SPARCHS Site Visit

6 September 2012

Agenda

Time	Topic	Speaker
11:35	Native Instruction Set Randomization	Kanad Sinha
12:05	KBouncer: ROP mitigation	Michalis Polychronakis
12:40	WHISK: DIFT For Heterogeneous Systems	Joel Porquet
1:15	Autotomic Binary Structure Randomization	Ang Cui
1:50	Efficient Deterministic Multithreading through Schedule Relaxation	Heming Cui
2:25	Liberty Architecture	Deep Ghosh
3:00	SPARCHS Architecture Spec. v0.1, Wrap up	Simha Sethumadhavan

Highlights

- ISR: Native implementation
  - Overhead is 0.2% As little as 5 lines of Verilog code
- DIFT: On heterogeneous systems-on-chip
  - Overhead without tagging ~0%; with tagging linear to percentage of memory that is tainted
- ROP: MSR Blue Hat Winner
- RBS: Spin off on Firmware security

Native Instruction Set Randomization

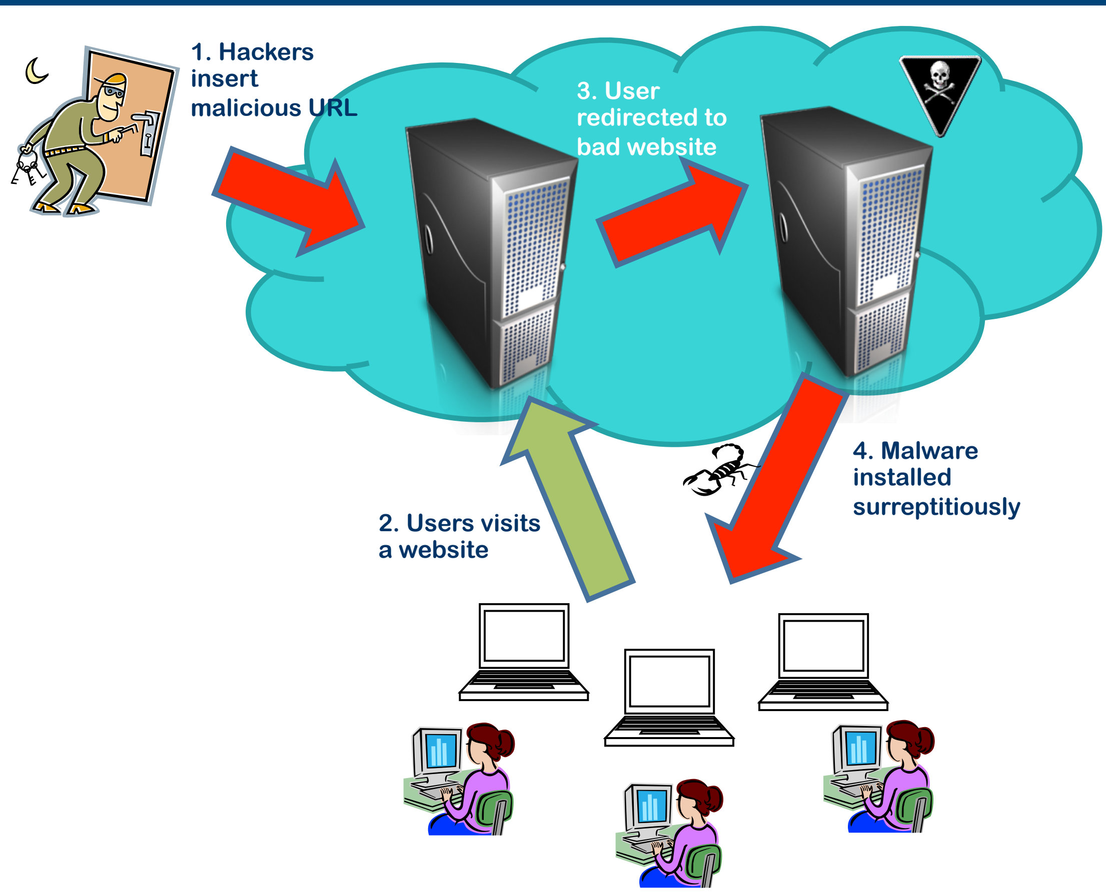
Kanad Sinha, Vasileios Kemerlis,
Vasilis Pappas, Angelos Keromytis,
Simha Sethumadhavan

The Problem

Today…

- Systems are largely similar
  - *e.g.,* Windows on x86, Android on ARM
- System security mature but far from flawless
- Attacks difficult to develop, but widely deployable
  - Similar flaws on similar machines

An Example – Drive-by Download



Talk Overview

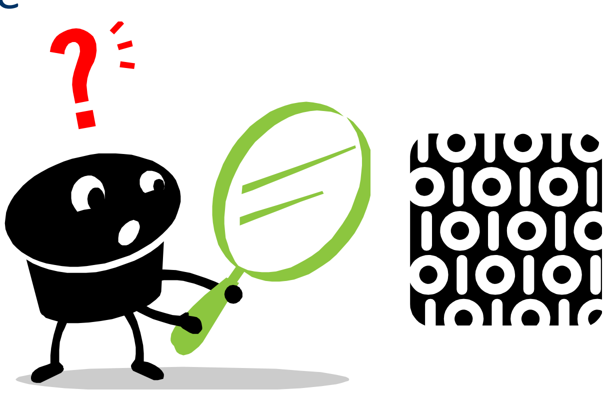
- Problem
- Solution
- Instruction Set Randomization
- Software Choices
  - Single-key
  - Page-mode
- Hardware Choices
  - Simulation Results
  - Implementation
- Security Analysis
- Conclusion

Countering Homogeneity

- Diversify!
  - No two systems should be alike
  - Not without problems
- Force attacks to be tailor-made
  - Low return-of-investment for development
  - No longer as lucrative

Instruction Set Randomization

- Each machine has a different ISA
  - Code for one doesn't work on another
  - Even better if this ISA can be kept secret
- Prevents *all* code-injection attacks
  - If malicious "code" inserted into application not in local ISA, it cannot execute
  - Unauthorized generic binary cannot execute



How to ISR'ize?

- Problematic. Cannot really have processors with unique ISA's
- One solution: Encrypt with random keys of fixed width
  - With 128-bit keys, large enough key-space (for now)
  - Should not disrupt established system standards (*e.g.*, virtual memory subsystem, caches)

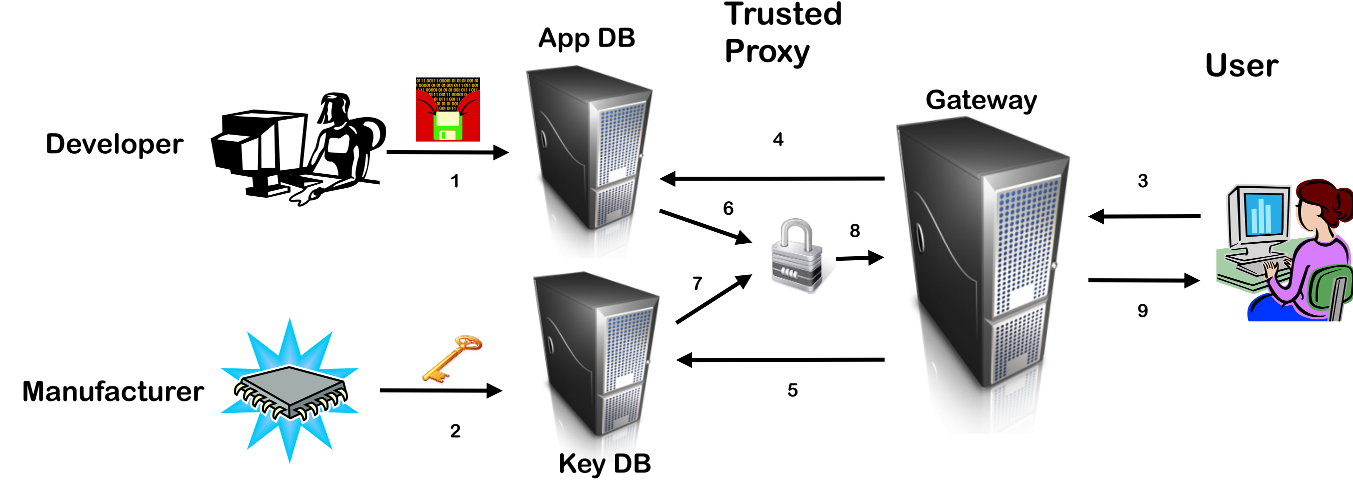
Software Design Options - Encryption

- Ideally ISR should be always on
- How strong an encryption to allow?
  - Classic performance/functionality trade-off
- Weak encryption
  - XOR, transposition
  - Low overhead but also low security
  - Reason why earlier software-based implementations didn't take off
- Strong encryption
  - AES, RSA
  - High security, but also high overheads

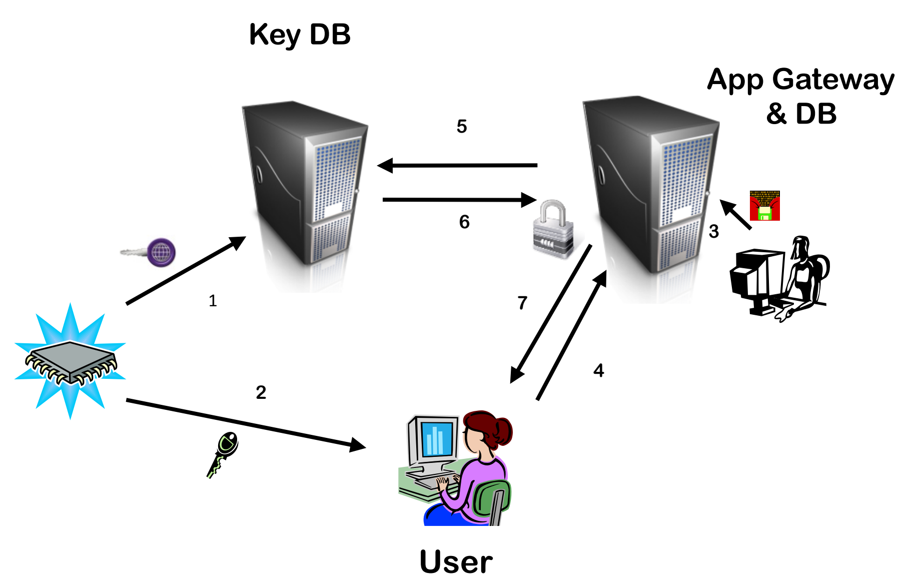
Software Design Options – Key Usage Granularity

- Per system
- Per privilege level
- Per process
- Per file with executable code
- Per (executable) memory page
- Others (network, functions, instructions, *etc.*)

Single-Key ISR – Ecosystem



Page-Mode ISR - Ecosystem

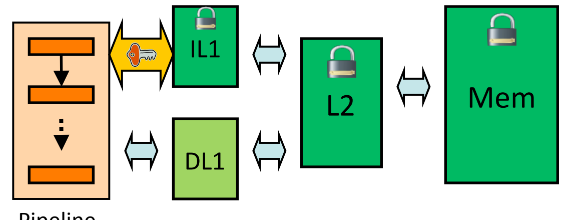


Hardware Design Options

- Already ruled out unique native ISAs

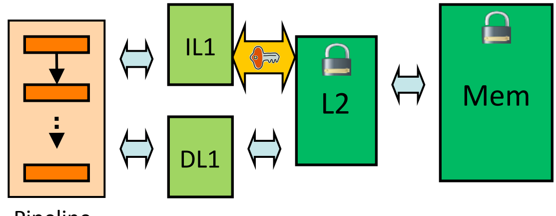
Hardware Design Options

- Already ruled out unique native ISAs
- Decrypt just before/within pipeline



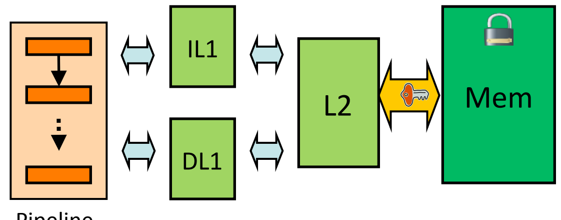
Hardware Design Options

- Already ruled out unique native ISAs
- Decrypt just before/within pipeline
- Decrypt at L1-I and L2 interface



Hardware Design Options

- Already ruled out unique native ISAs
- Decrypt just before/within pipeline
- Decrypt at L1-I and L2 interface
- Decrypt at L2 and memory interface



Caveats with L2/memory Decryption

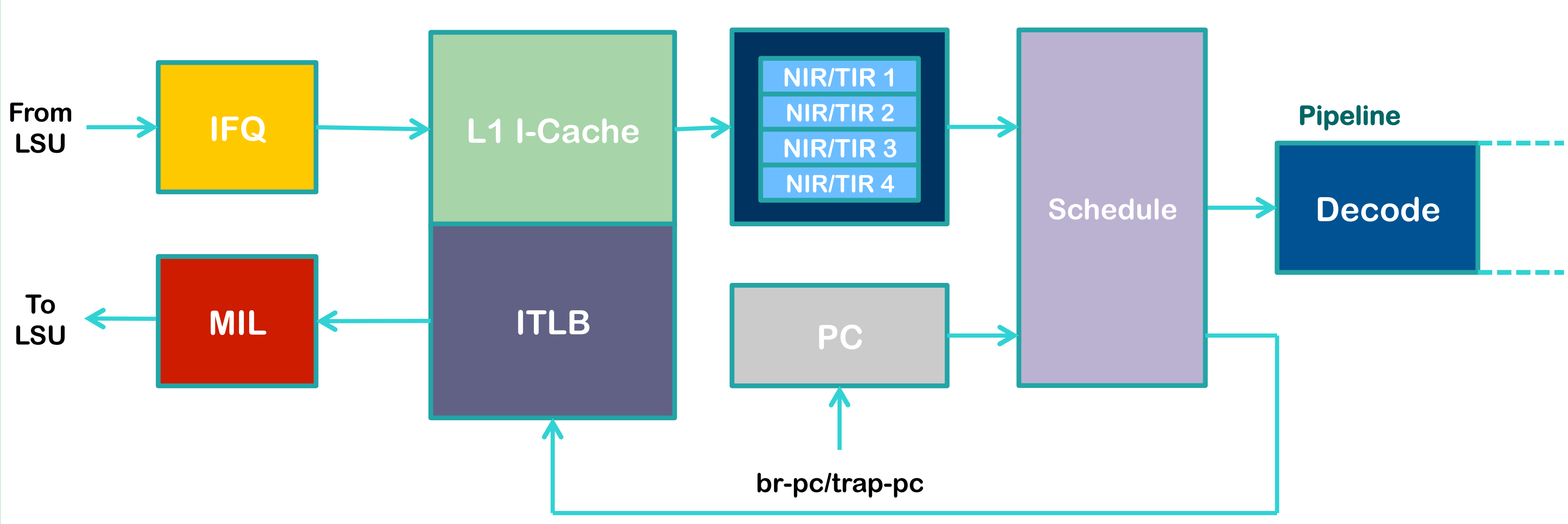
- How do we decrypt instructions exclusively?
  - Tag data vs. instruction
  - Track miss requests to figure if cache fill is D/I
- Not enough
  - Disallow cross-D/I cache fills above
  - Why?

