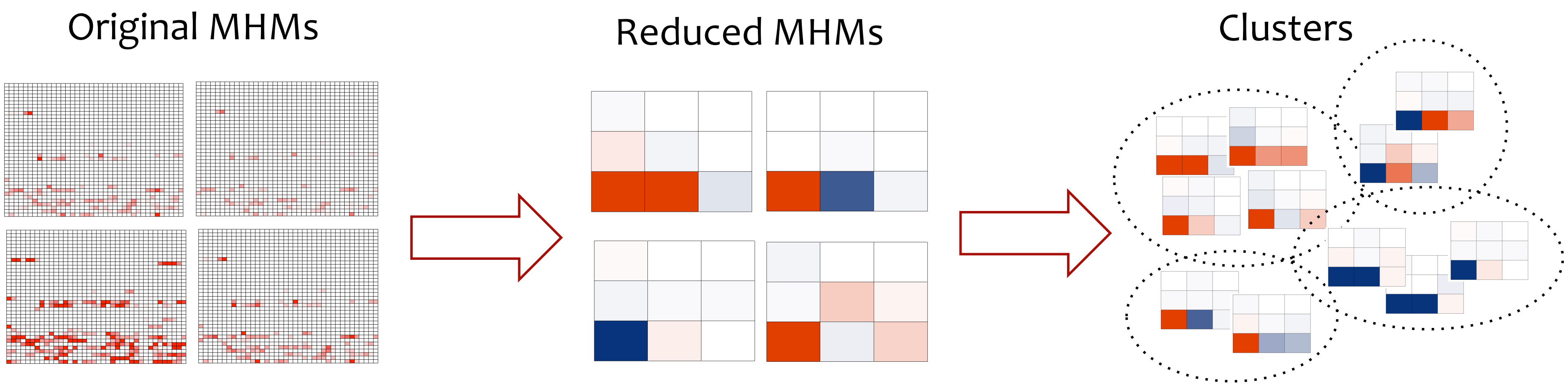
Examples of MHMs captured as a Sequence



Note: One MHM is captured every 10 ms

MHM Learning & Anomaly Detection

- In our memory analysis domain:



Original MHMs

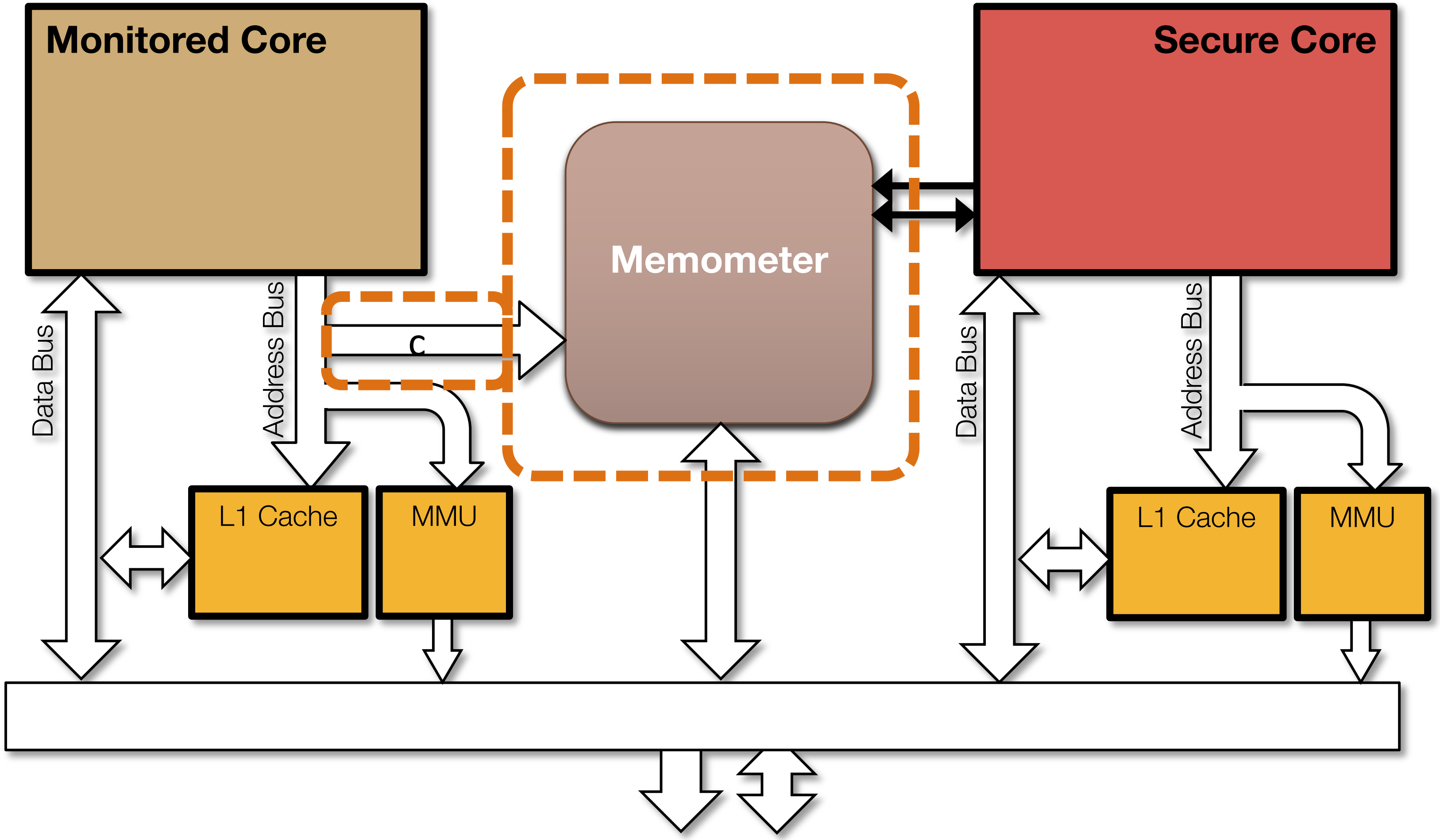
Reduced MHMs

Clusters

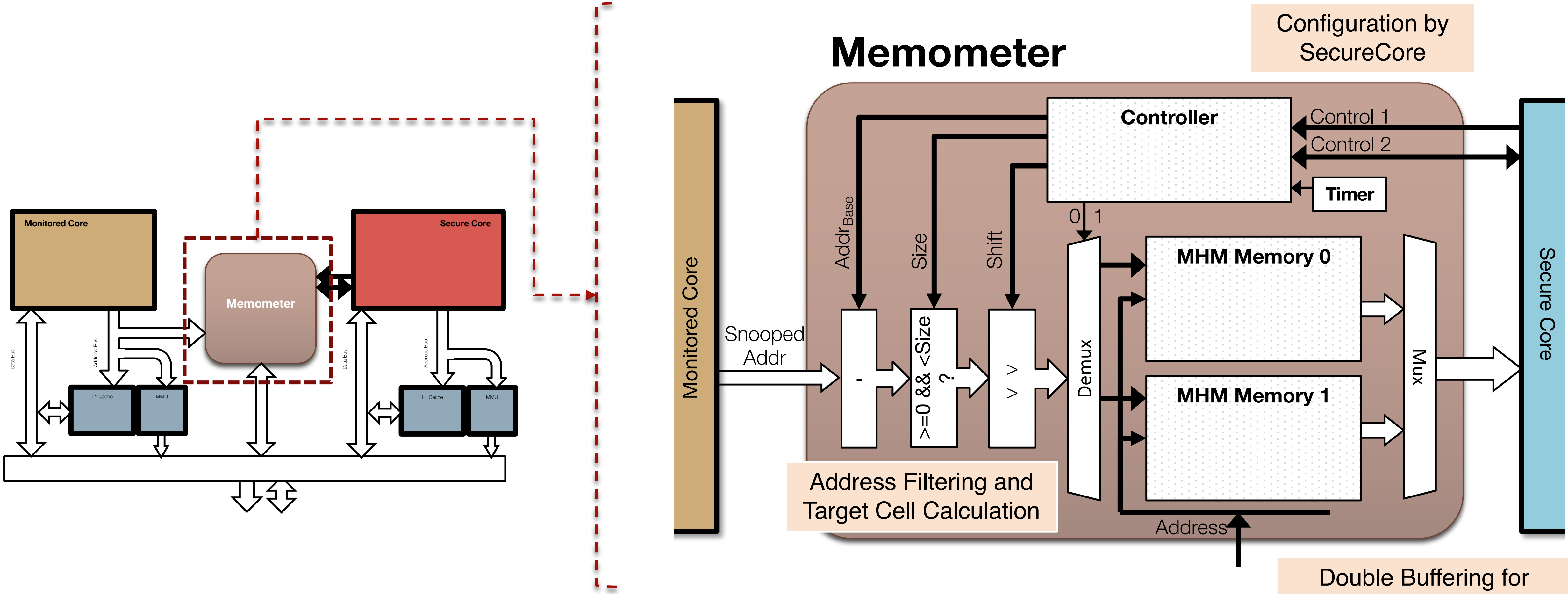
Pattern learning using clustering

- *E.g.* Gaussian Mixture Model (GMM)
- Identify representative MHM patterns

SecureCore Architecture for Memory Monitoring



Memometer



Configuration by SecureCore

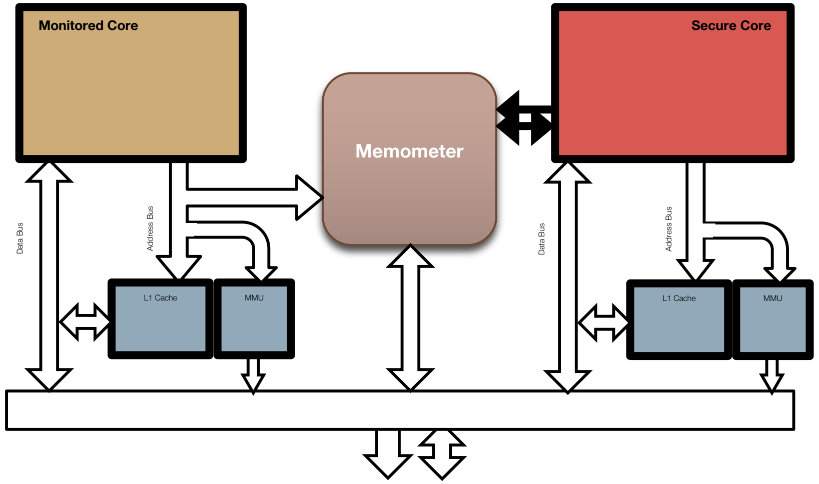
Address Filtering and Target Cell Calculation

Double Buffering for Uninterrupted Monitoring

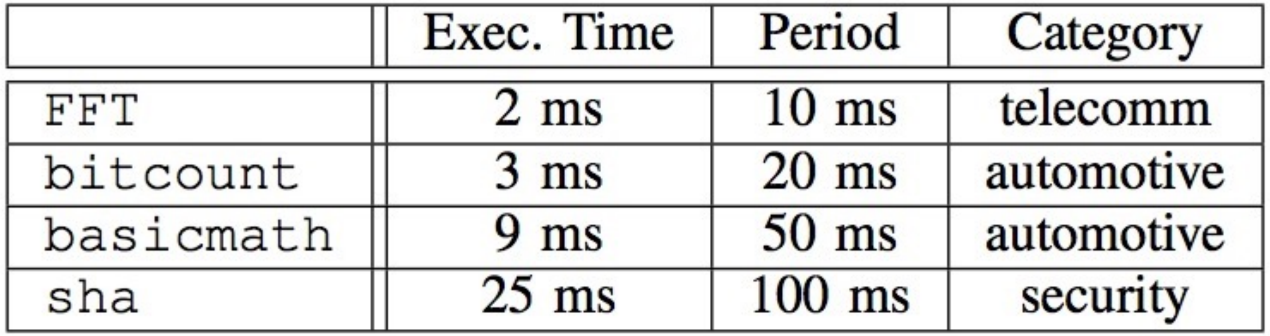
Implementation & Evaluation

Prototype implementation

- ARM Cortex-A9 on Simics
- Linux 3.4 → .text segment is monitored
- 10ms interval, 2KB cell size
- Embedded benchmarks