Simulations show…

- Benchmarks from SPEC CPU 2k6, gem5 SPARC/Linux architecture
- 120 cycle latency for decryption

Benchmark	Decode	L1/L2	L2/Memory
bzip2	4292.346	0.003	0.003
gobmk	2624.427	11.413	0.167
hmmer	4539.352	0.004	0.003
mcf	2278.410	0.001	0.001
namd	4773.104	1.039	0.0
libquantum	5393.031	0.014	0.012
lbm	1626.031	0.0	0.0
Average	3646.732	1.782	0.026

Implementation - OpenSPARC



Implementation Simplicity

- Minimal changes to hardware code
  - For single-key, <5 lines
  - For page-mode, ~500 lines
- Requires software support too
  - But software design relatively inexpensive

Security Analysis - Protections

- Code Injection Attacks
- BIOS/Boot Protection
- Code Obfuscation

Security Analysis - Vulnerabilities

- Does not guarantee integrity
  - Replay and splicing attacks
  - Can be integrated with measures which do
- Still vulnerable to data-driven attacks
- Page-mode ISR requires trusted OS

To conclude…

- Homogeneity bad, diversity good
- ISR provides scalable and non-disruptive diversifying opportunity
- Developed hardware ISR support for the first time
  - Very simple implementation with negligible overheads

Questions?

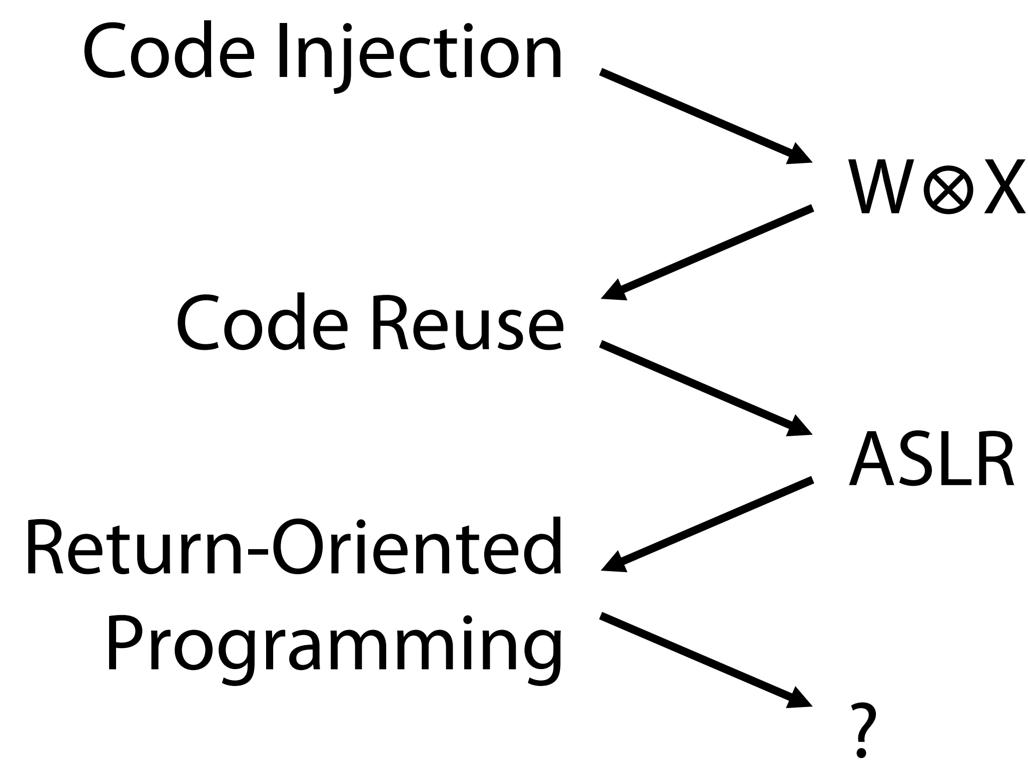
Defending against Return-Oriented Programming

Vasilis Pappas, Michalis Polychronakis, Angelos Keromytis

Columbia University

SPARCHS meeting – September 6, 2012

(Machine Code) Attacks and Defenses



ASLR is not Fully Supported

Executable programs in Ubuntu Linux

Only 66 out of 1,298 binaries in /usr/bin [SAB11]

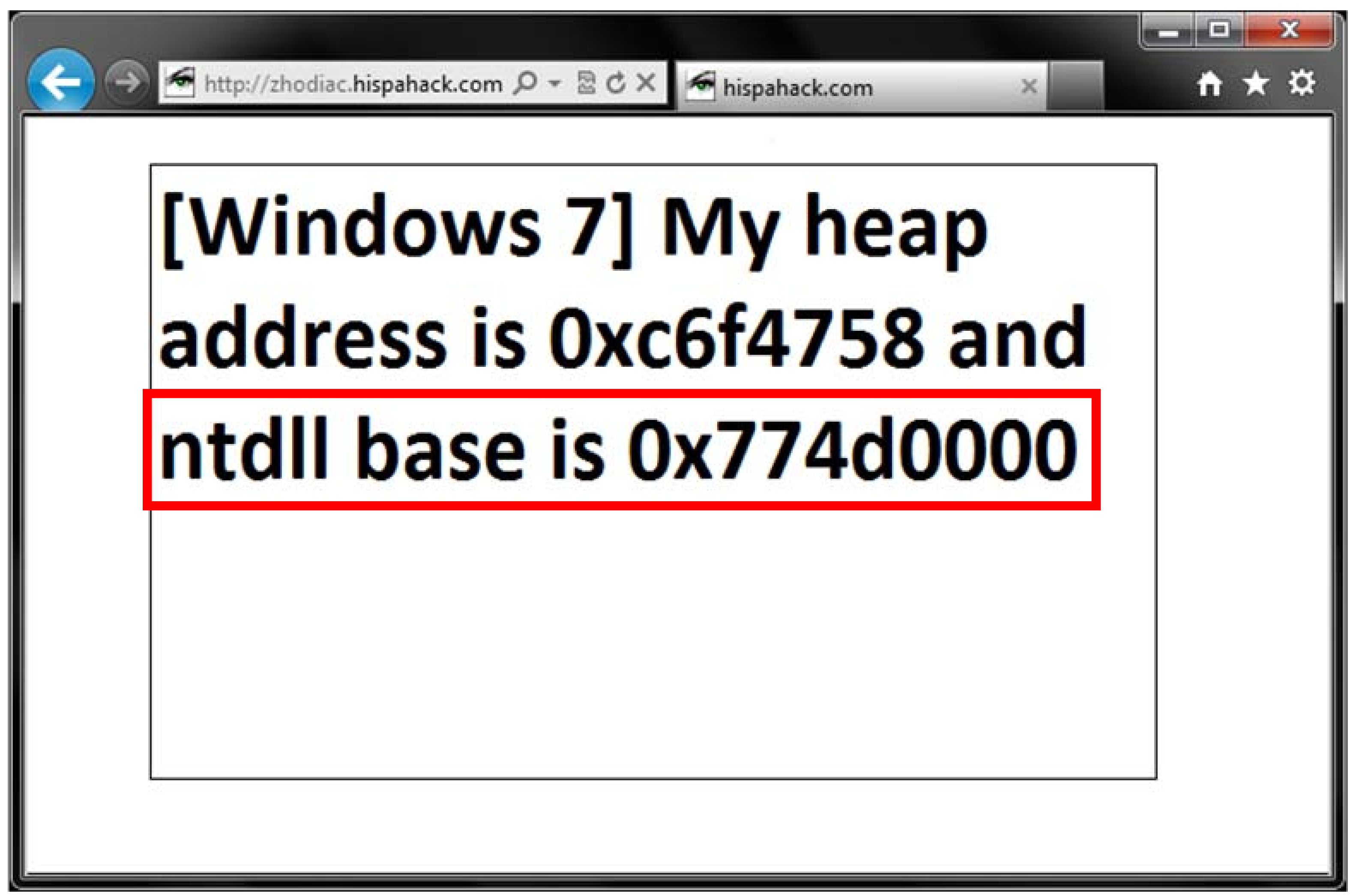
Popular third-party Windows applications

Only 2 out of 16 [Pop10]

Even applications that enable ASLR sometimes have statically mapped DLLs

EMET forced randomization

Information Leaks Break ASLR [Ser12]



Outline

Background

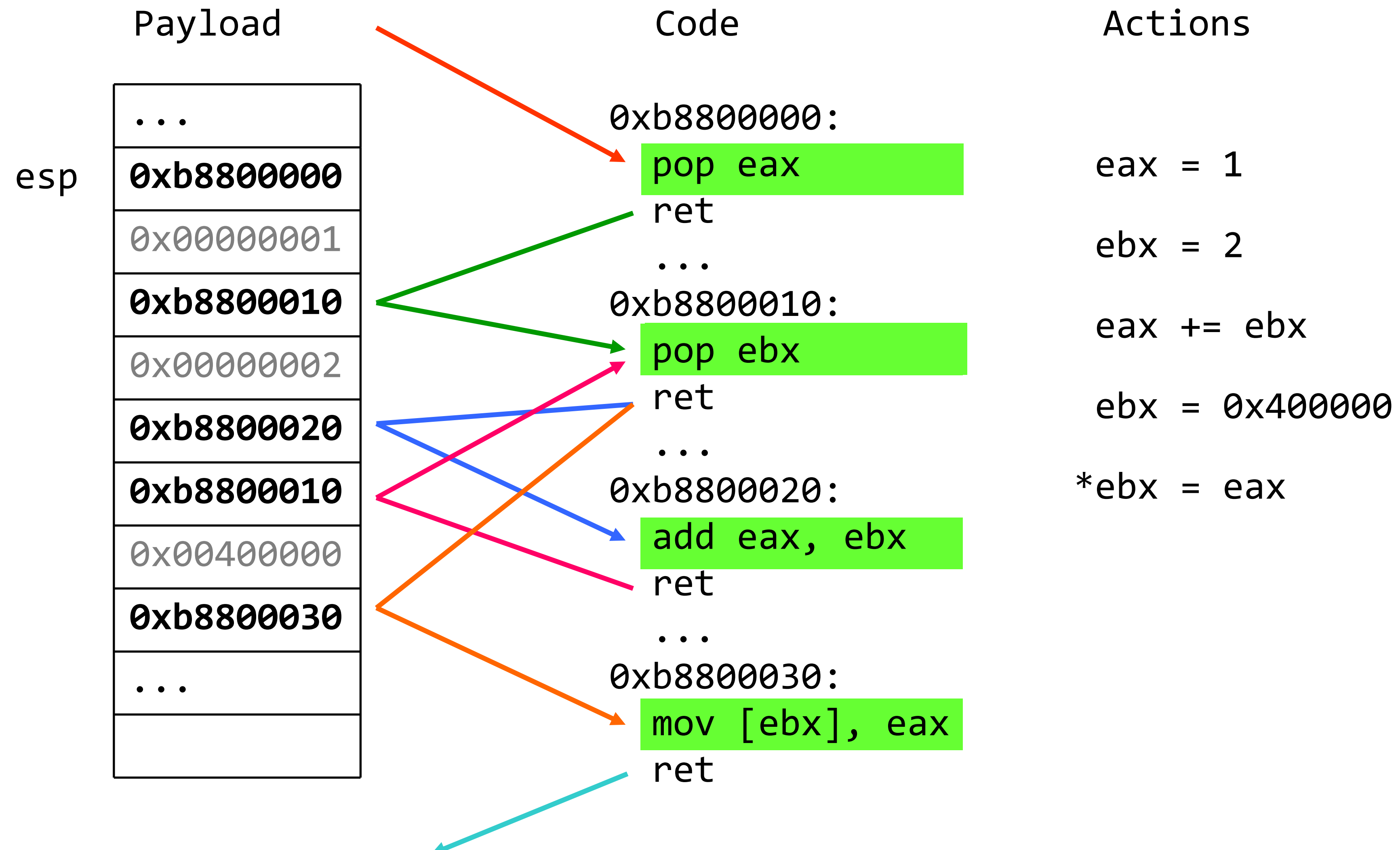
In-place code randomization

  IEEE Security & Privacy 2012

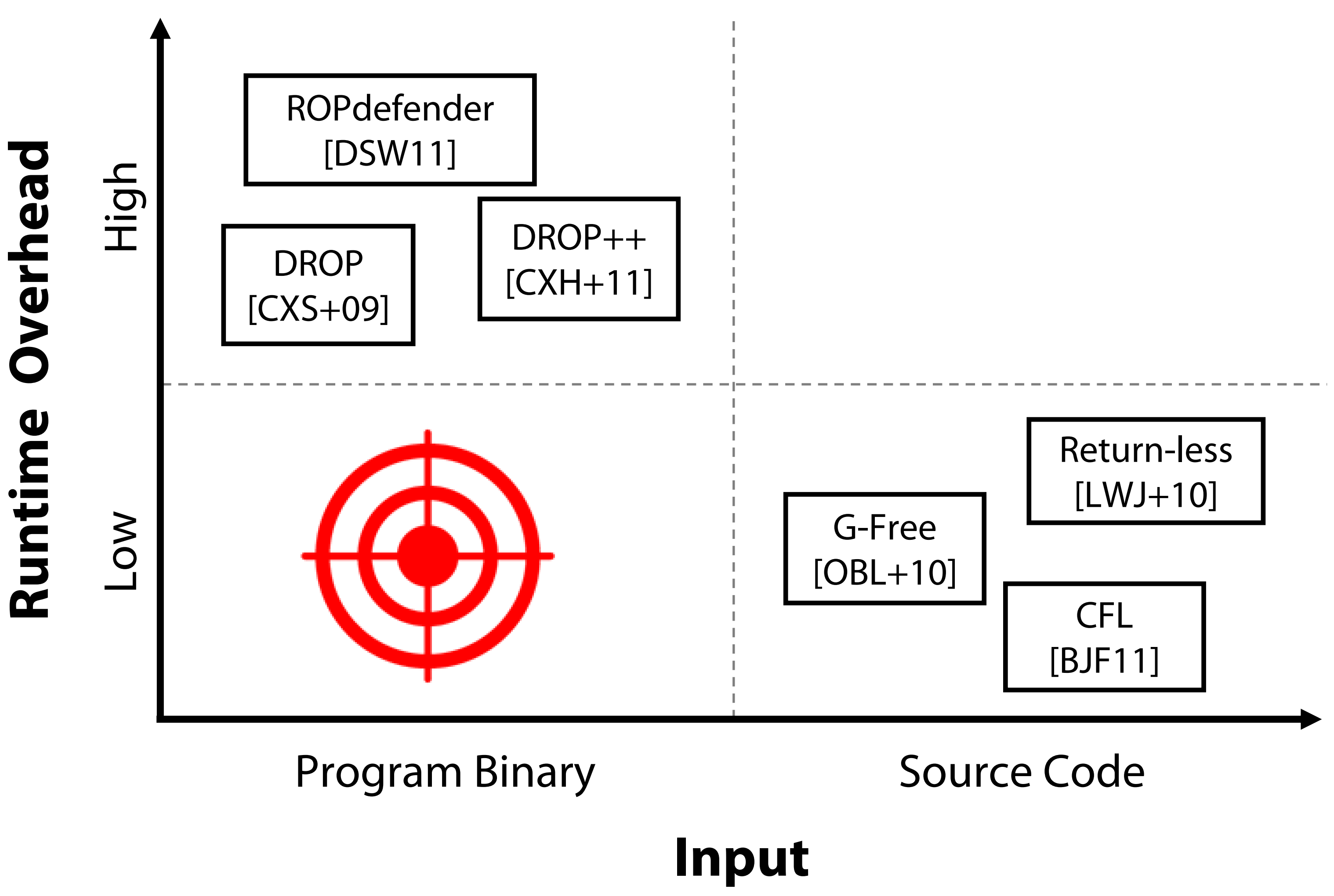
kBouncer

  Microsoft BlueHat Prize v1.0 winner!

Future directions


Payload
Code
Actions
...
esp
0xb8800000
0x00000001
0xb8800010
0x00000002
0xb8800020
0xb8800010
0x00400000
0xb8800030
...
0xb8800000:
pop eax
ret
...
0xb8800010:
pop ebx
ret
...
0xb8800020:
add eax, ebx
ret
...
0xb8800030:
mov [ebx], eax
ret
eax = 1
ebx = 2
eax += ebx
ebx = 0x400000
*ebx = eax

ROP Defenses



In-Place Code Randomization

Software diversification

Applicable on third-party applications

Zero (non-measurable) performance overhead

Why In-Place?

Randomization usually changes the code size

- Need to update the control-flow graph (CFG)

Accurate disassembly of stripped binaries is hard

- Incomplete CFG (data vs. code)
- Code resize not an option

Must randomize in-place!

Code Transformations

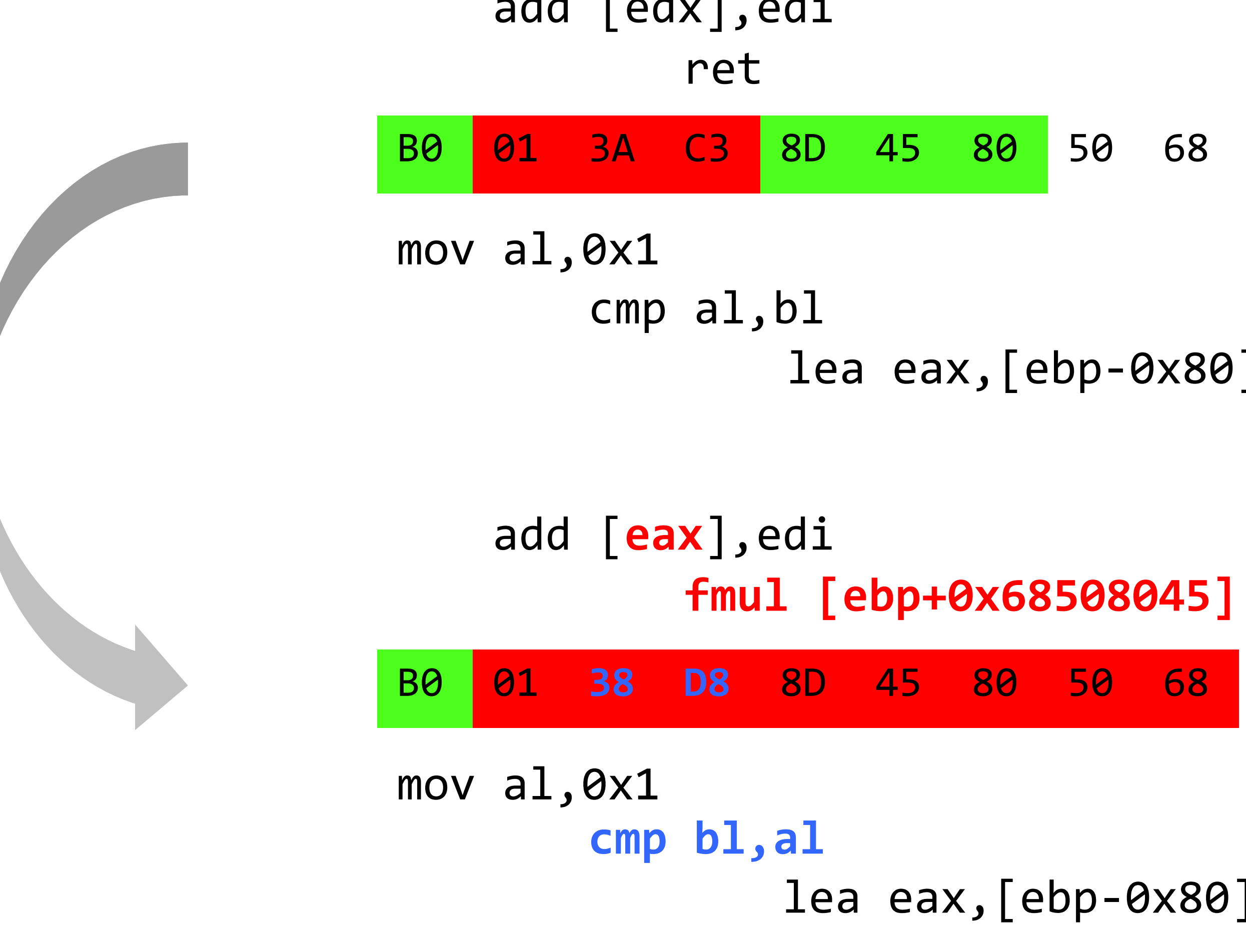
Instruction Substitution

Instruction Reordering

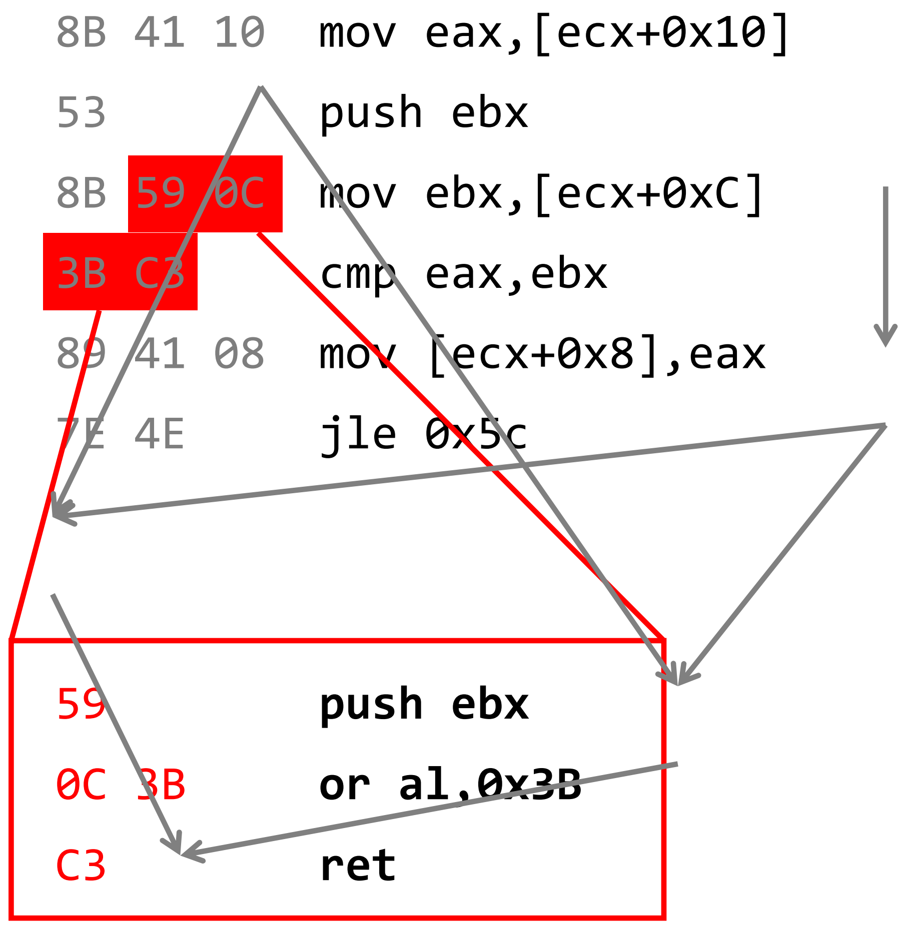
- Intra Basic Block
- Register Preservation Code

Register Reassignment

Instruction Substitution



Instruction Reordering (Intra Basic Block)

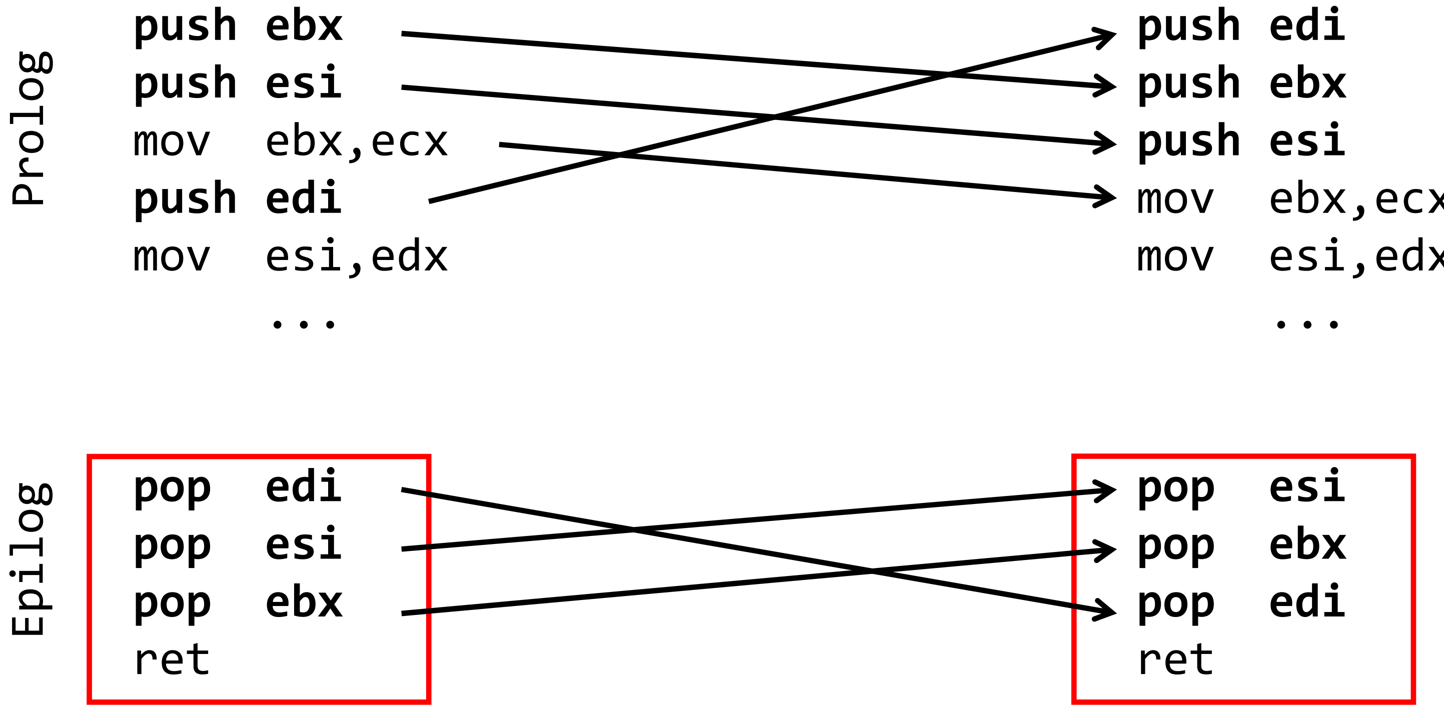


Instruction Reordering (Intra Basic Block)

```
8B 41 10   mov eax,[ecx+0x10]
53         push ebx
8B 59 0C   mov ebx,[ecx+0xC]
3B C3      cmp eax,ebx
89 41 08   mov [ecx+0x8],eax
7E 4E      jle 0x5c
```

```
41         inc ecx
10 89 41 08 3B C3
  adc [ecx-0x3CC4F7BF],cl
```