Embedded benchmarks



	Exec. Time	Period	Category
FFT	2 ms	10 ms	telecomm
bitcount	3 ms	20 ms	automotive
basicmath	9 ms	50 ms	automotive
sha	25 ms	100 ms	security

Anomalies/attacks

- Unknown application launch
- Application kill
- Shellcode execution
- Kernel rootkit

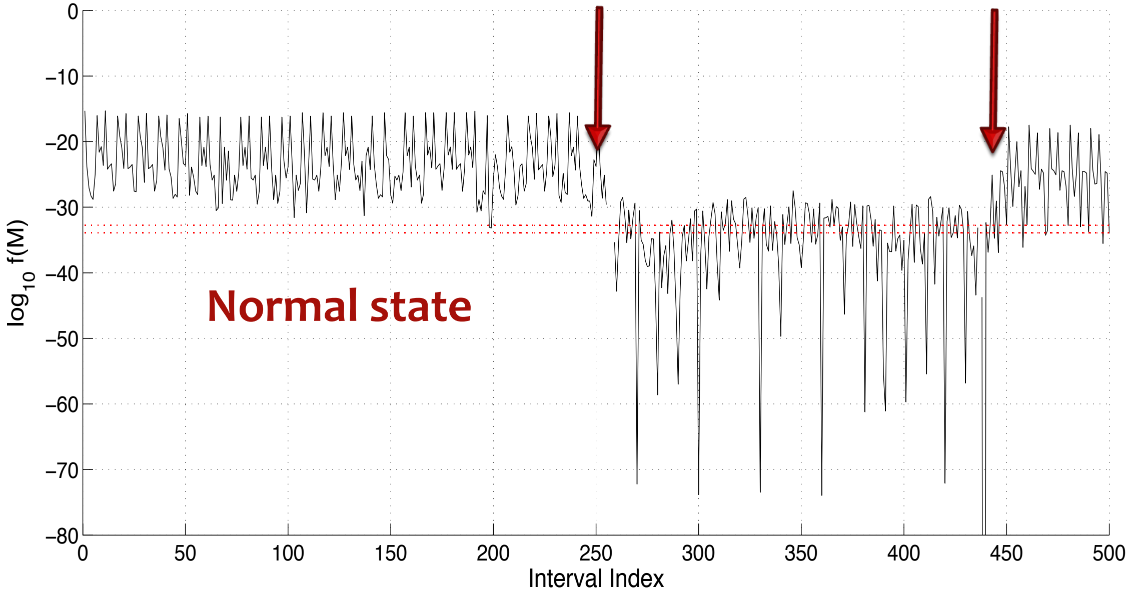
Anomaly Scenario 1 | Unknown Application Launch

Similarity to Patterns

Unknown app

launched

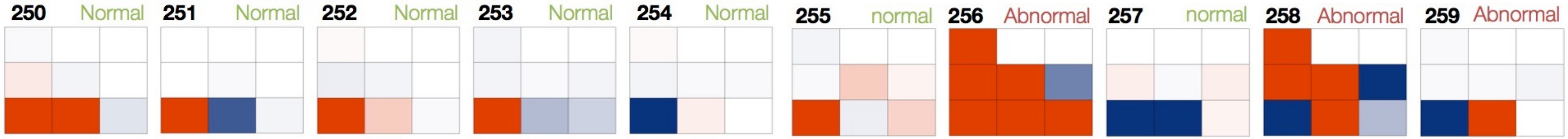
exited



Normal state

Time

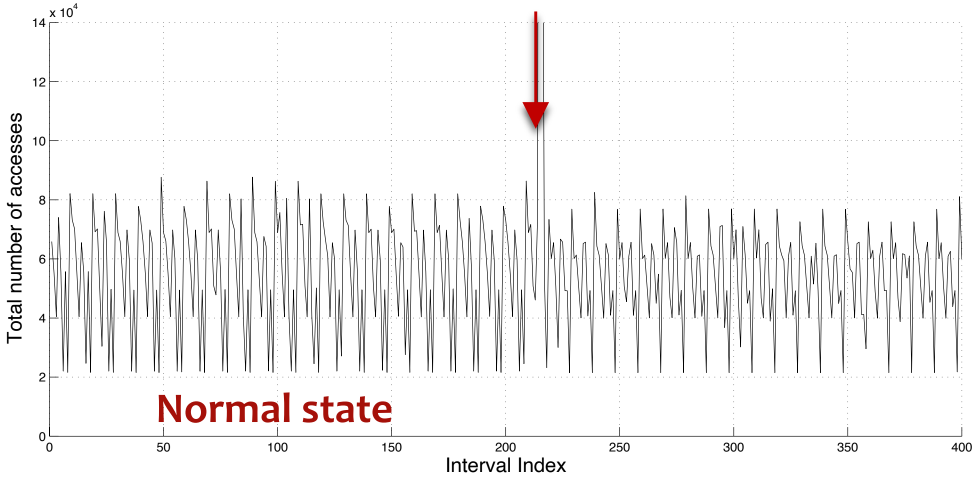
Log Probability Density of MHMs



Anomaly Scenario 2 | Kernel Rootkit

- A kernel rootkit (as a loadable kernel module) that **hijacks** 'read' system calls
  - Calls the original handler and then read the buffer
  - Note: the loadable kernel module is outside the target monitoring region

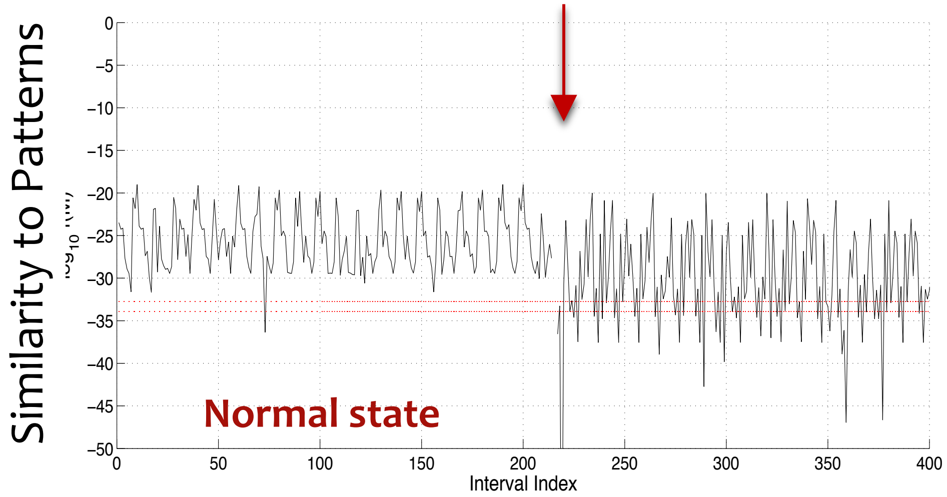
Rootkit launched



Normal state

Memory Traffic Volume

Rootkit launched



Normal state

Log Probability Density of MHMs

Analysis Overhead

- *How long does it take to decide whether an MHM is normal or not*
- Cost: Transforming to reduced space + probabilistic calculation using GMM

MHM Size (# cells)	**1472**	**368**	**1472**
# Eigenfaces	9	9	5
# GMM components	5	5	5
Avg. time	358 μs	100 μs	216 μs

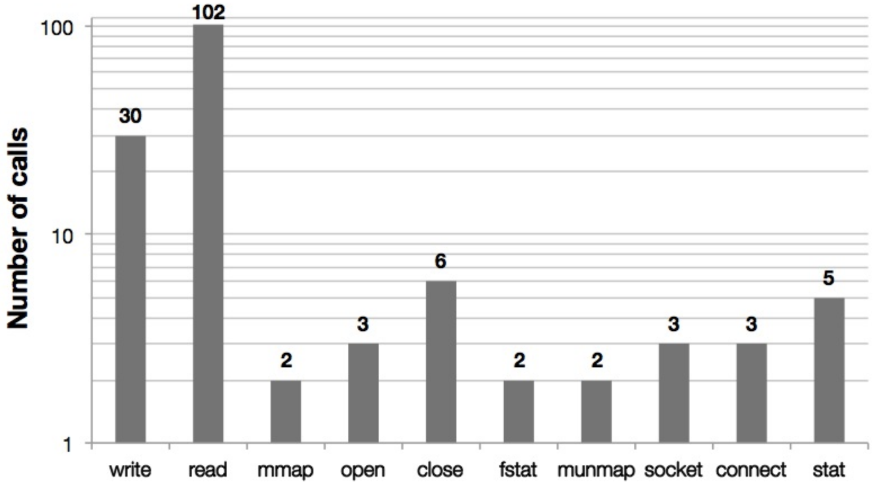
- Note: this analysis/transformation does not take place on critical path → happens in the secure core

Outline

- Background: Real-Time Systems & Simplex
- Cyber-Physical Systems Behavior
- SecureCore: Multicore-based Intrusion Detection
- Control Flow Monitoring
- Anomaly Detection using Kernel-memory Behavior
- **Execution Contexts Learned from System Call Distributions**
- Current and Future Work
- Conclusion

System Call Frequency Distribution (SCFD)

- Distribution of system call frequencies
  - How many times each system call type has been called by an application during one execution



- Intuition
  - *An application shows similar pattern for SCFDs*
    - When the input (e.g., from sensors) is similar
  - Malicious activities involve system calls
    - For privileged operations (example: socket, connect, write, …)
    - So, likely will show up as changes to SCFDs
- It's lightweight
  - No sequence. Just counting!

Challenges and Solutions

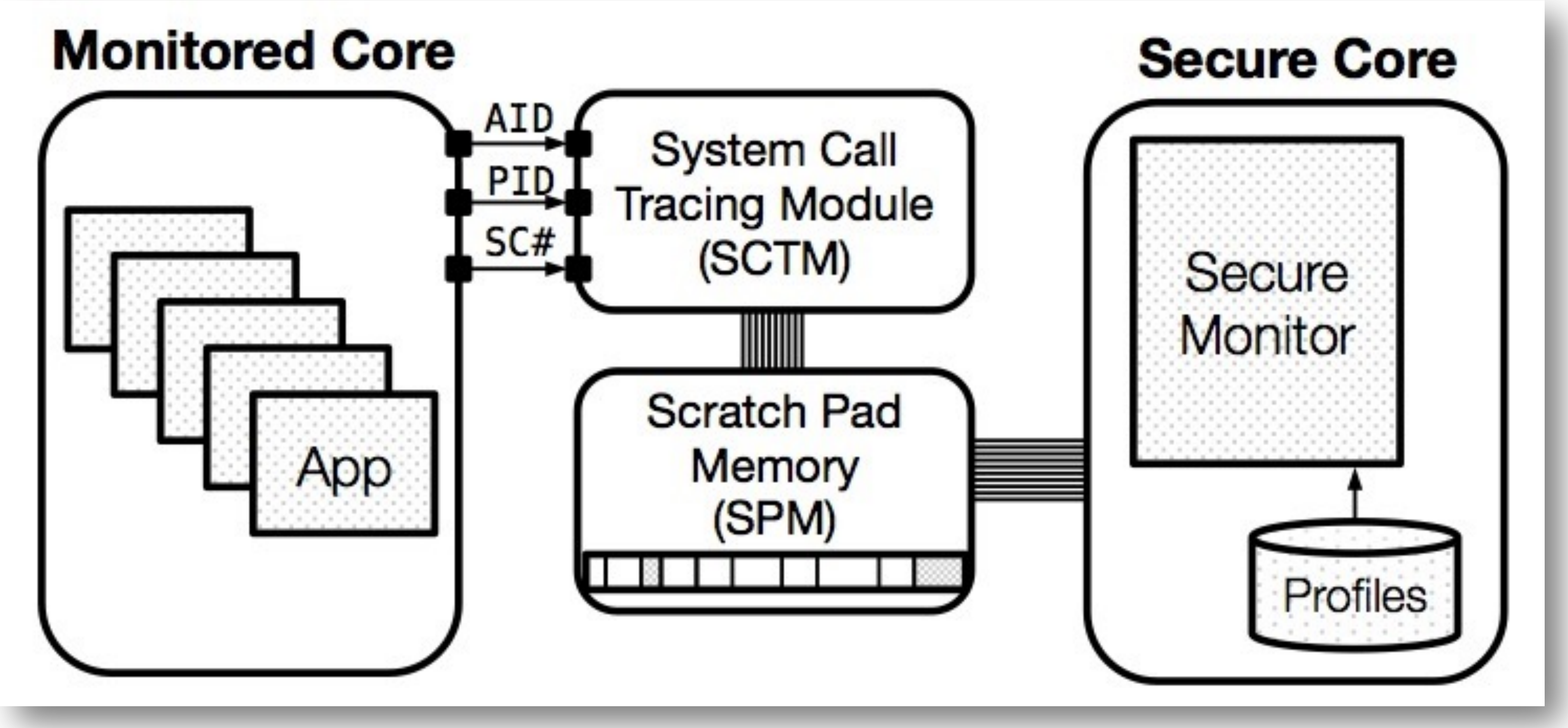
- Multiple execution contexts
  - Due to various execution modes and inputs
  - So, even benign SCFDs vary so greatly

  Solution: **Clustering SCFDs**

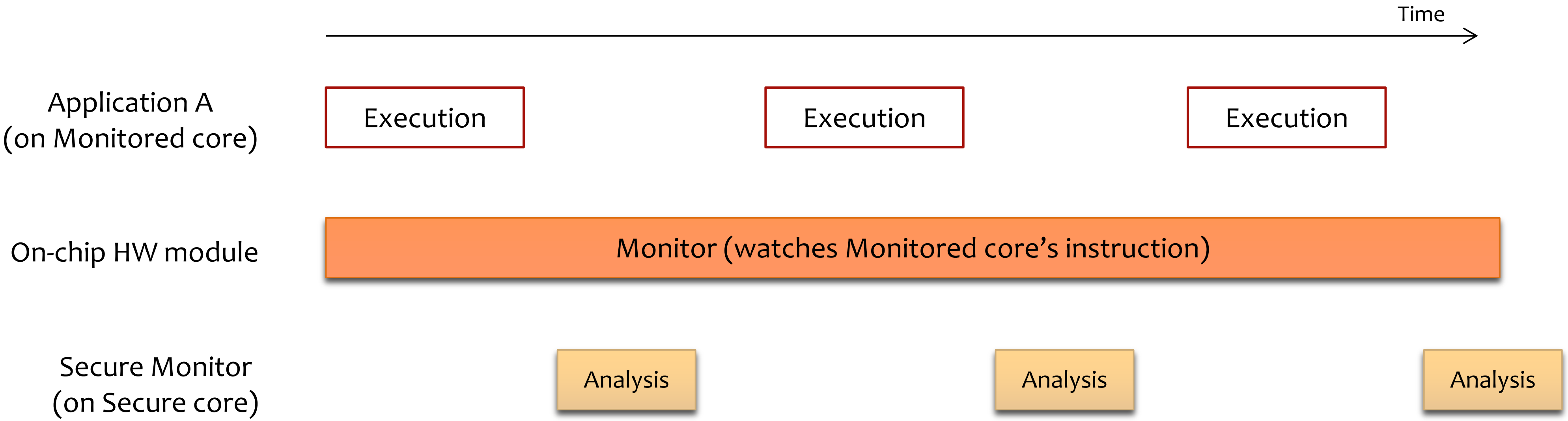
- How to catch system calls using hardware?
  - By not relying on system call interposition in SW level
  - Not easy to deceive HW-based approach

  Solution: **Catch system call 'instruction'**

SecureCore | System Call-based Detection Architecture

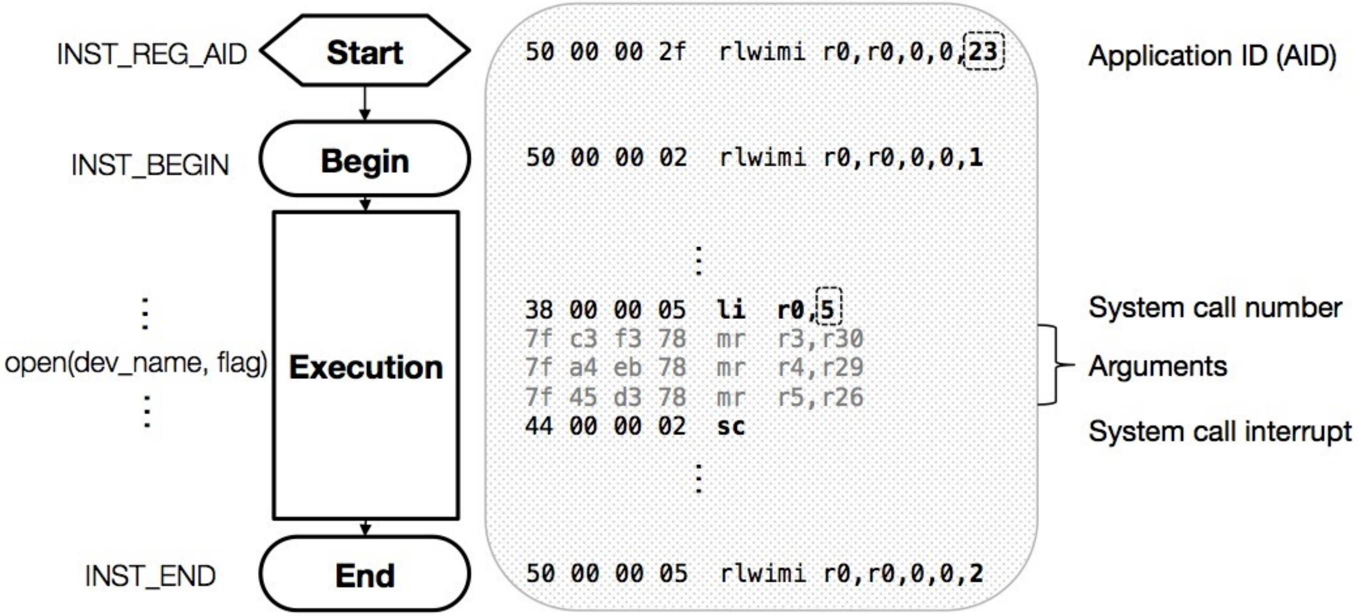


System Call-based Detection Process



- Same process for offline learning and online detection
  - Learning: collect a set of SCFDs
  - Detection: Analyze SCFDs one at a time

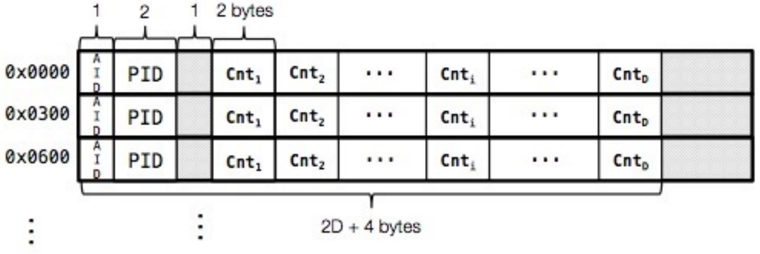
System Call Instruction



```
50 00 00 2f   rlwimi r0,r0,0,0,23

50 00 00 02   rlwimi r0,r0,0,0,1

                   ⋮

38 00 00 05   li    r0,5
7f c3 f3 78   mr    r3,r30
7f a4 eb 78   mr    r4,r29
7f 45 d3 78   mr    r5,r26
44 00 00 02   sc
                   ⋮

50 00 00 05   rlwimi r0,r0,0,0,2
```

open

- Catch the invocation of the designated instruction for system call
  - Instruction x86: `int 80h`, x64: `syscall`, PowerPC: `sc`
  - System call number x86: `eax`, x64: `rax`, PowerPC: `r0`
- SCTM updates the record (i.e., current SCFD) on scratchpad memory (SPM)
  - Using AID, PID and the system call number

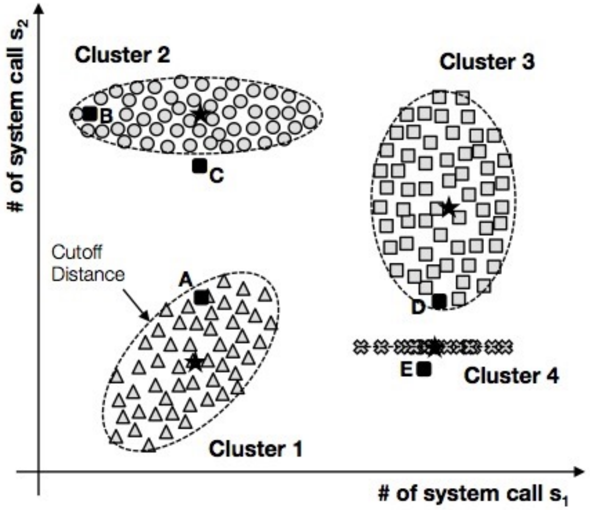


0x0000	AID	PID		Cnt_1	Cnt_2	···	Cnt_i	···	Cnt_D	
0x0300	AID	PID		Cnt_1	Cnt_2	···	Cnt_i	···	Cnt_D	
0x0600	AID	PID		Cnt_1	Cnt_2	···	Cnt_i	···	Cnt_D	

SCFD Clustering

- **K-means** with **Mahalanobis** distance
  - *Cluster SCFDs in the training set into **K** clusters*
    - Each cluster represents similar execution patterns
- Why not Euclidean distance, but Mahalanobis distance?

$$\sqrt{(\mathbf{x}^* - \boldsymbol{\mu})^T(\mathbf{x}^* - \boldsymbol{\mu})} \qquad \sqrt{(\mathbf{x}^* - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}^* - \boldsymbol{\mu})}$$



- To give more 'weight' to some system call types
  - Types with smaller variance
    - Ex: `execve`, `socket`
    - Δdist(Δ`execve`) > Δdist(Δ`read`)
    - Their change in the run-time should be small too
    - See Cluster 2 and Points B and C

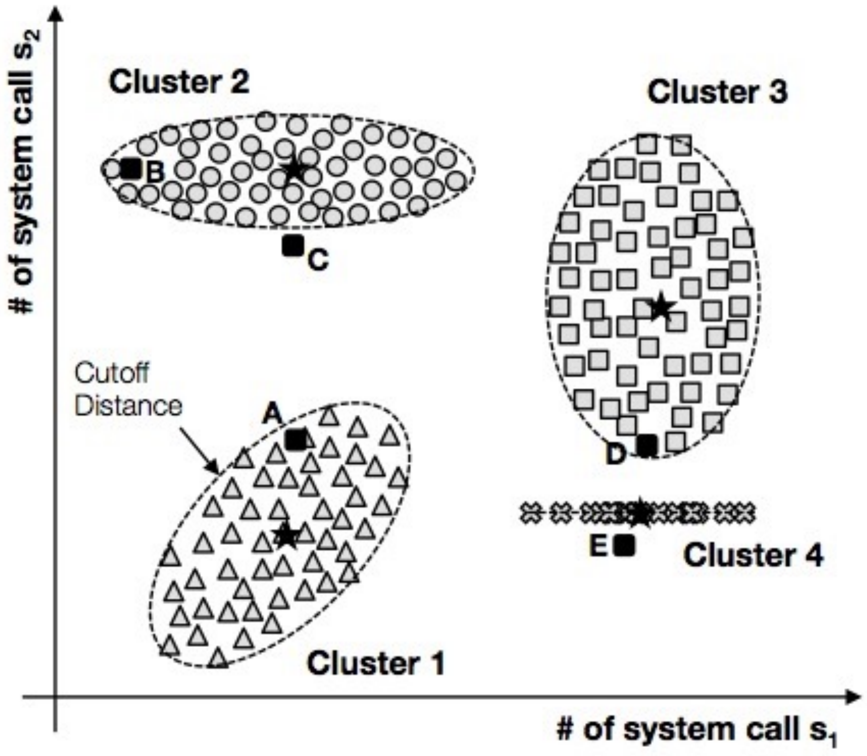
- Also, to learn correlation between types
  - i.e., how they should vary together
  - Ex: `socket` and `open`

How do we know what **K** is?