Register Preservation Code Reordering



Register Reassignment

Live regions

eax edi

```
function:
 push esi
 push edi
 mov  edi,[ebp+0x8]
 mov  eax,[edi+0x14]
 test eax,eax
 jz   0x4A80640B
 mov  ebx,[ebp+0x10]
 push ebx
 lea  ecx,[ebp-0x4]
 push ecx
 push edi
 call eax
 ...
```

```
function:
 push esi
 push edi
 mov  eax,[ebp+0x8]
 mov  edi,[edi+0x14]
 test edi,edi
 jz   0x4A80640B
 mov  ebx,[ebp+0x10]
 push ebx
 lea  ecx,[ebp-0x4]
 push ecx
 push eax
 call edi
 ...
```

Implementation: Orp

Focused on the Windows platform

- Could be integrated in Microsoft's EMET

CFG extraction using IDA Pro

- Implicitly used registers
- Liveness analysis (intra and inter-function)
- Register categorization (arg., preserved, …)
- Randomization
- Binary rewriting (relocations fixing, …)

Evaluation

Correctness and performance

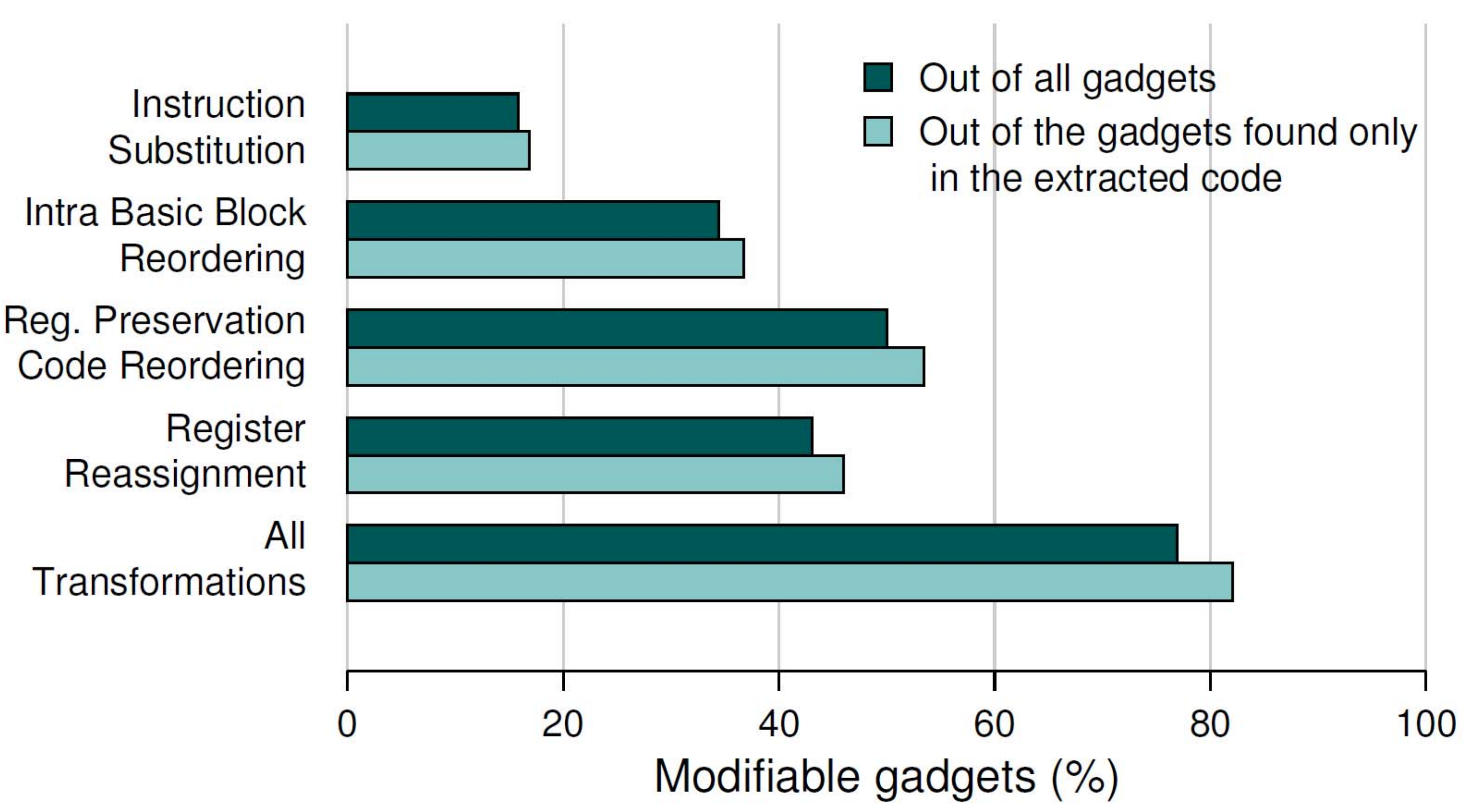
Used Wine's extensive test suite with randomized versions of Windows DLLs

Randomization Coverage

Effectiveness against real-world exploits

Robustness against ROP Compilers

Randomization Coverage



Dataset: 5,235 PE files (~0.5GB code) from Windows, Firefox, iTunes, Reader

Real-World Exploits

Exploit/Reusable Payload	Unique Gadgets	Modifiable	Combinations
Adobe Reader v9.3.4	**11**	**6**	287
Integard Pro v2.2.0	**16**	**10**	322K
Mplayer Lite r33064	**18**	**7**	1.1M
msvcr71.dll (While Phosphorus)	**14**	**9**	3.3M
msvcr71.dll (Corelan)	**16**	**8**	1.7M
mscorie.dll (White Phosphorus)	**10**	**4**	25K
mfc71u.dll (Corelan)	**11**	**6**	170K

Modifiable gadgets were not always directly replaceable!

ROP Compilers

Is it possible to create a randomization-resistant ROP payload?

Using only the remaining non-randomized gadgets

Tested two ROP payload construction tools

mona.py: constructs DEP+ASLR bypassing code

Allocate a WX buffer, copy shellcode, and jump to it

Q: state-of-the-art ROP compiler [SAB11]

Designed to be robust against small gadget sets

ROP Compiler Results

Non-ASLR Code Base	Mona		Q	
	Original	Rand.	Original	Rand.
Adobe Reader v9.3.4	✓	✗	✓	✗
Integard Pro v2.2.0	✗	✗	✓	✗
Mplayer Lite r33064	✓	✗	✓	✗
msvcr71.dll	✗	✗	✓	✗
mscorie.dll	✗	✗	✗	✗
mfc71u.dll	✓	✗	✓	✗

Both tools failed to construct ROP payloads using non-randomized code!

kBouncer

Partial control-flow integrity against ROP

Transparent

- Applicable on third-party applications
- Compatible with code signing, self-modifying code, JIT, ...

Lightweight

- Less than 5% runtime overhead

Effective

- Prevents real-world exploits

ROP disrupts the regular call path pattern

Legitimate code:
`ret` transfers control to the instruction right after the corresponding `call` ➔ legitimate call sites

ROP code:
`ret` transfers control to the first instruction of the next gadget ➔ arbitrary locations

Main idea:
Runtime monitoring of `ret` instructions' targets

Last Branch Record (LBR)

Introduced in the Intel Nehalem architecture

Stores the last 16 executed branches in a set of model-specific registers (MSR)

- Can filter certain types of branches (relative/indirect calls/jumps, returns, ...)

Multiple advantages

- Zero overhead for recording the branches
- Fully transparent to the running application
- Does not require source code or debug symbols
- Can be dynamically enabled for any running application

Monitoring Granularity

Non-zero overhead for reading the LBR cache (accessible only from kernel level)

Lower frequency ➔ lower overhead

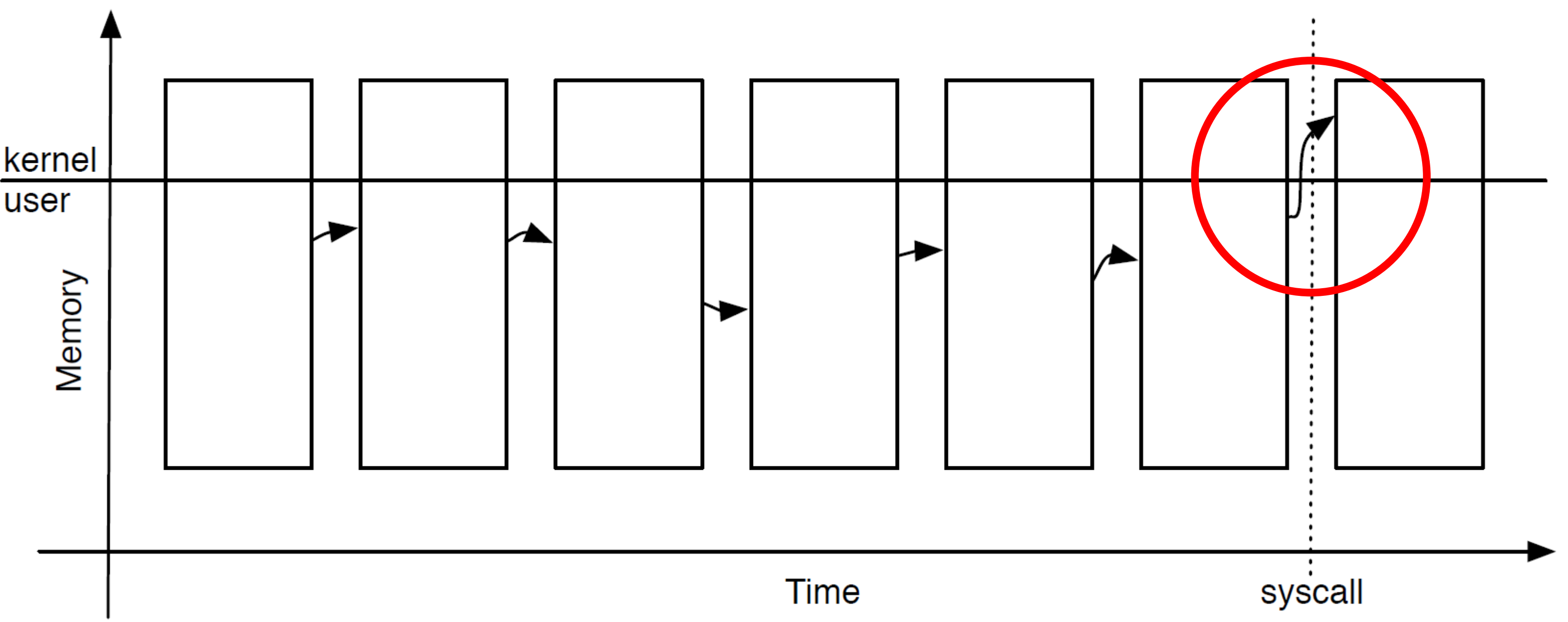
ROP code can run at any point

Higher frequency ➔ higher accuracy

Monitoring Granularity

Meaningful ROP code will eventually interact with the OS through system calls

Check for abnormal control transfers on system call entry

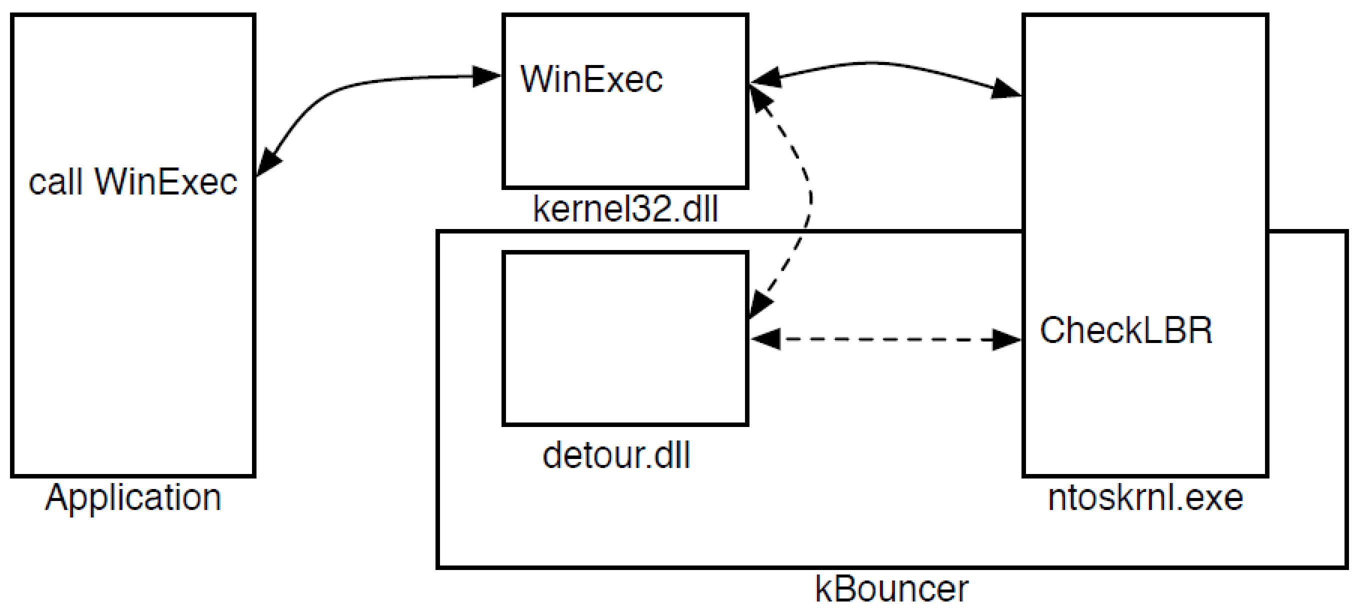


Implementation

Working prototype for Windows 7 x64 SP1

API interception using Detours for PatchGuard compatibility

Uses only the Windows SDK and DDK (no third-party code)



Runtime Overhead

Application	real/usr/sys time	# Call/Ret	# Syscall	False Positives	Overhead ms (%)
WM Player	30.73/0.27/0.21	30.8M	194K	0	33.8 (6%)
Internet Explorer	7.24/0.06/0.04	1.5M	34K	0	5.4 (5%)
Adobe Reader	4.11/1.32/0.24	35.3M	107K	0	18.7 (1%)

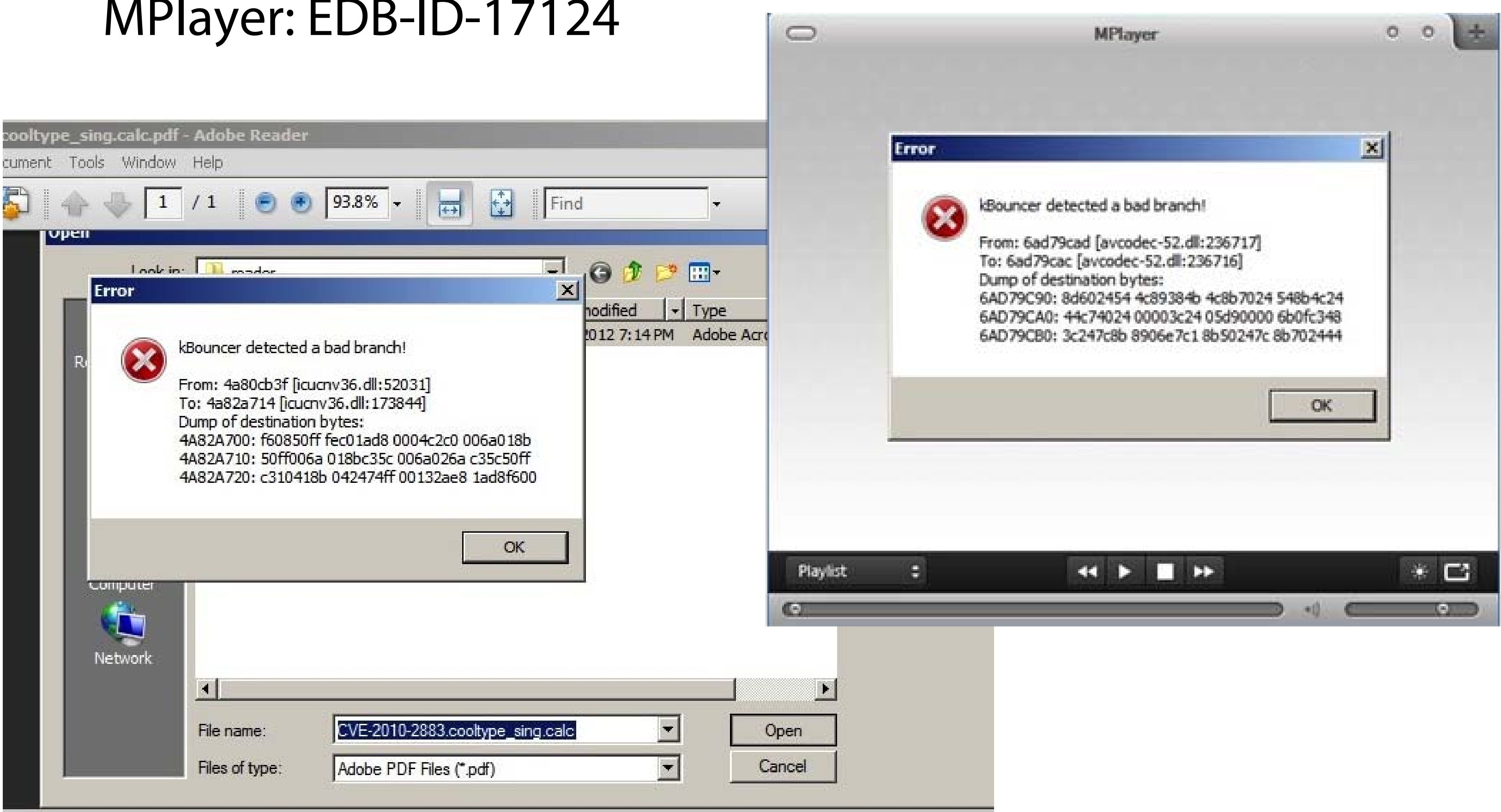
Low overhead (1-6%) even when checking all syscalls

No false positives

Effectiveness

Successfully prevented two real-world exploits

- Adobe Reader: CVE-2010-2883
- MPlayer: EDB-ID-17124



BlueHat Prize
$200,000 USD
$200K

Future Directions

The Limited LBR size (16) might allow for evasion

- Invoke syscall through a path of legitimate branches
- Seems hard, but might be possible...

ROP without returns (JOP)

- Characteristic runtime pattern (dispatcher gadget)
- Could be detected by enabling tracking of all indirect branches
- More pressure on the LBR cache...

What would be an ideal LBR size?

Other hardware features that could help?

Function Call Return Value Profiling

Build profiles of benign program behavior for anomaly detection

Modeling based on a small window of previous function calls and their return values [LSC+08]

Explore the use of LBR or other hardware features for runtime checking

Combining control and data flow tracking

Build models of expected behavior based on memory footprints

- Causality of data inputs and generated outputs
- Lifetime and interactions of program-specific objects
- Accessed memory locations

Control + data flow information

Prototyping using Libdft (Pin-based DFT)

Explore optimizations based on hardware features

REASSURE

Enables software self-healing using rescue points

- Rescue points reuse valid error codes returned by functions to handle unforeseen errors

Handles NULL pointer dereference bugs

Transforms fail-stop protection mechanisms to fail once

- Generate a rescue point definition after observing an error the first time

Self-contained

Future Work on REASSURE

Self-healing kernels

Challenge: achieve low performance overhead

Our approach: Hardware assisted self-healing

Use hardware transactional memory (HTM)

- Provide checkpoint/rollback
- Handle concurrency efficiently

Software transactional self-healing prototype

Summary

Return-Oriented Programming is increasingly used in real-world exploits

In-place code randomization and branch target monitoring prevent real exploits

Focus on hardware-assisted runtime detection and protection mechanisms

In-place code randomization prototype (Python)
http://nsl.cs.columbia.edu/projects/orp

References

[Ser12] Fermin J. Serna. The case of the perfect info leak, 2012.
http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[SAB11] Edward J. Schwartz et al. Q: exploit hardening made easy. USENIX Security, 2011.

[Pop10] Alin Rad Pop. Dep/aslr implementation progress in popular third-party windows applications, 2010.
http: //secunia.com/gfx/pdf/DEP_ASLR_2010_paper.pdf.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). CCS, 2007.

[CDD+10] Stephen Checkoway et al. Return-oriented programming without returns. CCS, 2010

[BJFL11] Tyler Bletsch et al. Jump-oriented programming: a new class of code-reuse attack. ASIACCS, 2011.

[LZWG11b] Kangjie Lu et al. Packed, printable, and polymorphic return-oriented programming, RAID, 2011.

[DSW11] Lucas Davi et al. Ropdefender: a detection tool to defend against return-oriented programming attacks. ASIACCS, 2011

[CXS+09] Ping Chen et al. Drop: Detecting return-oriented programming malicious code, ICISS, 2009.

[CXH+11] Ping Chen et al. Efficient detection of the return-oriented programming malicious code, ICISS, 2011.

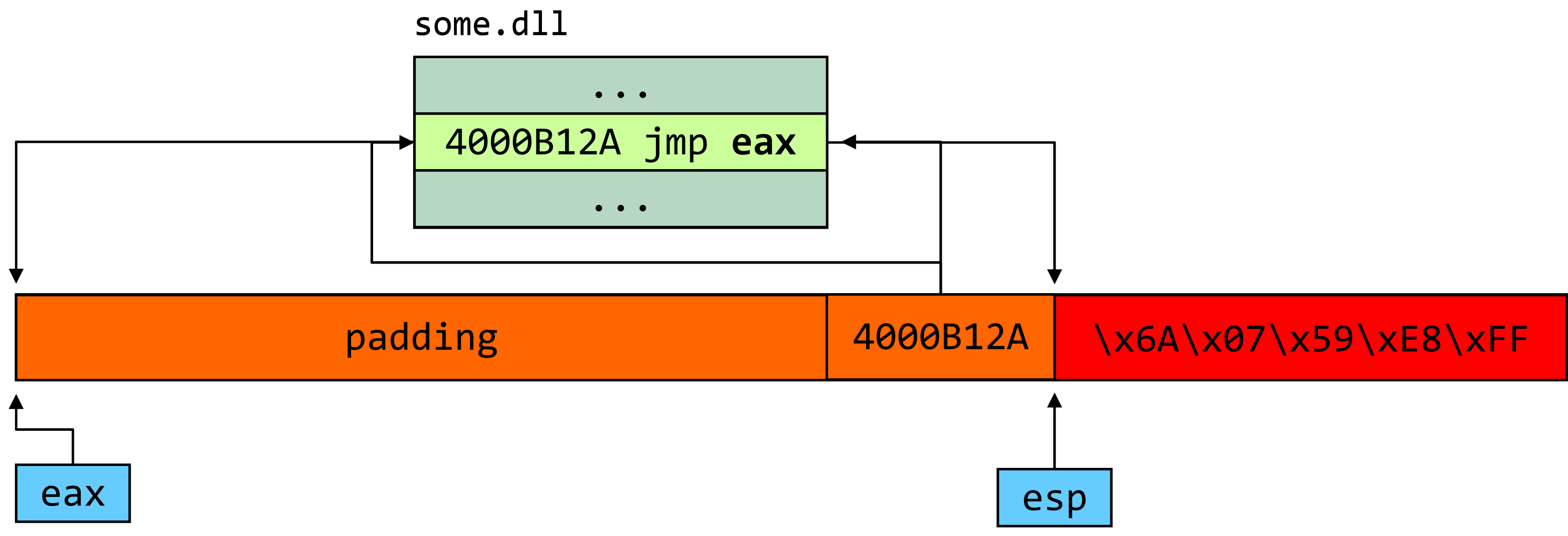
[OBL+10] Kaan Onarlioglu et al. G-free: defeating return-oriented programming through gadget-less binaries. ACSAC, 2010.

[LWJ+10] Jinku Li et al. Defeating return-oriented rootkits with “return-less” kernels. EuroSys, 2010.

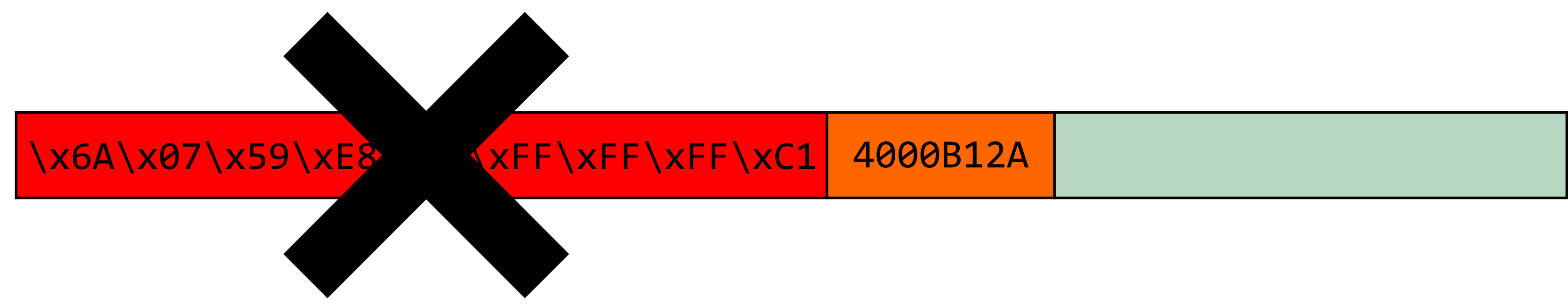
[BJF11] Tyler Bletsch et al. Mitigating code-reuse attacks with control-flow locking. ACSAC, 2011.

[LSC+08] Michael E. Locasto et al. Return value predictability for self-healing. IWSEC 2008.