- Learn K by *global k-means*

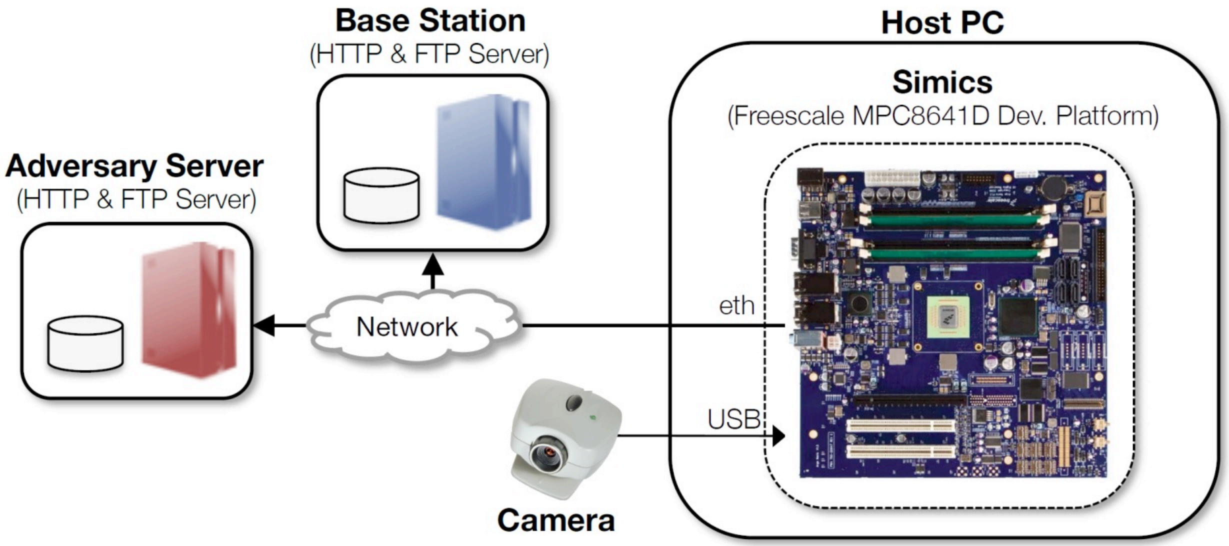
Legitimacy Test

- Given an SCFD to test,
  1. Find the closest cluster
  2. Test if the distance is within the cutoff distance
     - If not, the execution is malicious

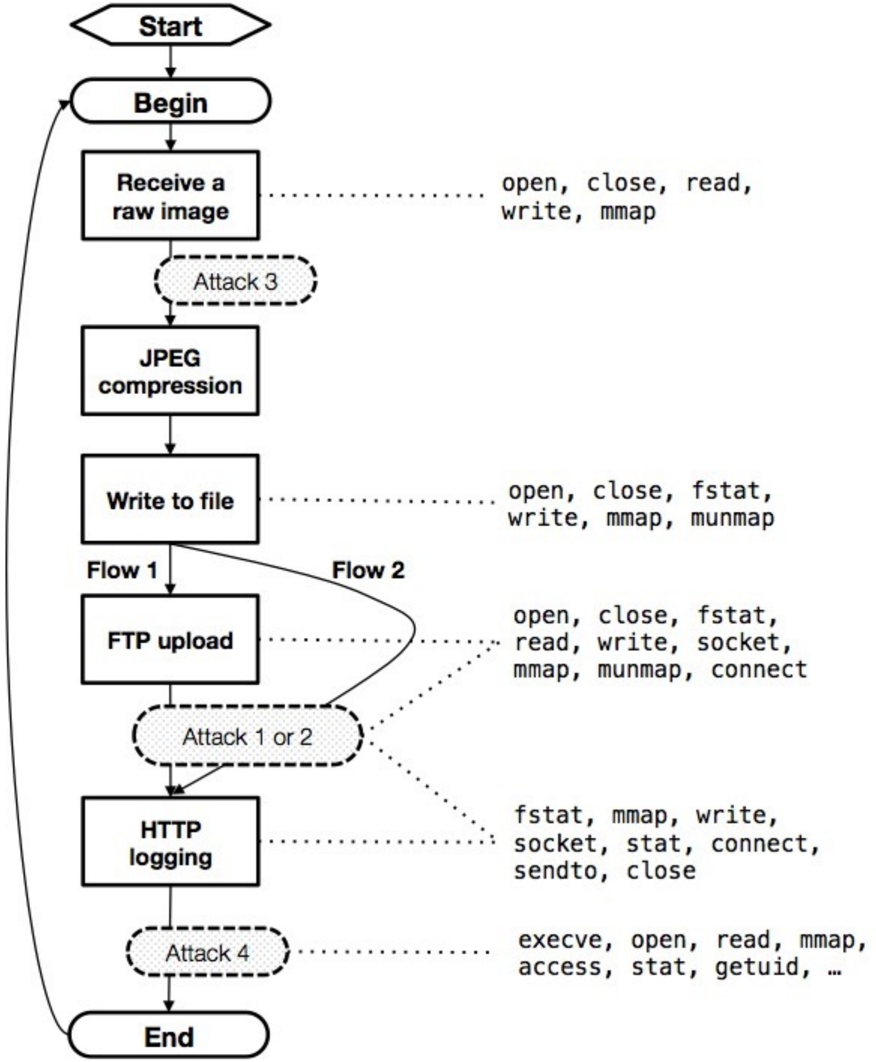


- A, B, D are legitimate
- C and E are malicious

Evaluation

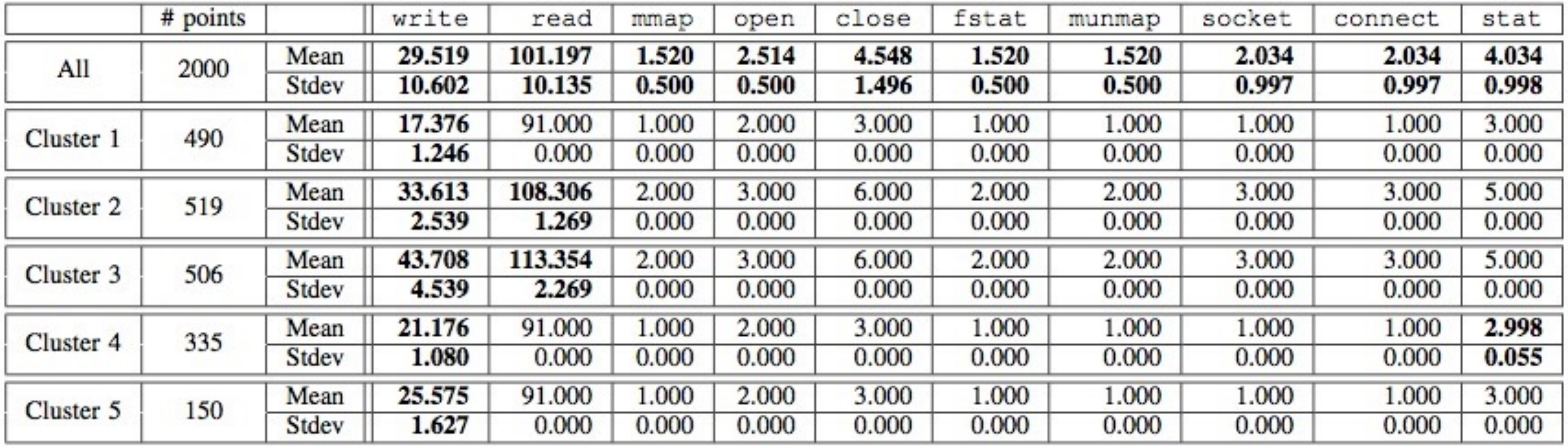


- PowerPC processor model on Simics
- Target Application
  - Raw image capture -> JPEG compression -> FTP upload -> HTTP logging
  - SCFDs vary due to 1) image content and 2) execution flow
- Attack scenarios
  - 1) Leak out user authentication information through HTTP
  - 2) Leak out the JPEG image through FTP
  - 3) `memset` the image array (which does not use any system calls)
  - 4) Shellcode (that spawns `/bin/sh`)


Start
Begin
Receive a raw image
open, close, read, write, mmap
Attack 3
JPEG compression
Write to file
open, close, fstat, write, mmap, munmap
Flow 1
Flow 2
FTP upload
open, close, fstat, read, write, socket, mmap, munmap, connect
Attack 1 or 2
HTTP logging
fstat, mmap, write, socket, stat, connect, sendto, close
Attack 4
execve, open, read, mmap, access, stat, getuid, …
End

Evaluation

- Training set
  - 2000 SCFDs
  - **14** system call types
- Global k-means found **5** clusters

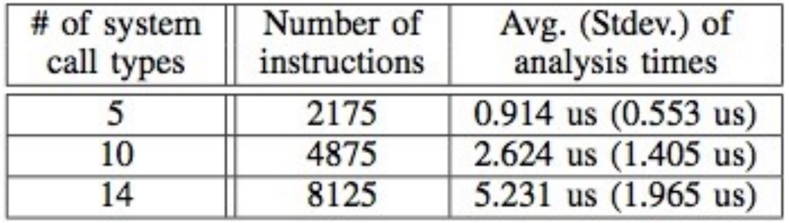


	# points		write	read	mmap	open	close	fstat	munmap	socket	connect	stat
All	2000	Mean	**29.519**	**101.197**	**1.520**	**2.514**	**4.548**	**1.520**	**1.520**	**2.034**	**2.034**	**4.034**
		Stdev	**10.602**	**10.135**	**0.500**	**0.500**	**1.496**	**0.500**	**0.500**	**0.997**	**0.997**	**0.998**
Cluster 1	490	Mean	**17.376**	91.000	1.000	2.000	3.000	1.000	1.000	1.000	1.000	3.000
		Stdev	**1.246**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 2	519	Mean	**33.613**	**108.306**	2.000	3.000	6.000	2.000	2.000	3.000	3.000	5.000
		Stdev	**2.539**	**1.269**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 3	506	Mean	**43.708**	**113.354**	2.000	3.000	6.000	2.000	2.000	3.000	3.000	5.000
		Stdev	**4.539**	**2.269**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 4	335	Mean	**21.176**	91.000	1.000	2.000	3.000	1.000	1.000	1.000	1.000	**2.998**
		Stdev	**1.080**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.055**
Cluster 5	150	Mean	**25.575**	91.000	1.000	2.000	3.000	1.000	1.000	1.000	1.000	3.000
		Stdev	**1.627**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

- 300 execution traces for each attack scenario
  - Attack 1 and 2 are detected well because of network-related system calls
  - Attack 3 is detected because of fewer calls of read and write
  - Attack 4 is detected because it calls execve which was never seen
- False positive rate: around 1% (depends on cutoff distance)

Evaluation

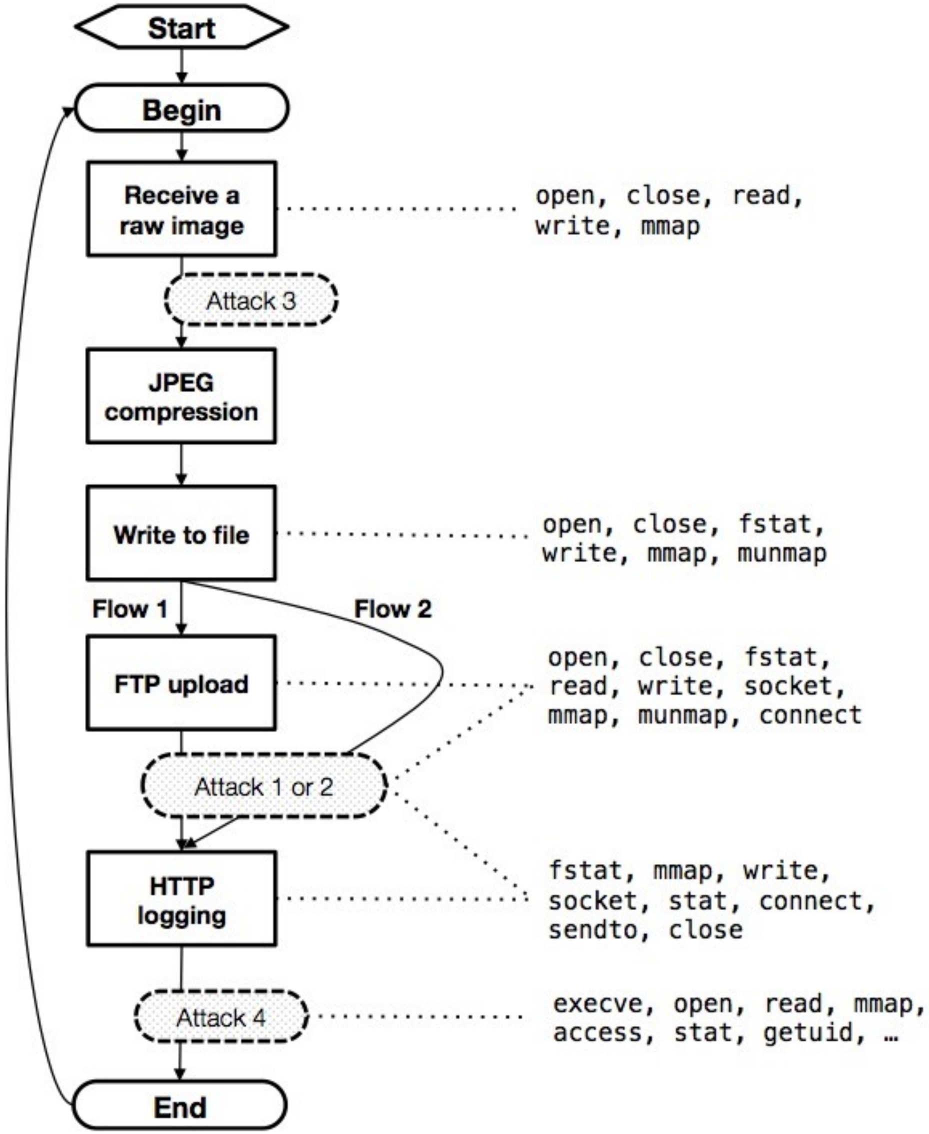
- Analysis overhead
  - Finding the closest cluster among 5 clusters



# of system call types	Number of instructions	Avg. (Stdev.) of analysis times
5	2175	0.914 us (0.553 us)
10	4875	2.624 us (1.405 us)
14	8125	5.231 us (1.965 us)

    - # instructions is measured on Simics and the times are
    - measured on a dual-core machine

- Detection problems:
  - Attack 2 is enabled on Flow 2.
  - Not differentiable from a benign execution on Flow 1


Start
Begin
Receive a raw image
open, close, read, write, mmap
Attack 3
JPEG compression
Write to file
open, close, fstat, write, mmap, munmap
Flow 1
Flow 2
FTP upload
open, close, fstat, read, write, socket, mmap, munmap, connect
Attack 1 or 2
HTTP logging
fstat, mmap, write, socket, stat, connect, sendto, close
Attack 4
execve, open, read, mmap, access, stat, getuid, …
End

Outline

- Background: Real-Time Systems & Simplex
- Cyber-Physical Systems Behavior
- SecureCore: Multicore-based Intrusion Detection
- Control Flow Monitoring
- Anomaly Detection using Kernel-memory Behavior
- Execution Contexts Learned from System Call Distributions
- **Current and Future Work**
- Conclusion

Current and Future Work

- Individual behavioral signals detect certain problems
  - **Combination of multiple signals** to improve detection accuracy and increased difficulty for would-be attackers
- Monitor multiple cores and “long term detection”
- Full system monitoring (multiple tasks/cores + OS)
- Demonstration on actual real-time systems
  - Developing UAV platforms & hardware-in-the-loop simulators
  - Working with power system vendors for such demonstrations
- Extension to mobile and other general-purpose devices

Conclusions

- Intrusion detection for Cyber-Physical Systems
- Focus on specific characteristics of such systems
- System architecture = isolated hardware + novel intrusion detection methods
- Multiple solutions
  - Hardware: Multicore, FPGA-based, simulation platform, etc.
  - Analysis: compile-time analysis, statistical/learning-based approach
  - Different behavioral signals: timing, memory, control flow, system calls

- Detect intrusions in short timeframes → **prevent harm to physical systems**
- Resilient to attackers gaining administrative access on main systems

Thanks

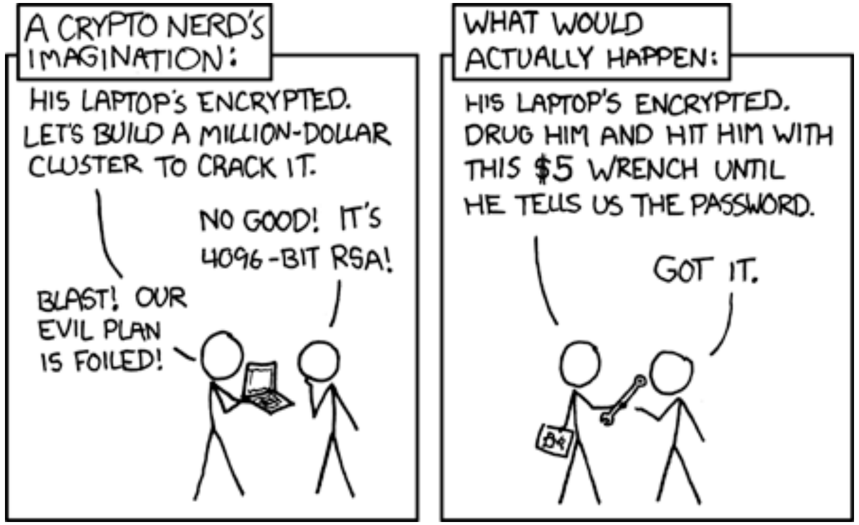
Co-conspirators:

- Lui Sha [UIUC], Marco Caccamo [UIUC], Frank Mueller [NCSU], Heechul Yun [Kansas], Stanley Bak [AFRL], Emiliano Betti [UIUC/Univ. of Rome]

Students [People that do the real work!]:

- **Man-Ki Yoon** [UIUC], Fardin Abdi [UIUC], William Condon [UIUC] , Yi Lu [UIUC], Joel van der Woude [UIUC], Chris Zimmer [NCSU]

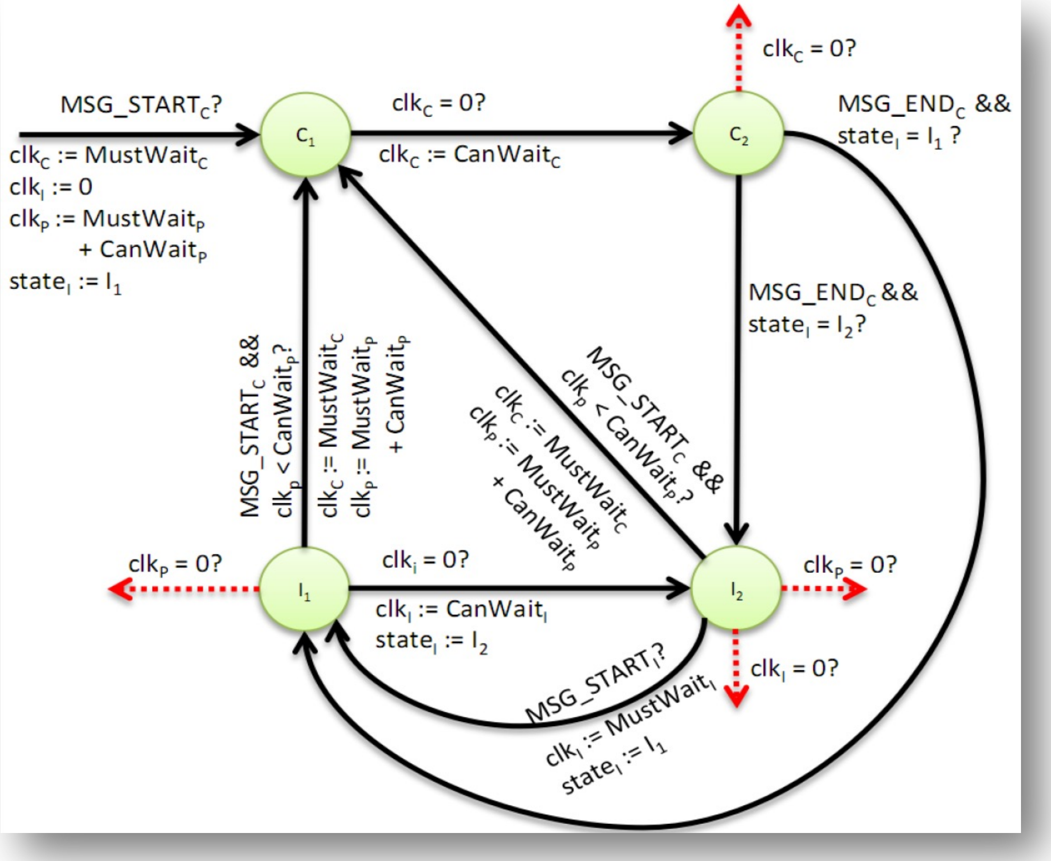
RUN IN A BLIND PANIC
OK, IT'S NOT PERFECT, BUT I'M SURE A MORE EFFICIENT AND EFFECTIVE SYSTEM WILL BE DEVELOPED IN THE FUTURE
SECURITY ALERT
HIDE IN A TREE
ALL CLEAR


A CRYPTO NERD'S IMAGINATION:
HIS LAPTOP'S ENCRYPTED. LET'S BUILD A MILLION-DOLLAR CLUSTER TO CRACK IT.
NO GOOD! IT'S 4096-BIT RSA!
BLAST! OUR EVIL PLAN IS FOILED!
WHAT WOULD ACTUALLY HAPPEN:
HIS LAPTOP'S ENCRYPTED. DRUG HIM AND HIT HIM WITH THIS $5 WRENCH UNTIL HE TELLS US THE PASSWORD.
GOT IT.