Code Injection



NX



W^X, PaX, Exec Shield, DEP

x86 support introduced by AMD, followed by Intel

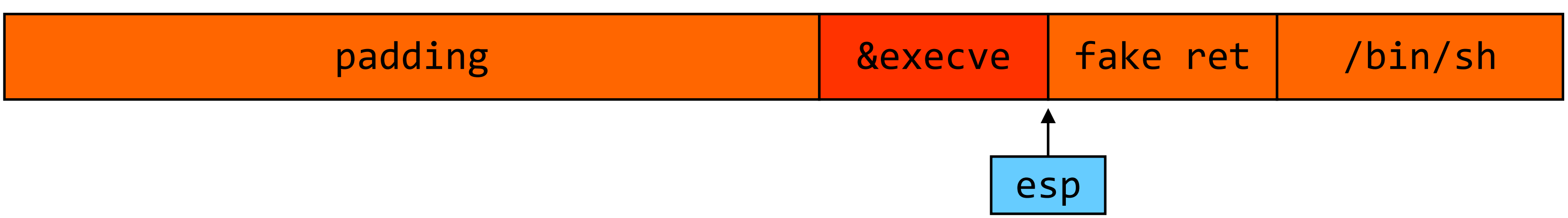
- Pentium 4 (late models)

DEP introduced in XP SP2 (hardware-only)

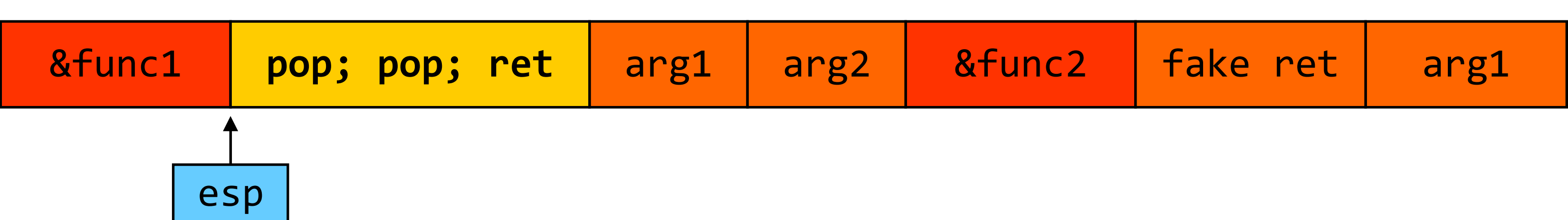
- Applications can opt-in (SetProcessDEPPolicy() or /NXCOMPAT)

Ret2libc → ROP

ret2libc [Solar Designer '97]



ret2libc chaining [Nergal '01]



Ret2libc → ROP

Borrowed code chunks technique [Krahmer '05]

Pass function arguments through registers (IA-64)

```
0x0000000000400a82:  pop %rbx
0x0000000000400a83:  retq

0x00002aaaaac743d5:  mov %rbx,%rax   → &system
0x00002aaaaac743d8:  add $0xe0,%rsp
0x00002aaaaac743df:  pop %rbx
0x00002aaaaac743e0:  retq

0x00002aaaaac50bf4:  mov %rsp,%rdi   → /bin/sh
0x00002aaaaac50bf7:  callq *%eax
```

Return-oriented programming [Shacham '07]

Turing-complete return-oriented "shellcode"

Jump-oriented programming [Shacham '10]

Current State of ROP exploits

First-stage ROP code for bypassing DEP

- Allocate/set W+X memory (`VirtualAlloc`, `VirtualProtect`, …)
- Copy embedded shellcode into the newly allocated area
- Execute!

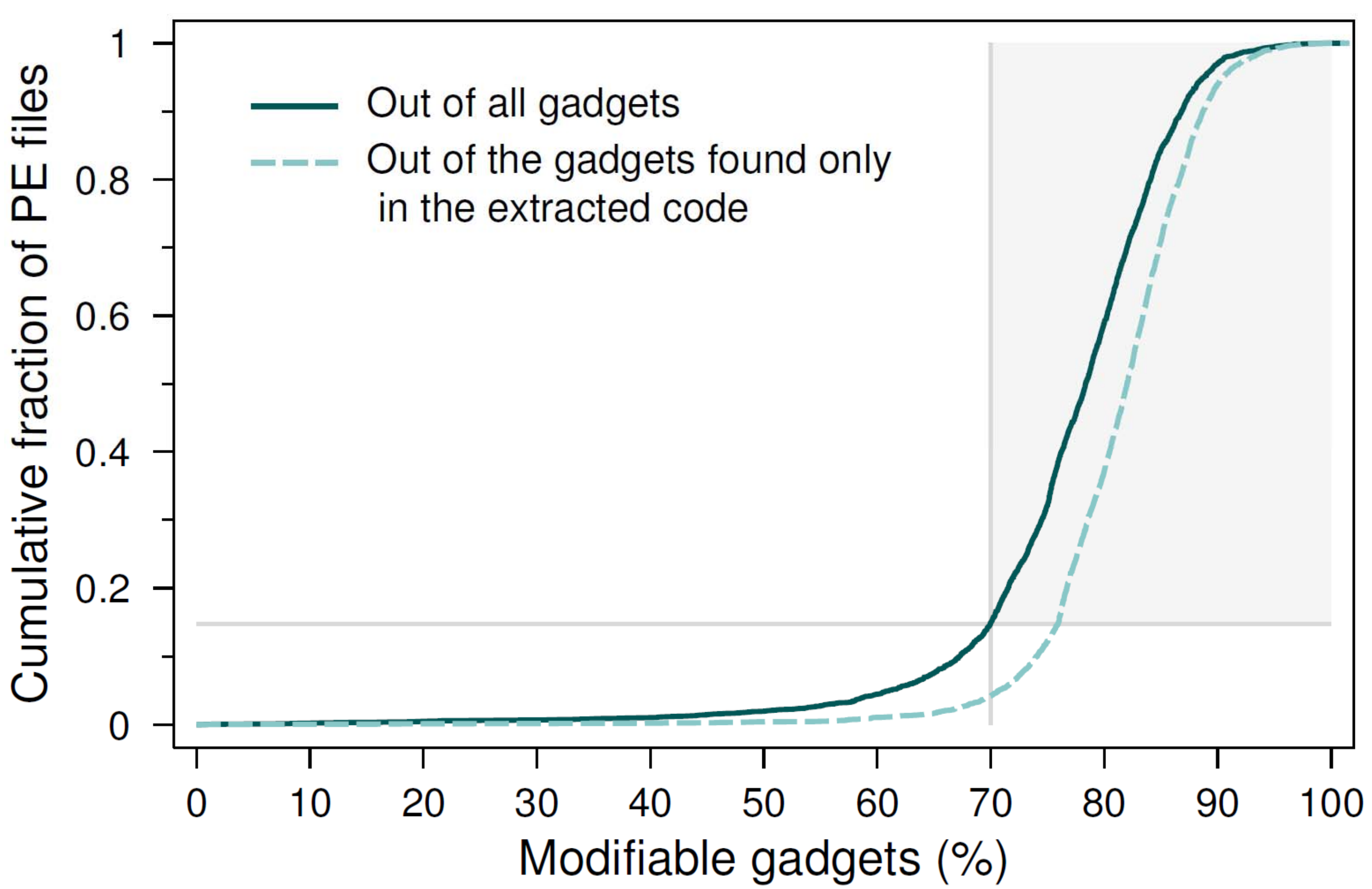
The complexity of ROP exploit code increases…

- New anti-ROP features in EMET
- ROP exploit mitigations in Windows 8

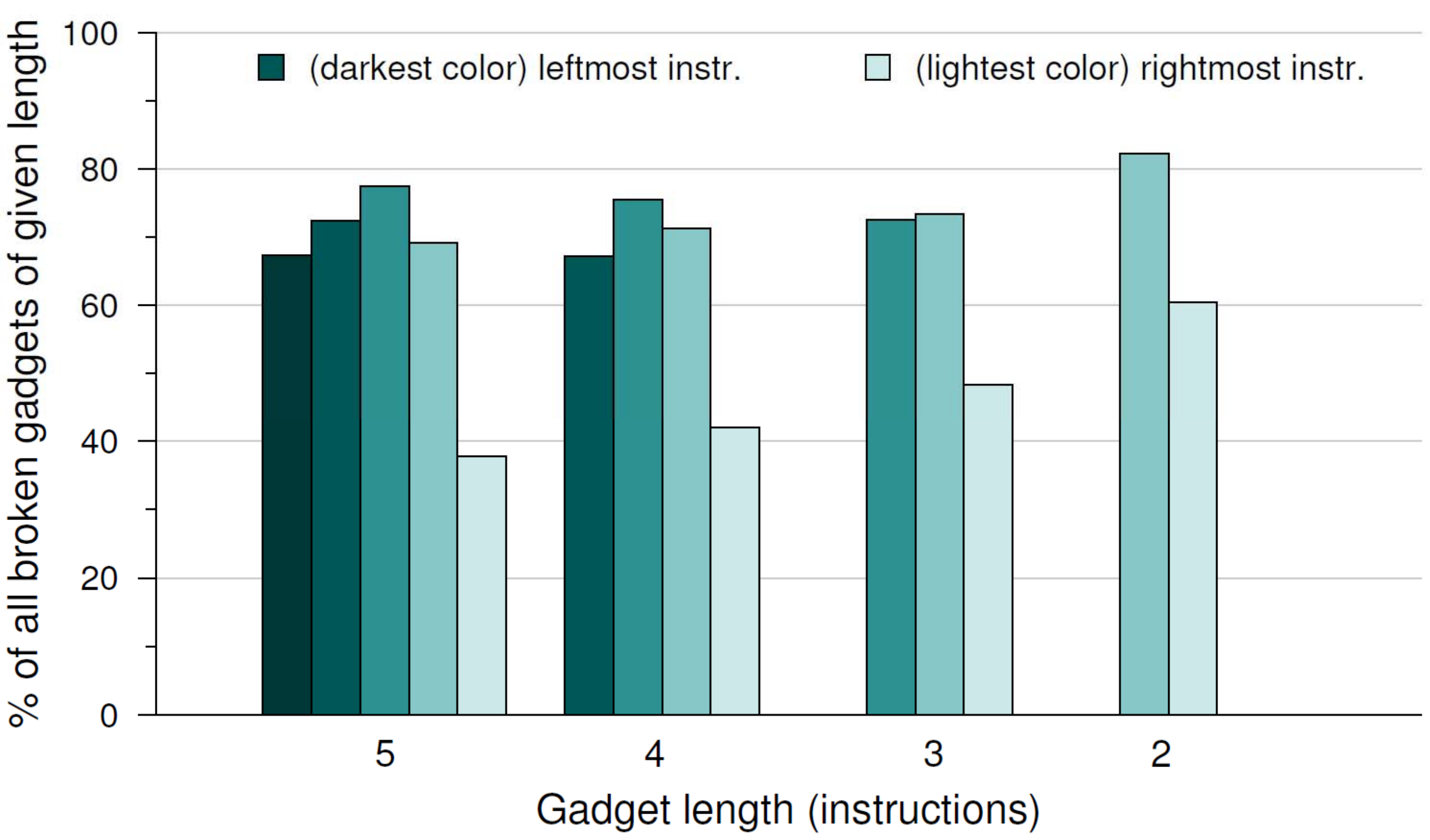
The embedded shellcode can be concealed

- ROP-based unpacker [Lu ’11]

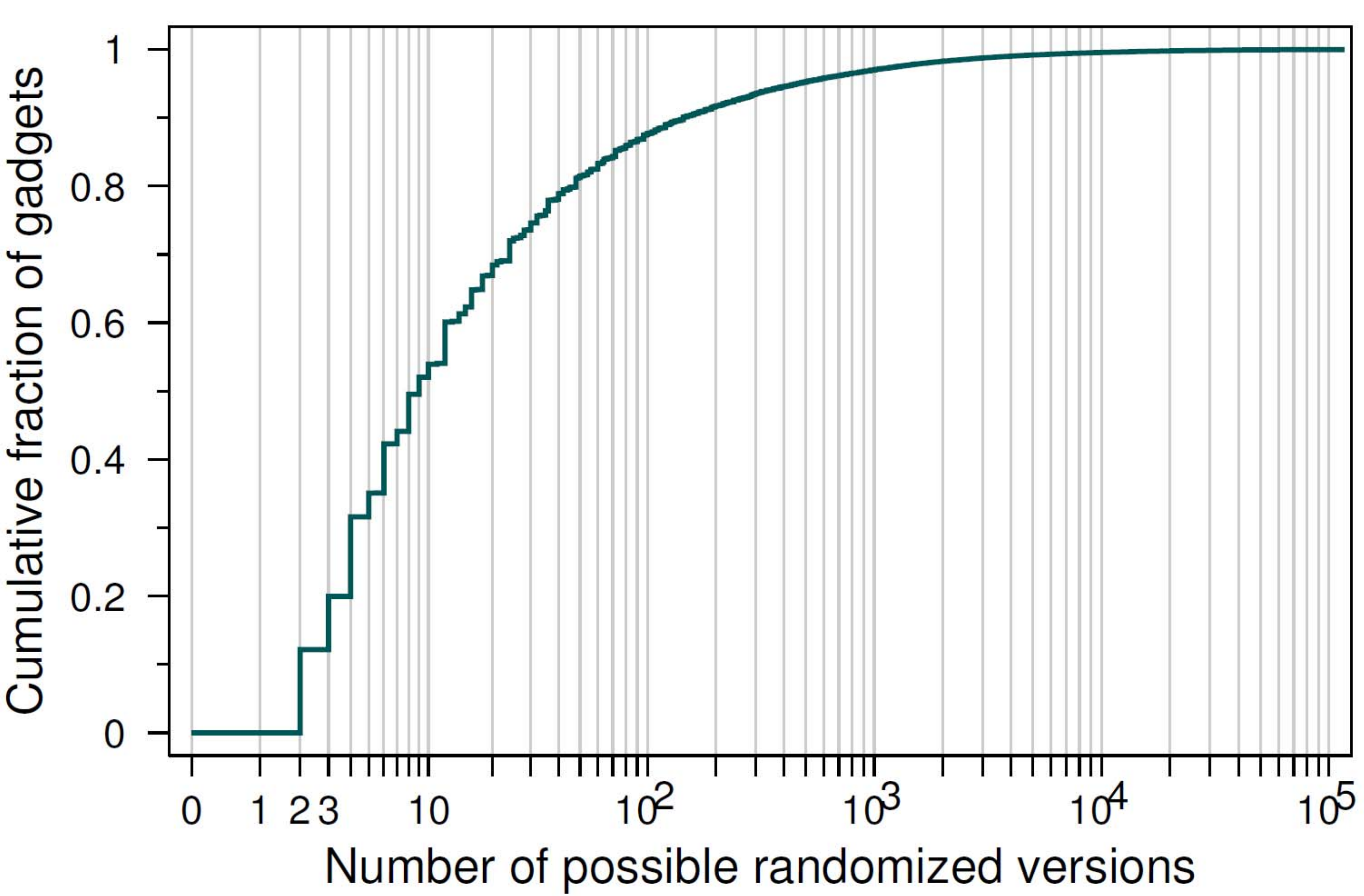
Modifiable Gadgets



Impact on Broken Gadgets' Instructions



Randomization Entropy for Broken Gadgets



WHISK: Dynamic Information Flow Tracking for Heterogeneous Systems

Joël Porquet and Simha Sethumadhavan

Columbia University – 9/6/2010

Outline

- Introduction
  - Dynamic Information Flow Tracking (DIFT)
  - Heterogeneous systems
- Related work
  - Tag management
- The WHISK architecture
- Implementation and (a few) results
- Conclusion