Backup Slides

Behavior Monitoring Module on FPGA

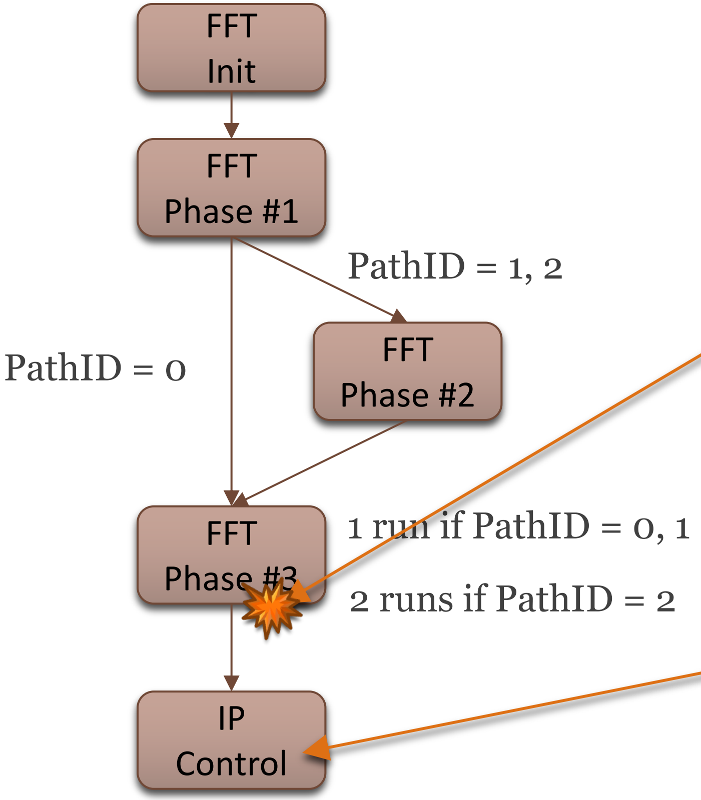
- Finite State Machine (on FPGA) to detect timing model violations

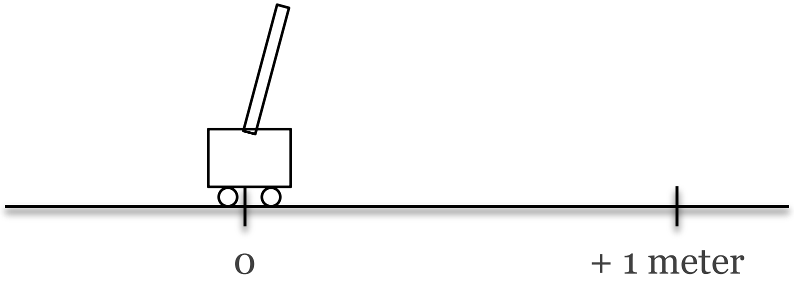


- S3A can be used to detect intrusions
  - Even if code has complex control flow (branches/loops/etc.)
  - Modification of this FSM

Application Model

- IP Control + FFT (EEMBC Suite)



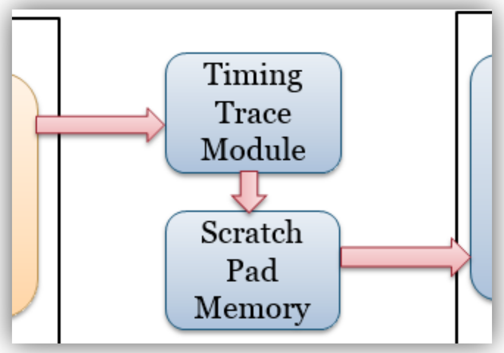


- Injected at the end of
- **Simple loop** (some array copy)
  for 1,3,5 loops
- Activated when the cart passes by
- **Execute randomly** thereafter
  - Loop execution

Timing Profile

- 10,000 runs (no malicious code activation)
- ‘ksdensity’ function in Matlab

Hardware Modification | Timing Trace Module



```
  () {
    INST_REG_PID;
    …
    INST_ENABLE_TRACE;
    …
    foo();
    …
    INST_DISABLE_TRACE;
}
```

```
foo() {
    INST_TRACE;
    Do_something();
    INST_TRACE;
    Do_something();
    INST_TRACE;
}
```

INST_REG_PID	*rlwimi 0,0,0,0,1*
INST_ENABLE_TRACE	*rlwimi 0,0,0,0,2*
INST_DISABLE_TRACE	*rlwimi 0,0,0,0,3*
INST_TRACE	*rlwimi 0,0,0,0,4*

Trace Instructions

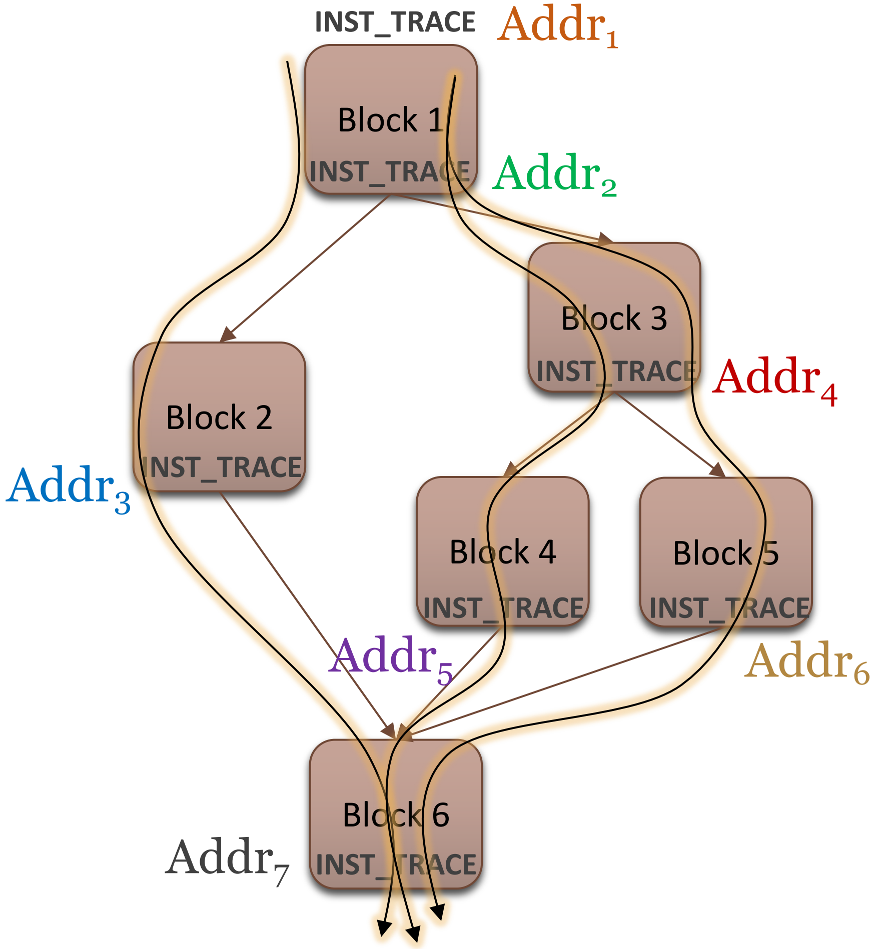
4 Bytes

0x000	PID	BA	$Addr^{Head}$	$Addr^{Tail}$
0x010	Timestamp *j+1*		Addr *j+1*	
	…			
0x8a0	Timestamp *i*		Addr *i*	
0x8b0	Timestamp *i+1*		Addr *i+1*	
0x8c0	Timestamp *i+2*		Addr *i+2*	
	…			
0xFF0	Timestamp *j*		Addr *j*	

SPM Layout

- PID registration for preventing traces from being forged
- BA: Base Address (i.e., PC of INST_REG_PID)
- Read Timestamp and Program Counter from the processor registers
- $Addr_i = PC_i - BA$ (relative address from BA)

Raw Trace Collection



(Addr$_{1,}$ t$_1$)

(Addr$_{2,}$ t$_2$)

(Addr$_{3,}$ t$_3$)

(Addr$_7$, t$_4$)

(Addr$_{1,}$ t$_5$)

(Addr$_{2,}$ t$_6$)

(Addr$_{4,}$ t$_7$)

(Addr$_{6,}$ t$_8$)

(Addr$_7$, t$_9$)

(Addr$_{1,}$ t$_{10}$)

(Addr$_{2,}$ t$_{11}$)

(Addr$_{4,}$ t$_{12}$)

(Addr$_{5,}$ t$_{13}$)

(Addr$_7$, t$_{14}$)

…

Trace Tree Generation

$(Addr_1, t_1)$

$(Addr_2, t_2)$

$(Addr_3, t_3)$

$(Addr_7, t_4)$

From a trace tree, we can get

- Execution time samples (each node)
- Legitimate execution flows

$(Addr_7, t_9)$

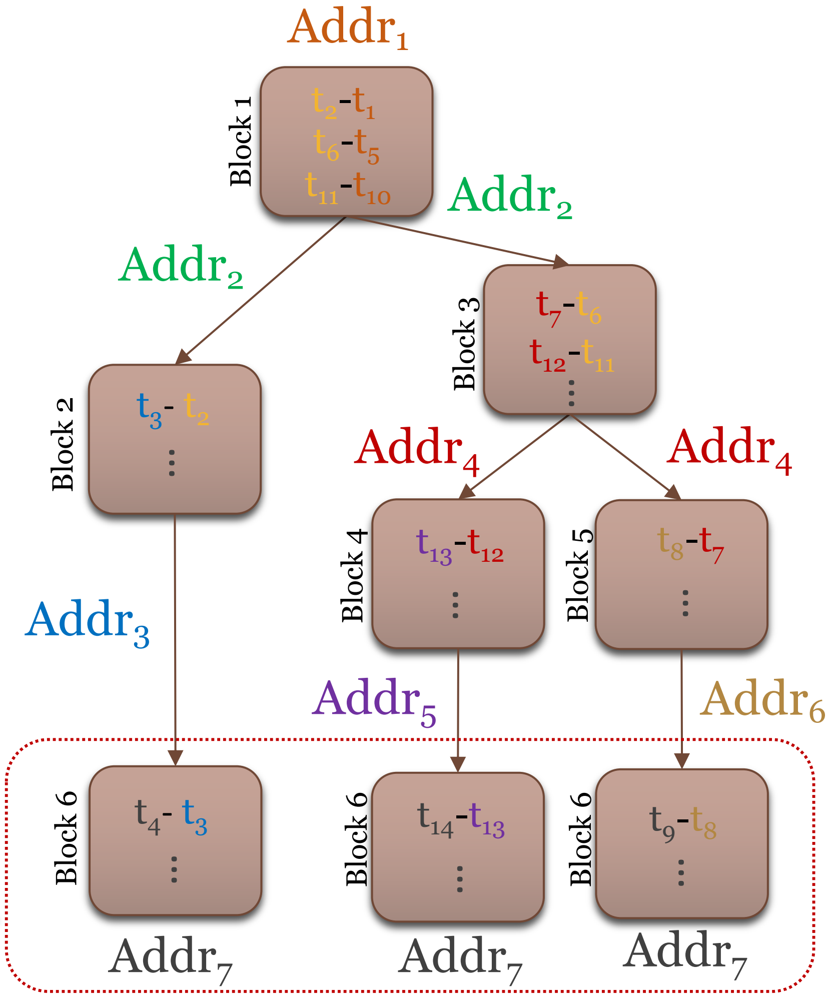
$(Addr_1, t_{10})$

Same execution block,
but on different paths.

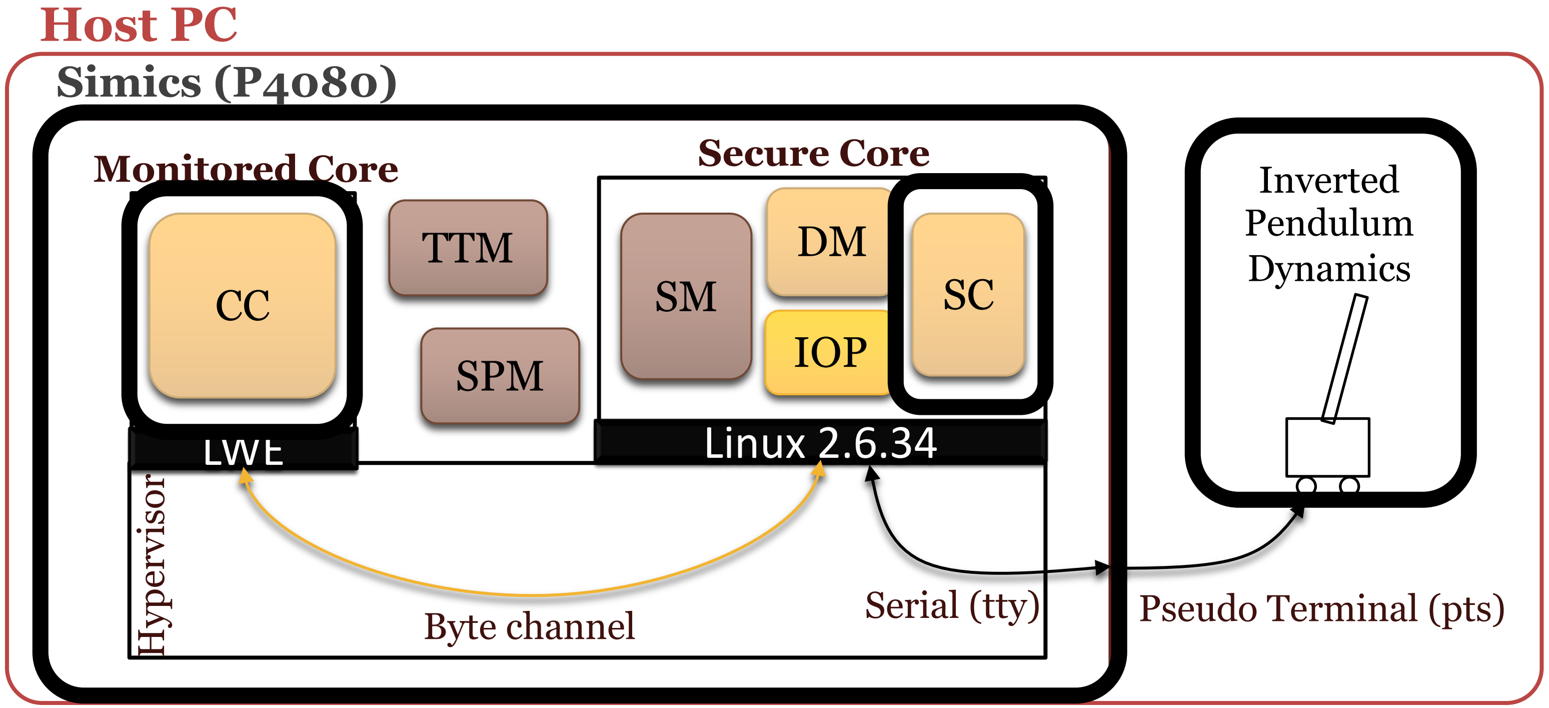
Each has its own timing profile

$(Addr_7, t_{14})$

...



SecureCore Implementation



Freescale P4080 on Simics

- Only two cores (Core 0 and 1)
- Cache (L1 and L2) and Bus models for system effects
- ISA modification for trace instruction

Inverted Pendulum Control

- Controller + Dynamics
- Generated from Simulink IP model
- (Cart position, Rod's angle)

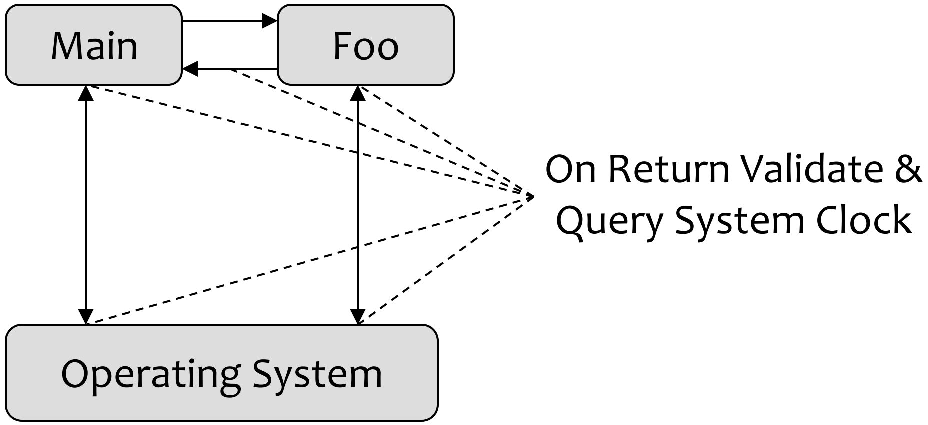
Early Work | WCET-based Intrusion Detection *

- WCET: guaranteed worst-case execution time on a specific hardware platform
- Security violations in hard real-time systems → **code injection** attacks
  - doesn’t even have to be fancy → just *cause delays* in hard real-time systems
- Approach: use WCET values to validate programs
  - Instrument task checks throughout entire system
- Two techniques:
  - Timed Return Path Security (TRPS)
  - Timed Code Section Security (TCSS)

* **[ICCPS 2010]** “Time-Based Intrusion Detection in Cyber-Physical Systems”
by C. Zimmer, B. Bhatt, F. Mueller and S. Mohan.

Timed Return Path Security (TRPS)

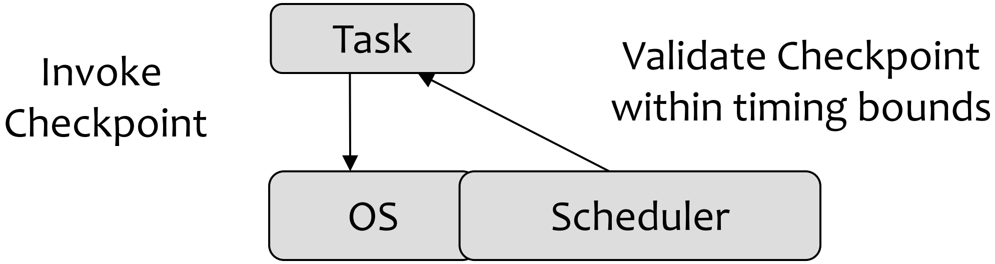
- Instrument **return paths** of functions with timing checks
- Perform timing analysis on small code regions to obtain WCET
- Validate on *return from function calls* → check to see if WCET exceeded?
- Also verify → order of syscalls



- Drawbacks:
  - Covert attacks that maintain consistent state → not detected
  - Requires clock protection → common in such systems anyways

Timed Code Section Security

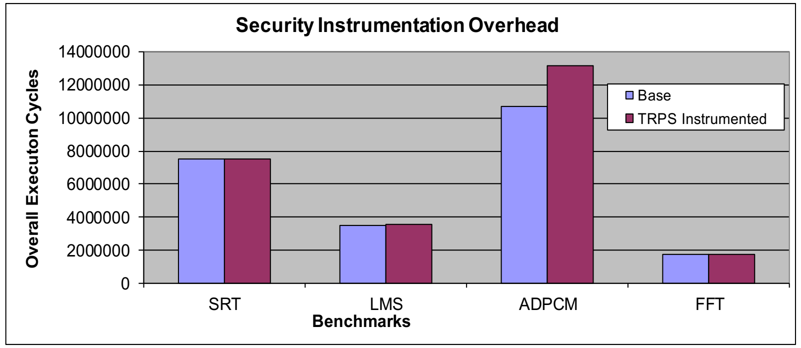
- Use **periodic scheduler interrupts**
- Use WCET and sequence of *checkpoints* → calculate WCET to next checkpoint
- Intrusion detection at next preemption/checkpoint
- Checks managed by **scheduler**



- At deadlines, verify if all critical checkpoints have been hit
- Use in conjunction with TRPS

Intrusion Detection using WCET

- TRPS:
  - low overheads
  - local checks



- TCSS:
  - Early detection → within 20 µs in most cases
  - Timer Interrupt dependent

Task	Hijack Location	Time Attack Ends
FFT	FFT Method Return	1660 cycles
LMS	LMS Method Return	869 cycles
CNT	Scheduler Check 2	1690 cycles

- Drawbacks
  - Depends on WCET → can be easily bypassed
  - Rely on software mechanisms to provide protection

Outline

- Background: Real-Time Systems
- Background: Simplex
- Cyber-Physical Systems Behavior
- Early Work: Worst-case Execution Time (WCET) based detection
- **S3A: Secure System Simplex Architecture**
- SecureCore: Multicore-based Intrusion Detection
- Control Flow Monitoring
- Current and Future Work
- Conclusion

S3A: Secure System Simplex Architecture *

- Intrusion detection and safety for individual nodes in RT control systems
- Uses behavior-based monitoring of system → **execution time,** in this case
- Combined with trusted hardware component on **separate FPGA**
- Essentially builds upon System-level Simplex
    - **Includes cyber state** to protect against malware directed at complex controller

- **Maintains safety even if attackers obtain administrative access to controller**

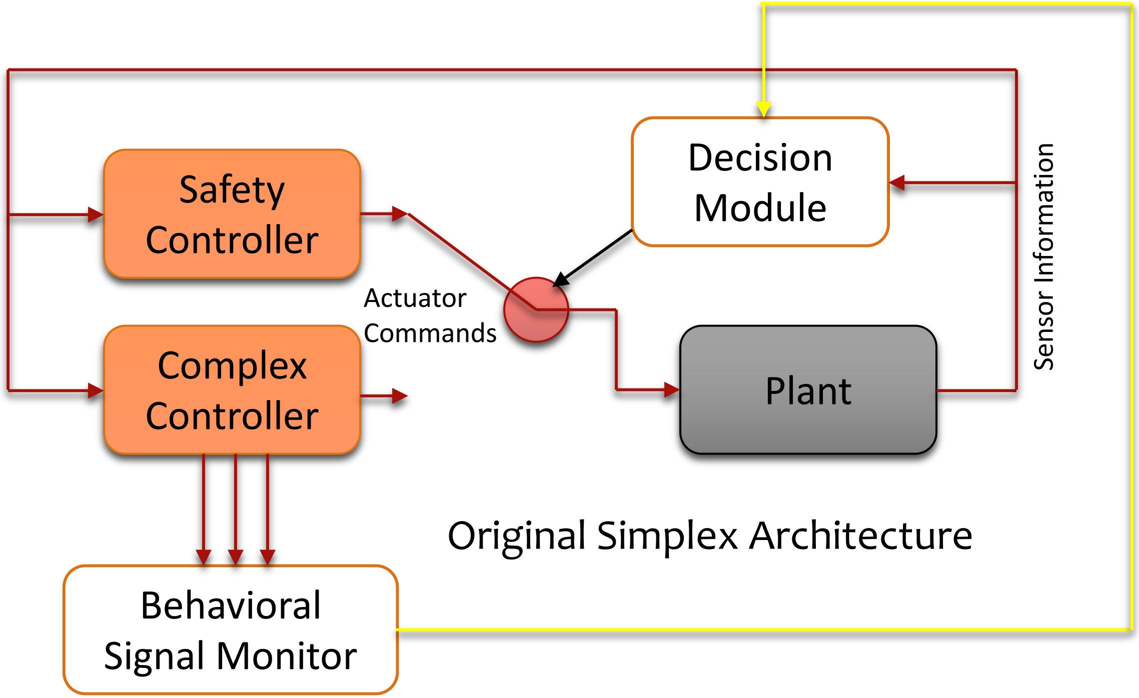
*** [HiCONS 2013]** "S3A: Secure System Simplex Architecture for Enhanced Security and Robustness of Cyber-Physical Systems" by S. Mohan et al. in HiCONS 2013.