DIFT

- Since almost a decade, many hardware approaches
- Core principle
  - Taint data from untrusted sources
    - Extra tag bit per byte/word
  - Propagate taint during program execution
    - Operation on tainted data produces tainted result
  - Check spurious uses of tainted data
    - Code execution
- Detection of low-level up to high-level attacks (and to some extend information leakage)

DIFT - Example

- Simple buffer overflow attack

```
int function (char *fname)
{
  char buf[256];
  FILE *src;

  src = fopen(fname, “r”);

  while (fgets(buf, 1024, src)) {
    …
  }

  return 0;
}
```

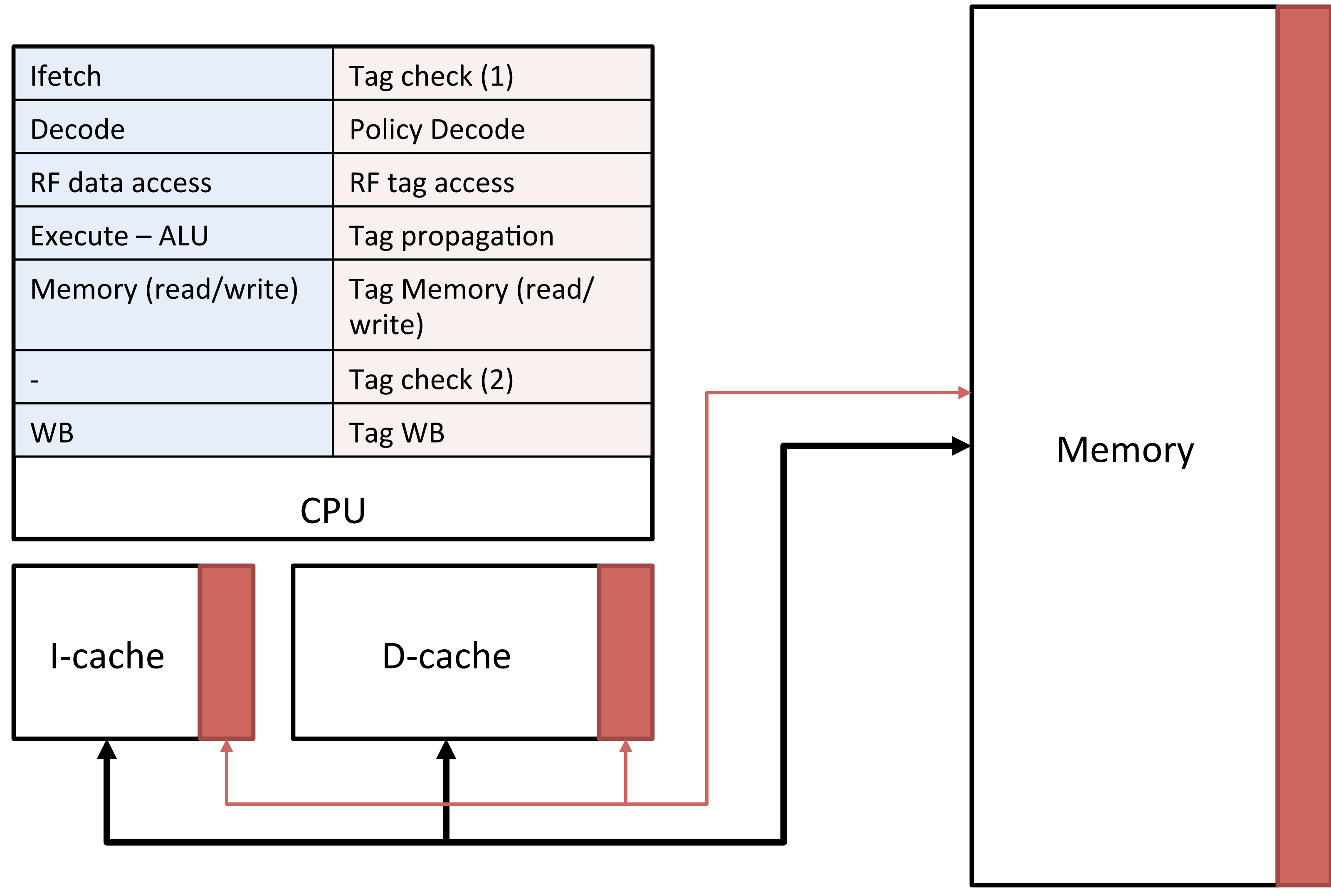
return address

buf [256]

attack

Malicious
input data

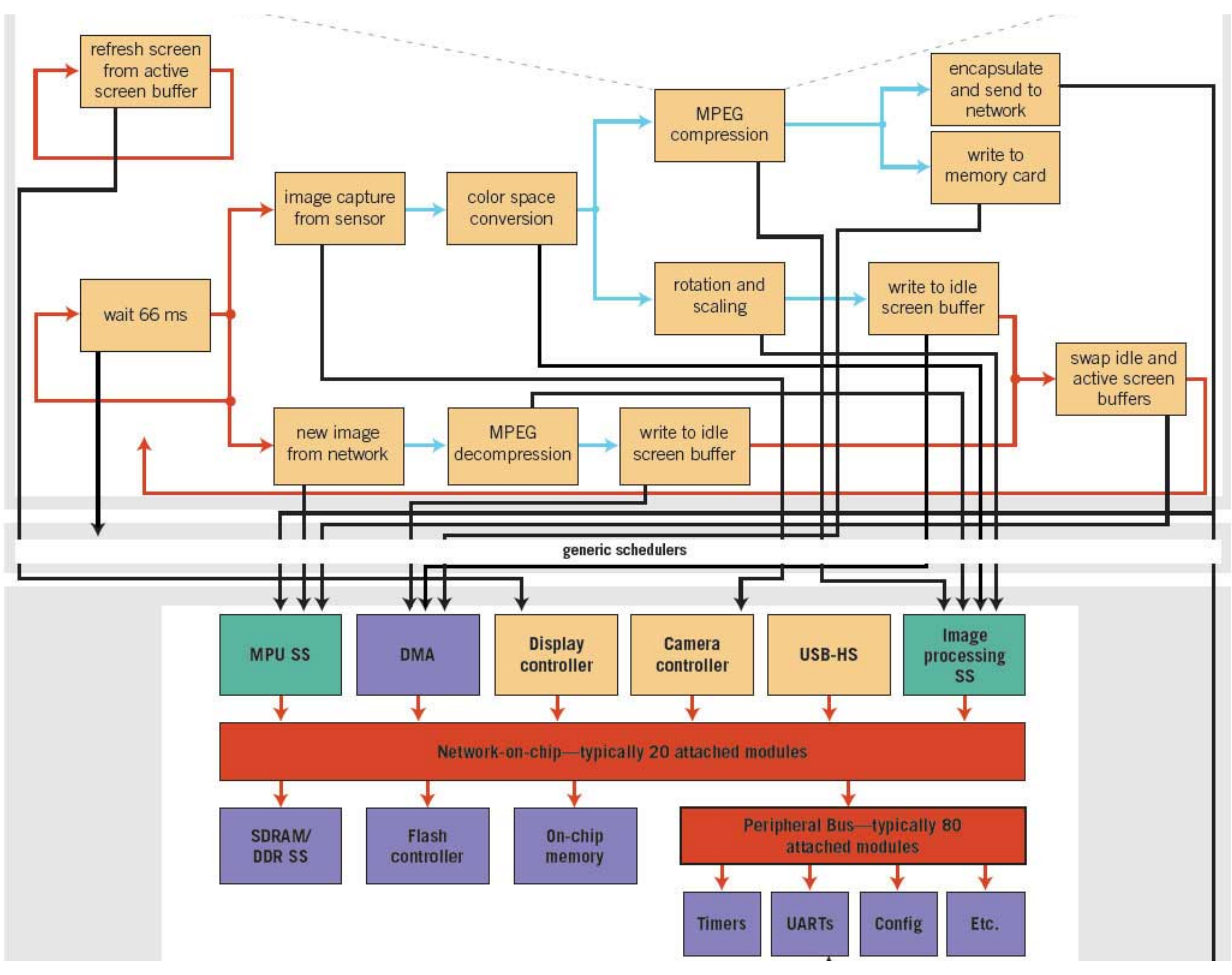
DIFT – conceptual implementation



Heterogeneous systems

- Embedded systems
  - Energy-efficiency concerns
  - Dedicated asymmetric processors
  - Accelerators

- Commodity systems
  - Performance concerns
  - GPGPU
  - Accelerators (crypto, etc.)


refresh screen from active screen buffer
image capture from sensor
color space conversion
MPEG compression
encapsulate and send to network
write to memory card
wait 66 ms
rotation and scaling
write to idle screen buffer
swap idle and active screen buffers
new image from network
MPEG decompression
write to idle screen buffer
generic schedulers
MPU SS
DMA
Display controller
Camera controller
USB-HS
Image processing SS
Network-on-chip—typically 20 attached modules
SDRAM/ DDR SS
Flash controller
On-chip memory
Peripheral Bus—typically 80 attached modules
Timers
UARTs
Config
Etc.

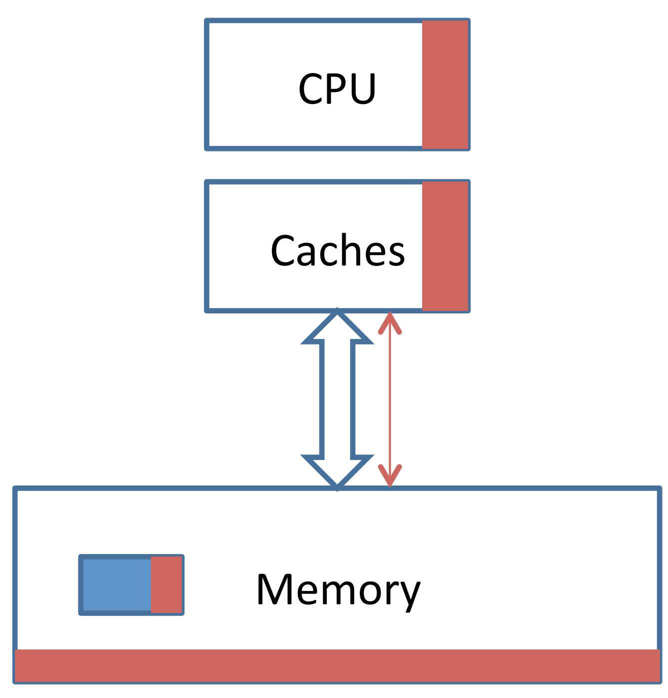
DIFT for heterogeneous systems

- Mainly peripheral devices so far
  - Sources or sinks, binary access control by software
- What about accelerators?
  - Memory to memory models
  - Should be integrated to the DIFT infrastructure

Related work – *Integrated* scheme

Data extension with tags

- Pros:
  - Low complexity
  - Consistency by default
  - Easy access for accelerators
- Cons:
  - Non-standard memory banks, special CPU instructions
  - High area overhead (wasteful in most cases)

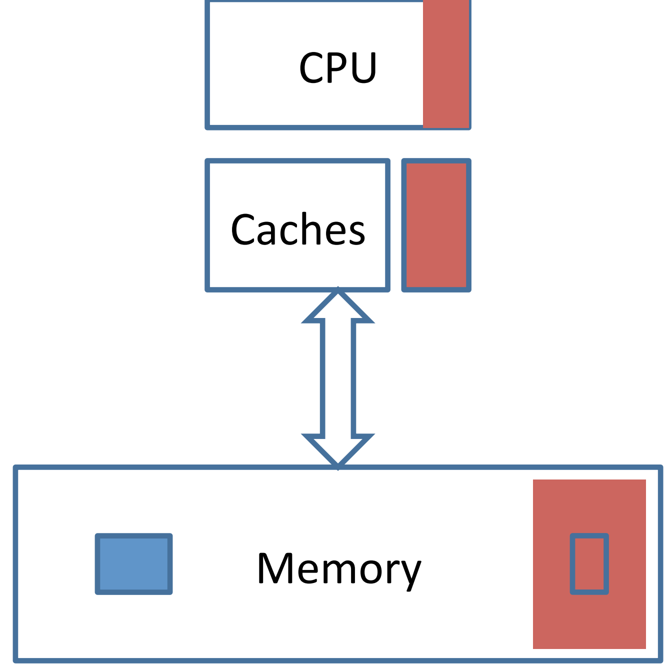


[Minos, Raksha]

Related work – *Decoupled* scheme

Separation of data and tags

- Pros:
  - Low area overhead
- Cons:
  - High complexity
  - Consistency must be addressed specifically
  - Difficult to adapt for accelerators