S3A Logical Architecture

- Timing-based

  Behavioral Signals:

  - Exec time too small
  - Exec time too large
  - Activation Period too small
  - Activation Period too large
  - Idle task behavior

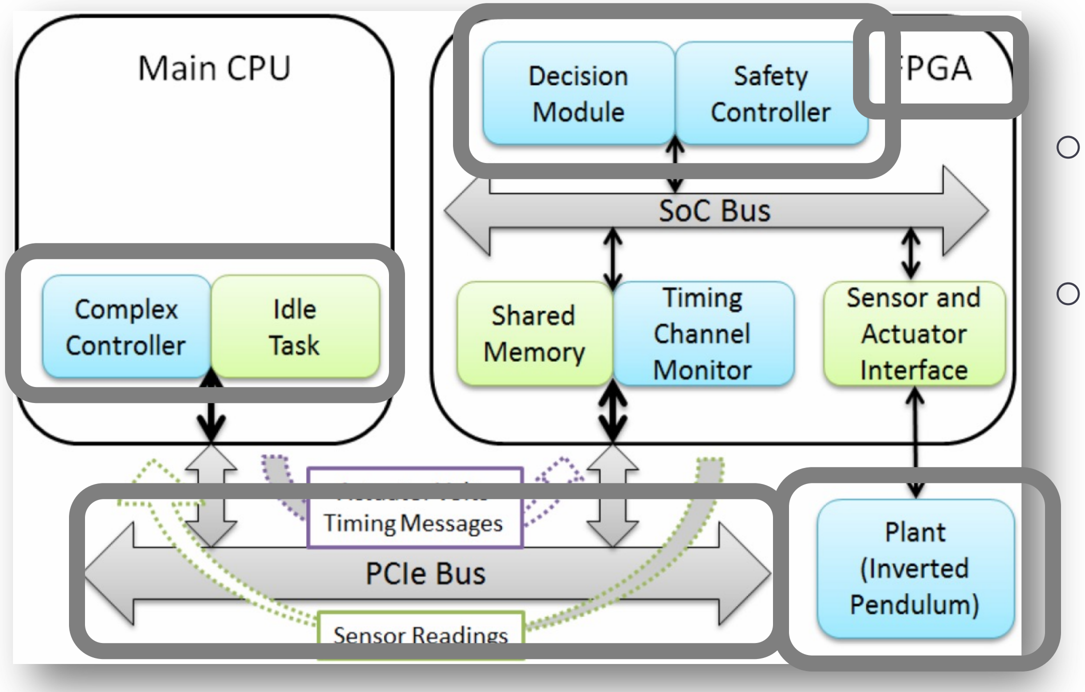


Original Simplex Architecture

- Behavior Signal Monitor
  - Checks if system is within performance envelope
  - Detect attacks **early**
  - Could even trigger restoration of Complex Controller

S3A Implementation

- Trusted hardware module for this implementation: FPGA
  - Contains: Decision Module, Behavioral Signal Monitor, Simple Controller
  - Communicates with and monitors Complex Controller
- Implementation Overview:



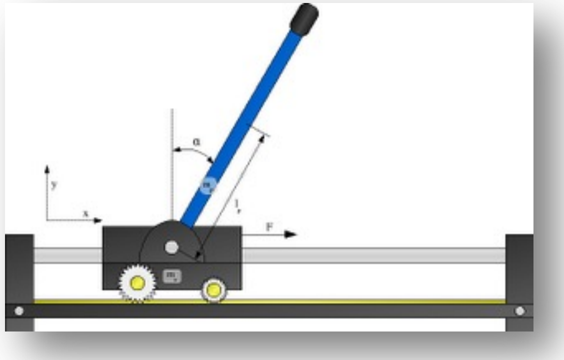
- Advantage of using FPGA:
  - Easy to retrofit existing systems
- Cannot be modified in field
  - Programmability turned off

S3A Implementation Details

- Hardware Components:

Component	Details
Inverted Pendulum	Quanser IP01
FPGA	Xilinx ML505
Computer with Complex Controller	Intel Quad Core 2.6 GHz
Operating System	Linux Kernel ver. 2.6.36
Timing Profile (dynamic timing analysis)	Timestamp Counter (can use other mechanisms e.g.: performance counters)

- Test Plant (control system): Inverted Pendulum
  - Rod must be maintained in upright position
  - Rod must be located near the center of the track

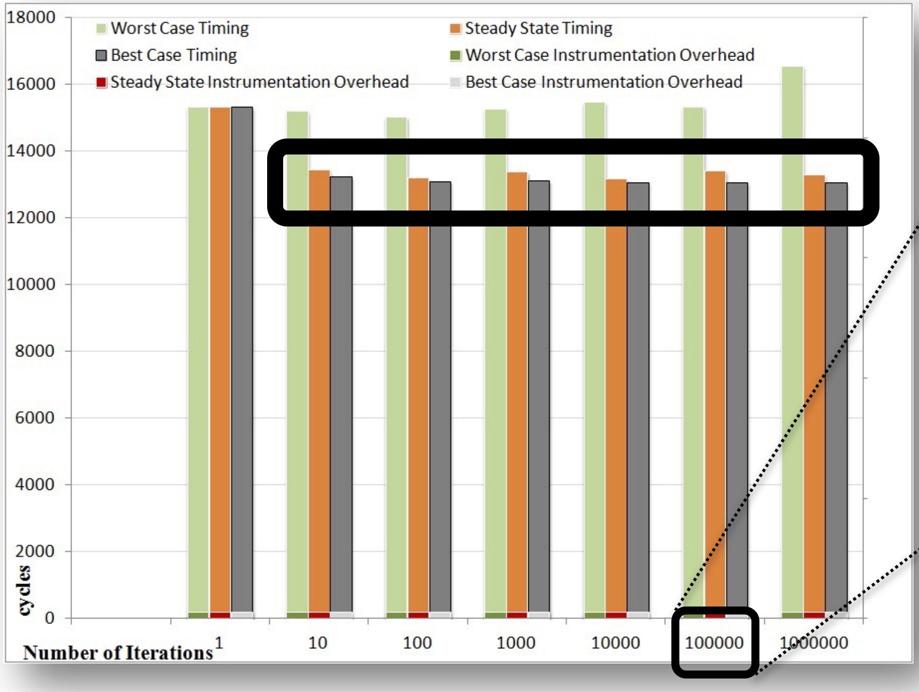


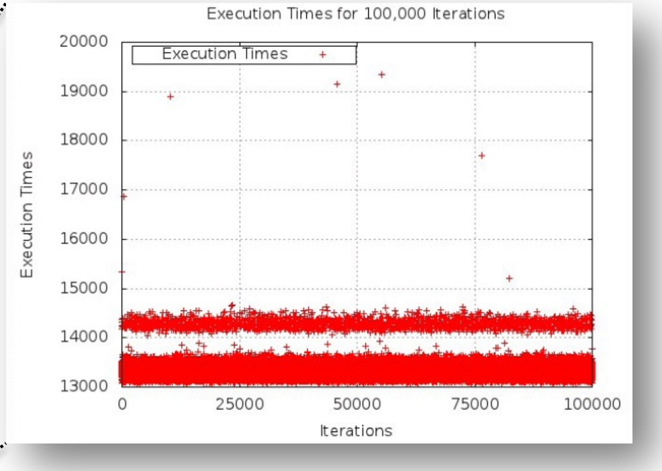
S3A Implementation Details (contd.)

- Unsafe states for plant (inverted pendulum)
  - Buggy/malicious code should not make pendulum fall over OR
  - Deviate too far from the center of the track
- FPGA
  - Monitors sensor readings on the bus (PCIe bus of the control computer)
  - Monitors the actuation commands being sent to the plant
- Behavior (timing) signal information
  - **Sent to FPGA from computer via memory mapped regions**
  - Actuation commands are also written on shared memory region
- Complex Controller
  - Few branches and statically bound loops → easy to analyze execution time

Timing Results

- Control code executed multiple times inside a loop





Most execution time stabilizes within:
1, 590 cycles i.e. ~ **0.6 μs** at **2.67** GHz

- As number of iterations increases, system effects make WCET worse
- Double-banded execution time behavior → cache replacement policy
- **Malicious code** → extra loop iterations to increase execution time

Intrusion Detection

- Overhead for sending a single timing message to FPGA: **50 ns**
- Jitter of timing messages due to interconnect: **0.6 µs**

Measured Quantity	Time (µs)
Control Task Exec. Time (single iteration)	4.8 – 5.4
Interconnect Extra Jitter	~ 0.6
Enforced Iteration Time	4.5 – 5.7
Timing Anomaly Detection Time (for IP)	5.7
Timing Message CPU Overhead	0.05
Simplex (vanilla) Anomaly Detection Time	10,000

- FPGA can detect an intrusion within **5.7 µs**
- Anything that changes timing by **0.6 µs** will be detected.

S3A Review

Limitations:

- System needs to be designed with S3A in mind
- Attacker may be able to replicate behavioral signal or hide its presence
  - E.g.: if code experiences significant timing deviations
- Trusted module depends on main system to provide critical information
  - Such as the timestamps sent from computer to FPGA → can be easily faked
- Complex controller code may not be easily analyzable to get strict exec. times
  - Significant engineering effort to get precise execution times

On the plus side,

- Physical system maintained in safe state → detection faster than Simplex
- Even if **attackers gains administrative privileges** on main system
- Not specific to any particular (classes of) attacks

Outline

- Background: Real-Time Systems
- Background: Simplex
- Cyber-Physical Systems Behavior
- Early Work: Worst-case Execution Time (WCET) based detection
- S3A: Secure System Simplex Architecture
- **SecureCore: Multicore-based Intrusion Detection**
- CFM: Control Flow Monitoring
- Other Current and Future Work
- Conclusion

SecureCore*

- Multicore architectures are here to stay
- Can use **redundancy in multiple cores** to improve security of CPS
- Multicore-based **on-chip hardware** for monitoring behavior of tasks

- Aims to address shortcomings of S3A
  - Directly obtain information from processor → don't rely on monitored task
  - Analyze more complex control flows and behavioral variations

- Uses **statistical/machine learning approach** to creating behavior profiles

*** [RTAS 2013]** "SecureCore: A Multicore based Intrusion Detection Architecture for Real time Embedded Systems" by M. K. Yoon, S. Mohan, J. Choi, J. E. Kim and L. Sha in RTAS 2013.