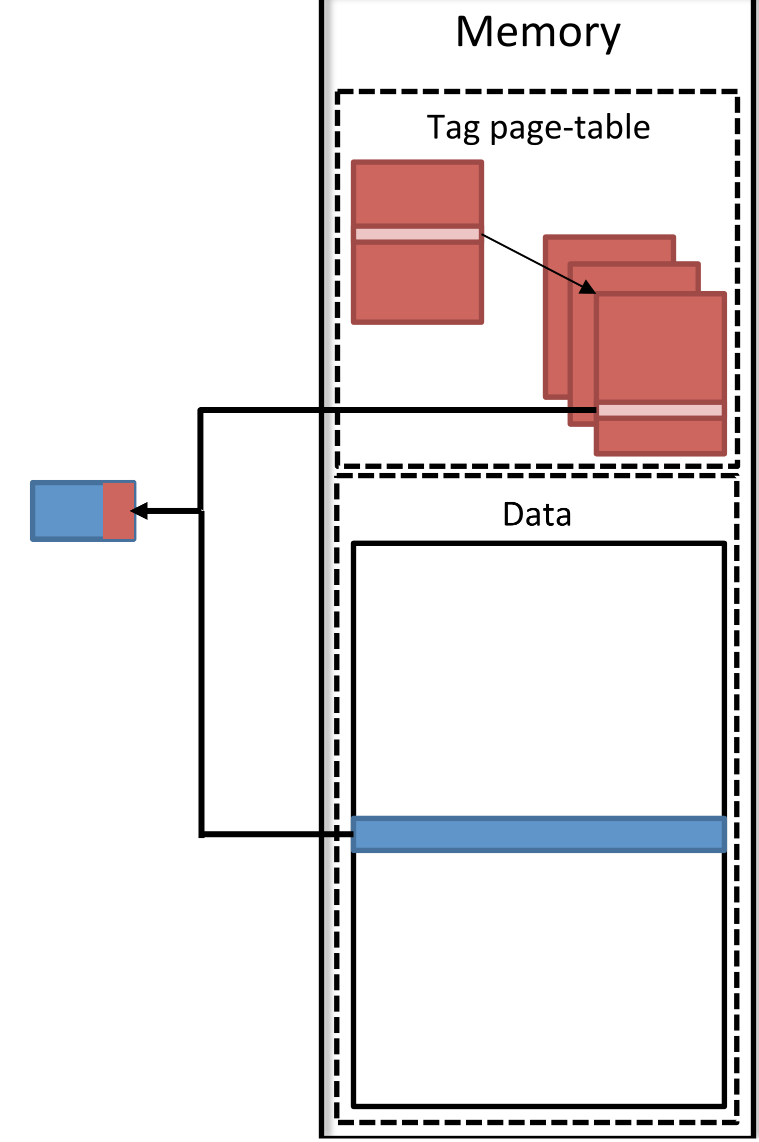
[Suh/DIFT, Flexitaint, Kannan/Copro, Deng/FPGA]

The WHISK architecture

- How to get the best of both schemes?
  - Low area overhead
  - Low complexity
- Hybrid scheme
  - “Decoupled” storage
  - “Integrated” interfaces

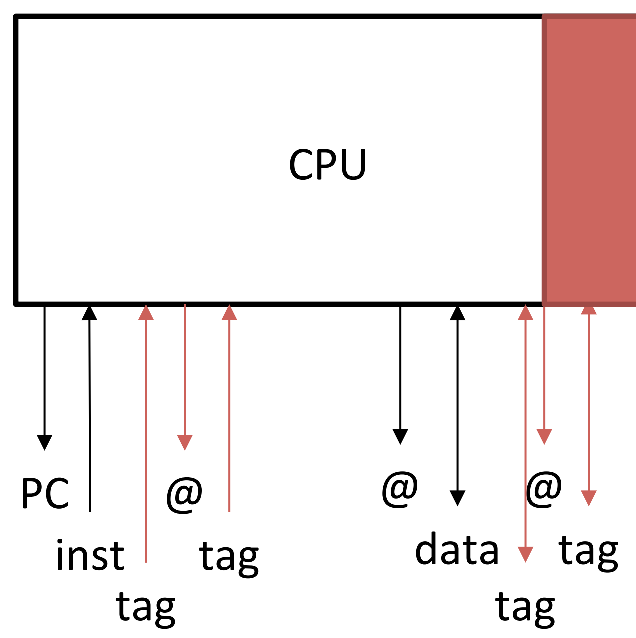
WHISK – Low area overhead

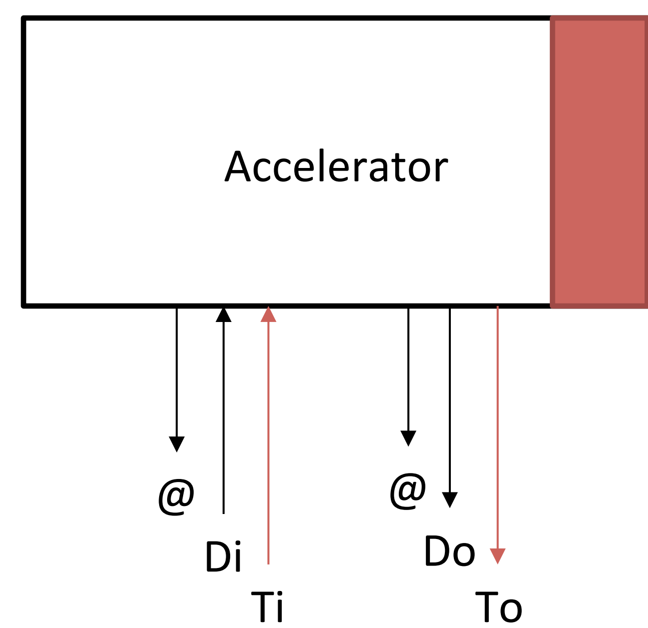
- Decoupled scheme
  - Page-table structure
    - First level: page granularity
    - Second level: (on-demand) word granularity
  - Physical address space



WHISK – Low complexity

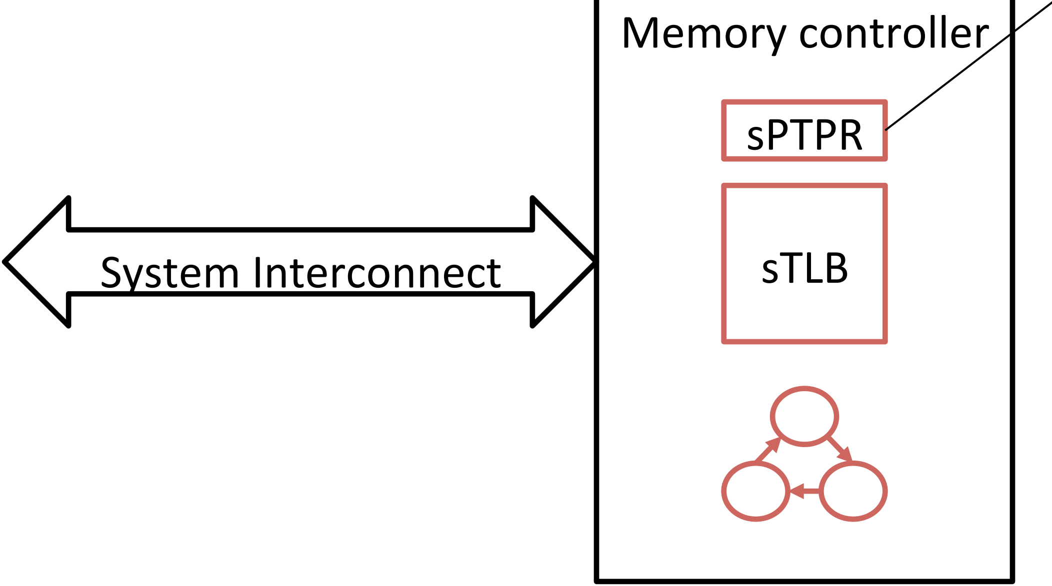
- Integrated interfaces
  - Processors
  - Accelerators

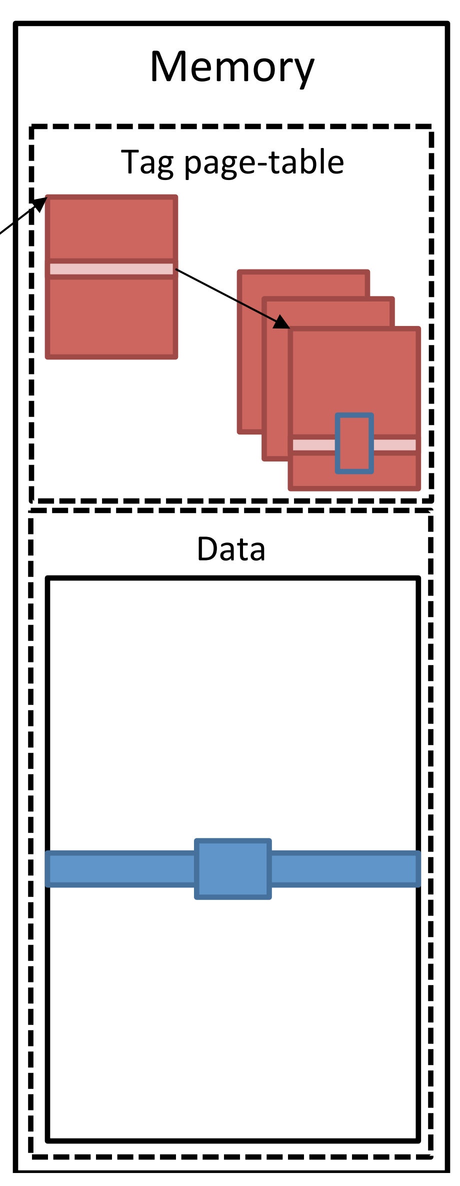




WHISK – Tag management (1)

- At platform level

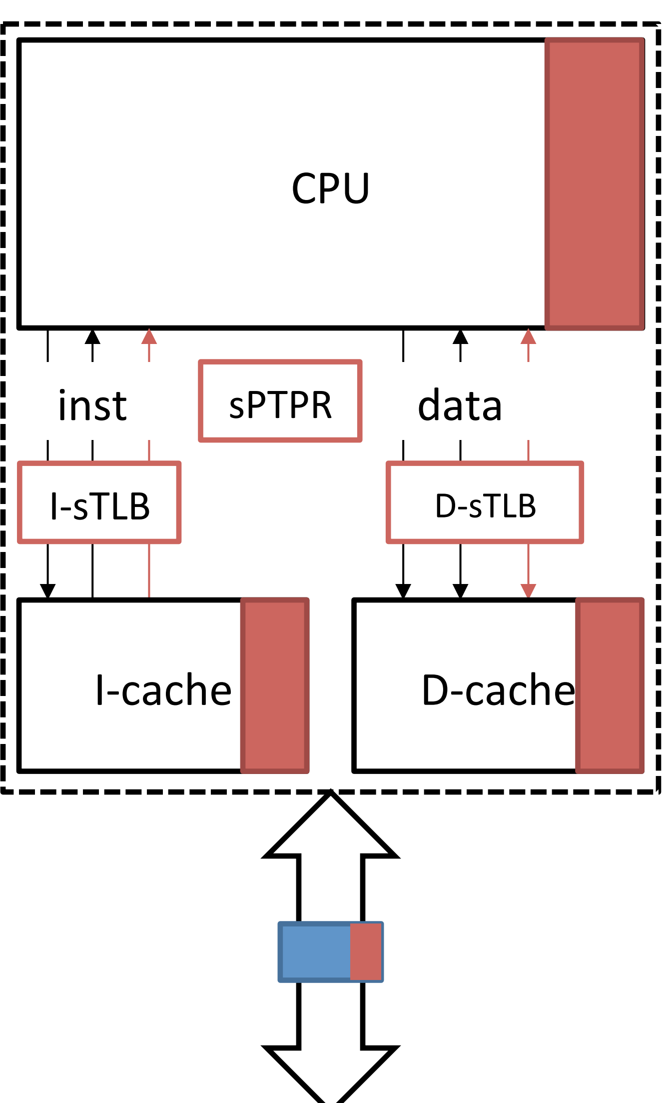




- Optimizations?

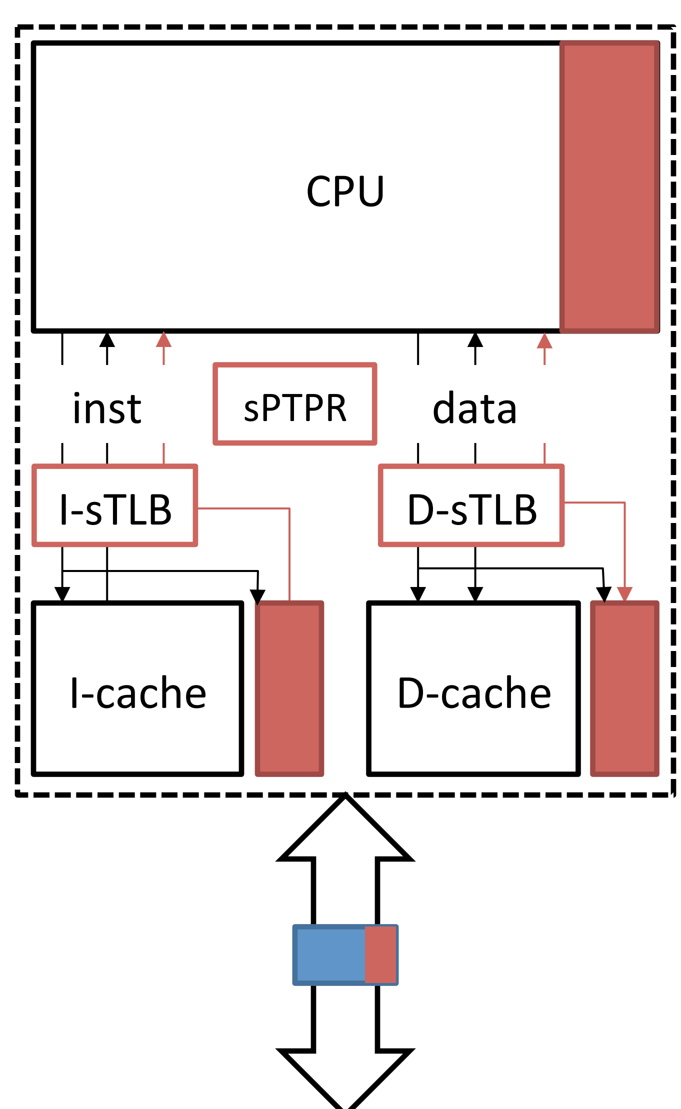
WHISK – Tag management (2)

- Processors
- sTLBs
  - Exploit page granularity
  - Handle page refinement



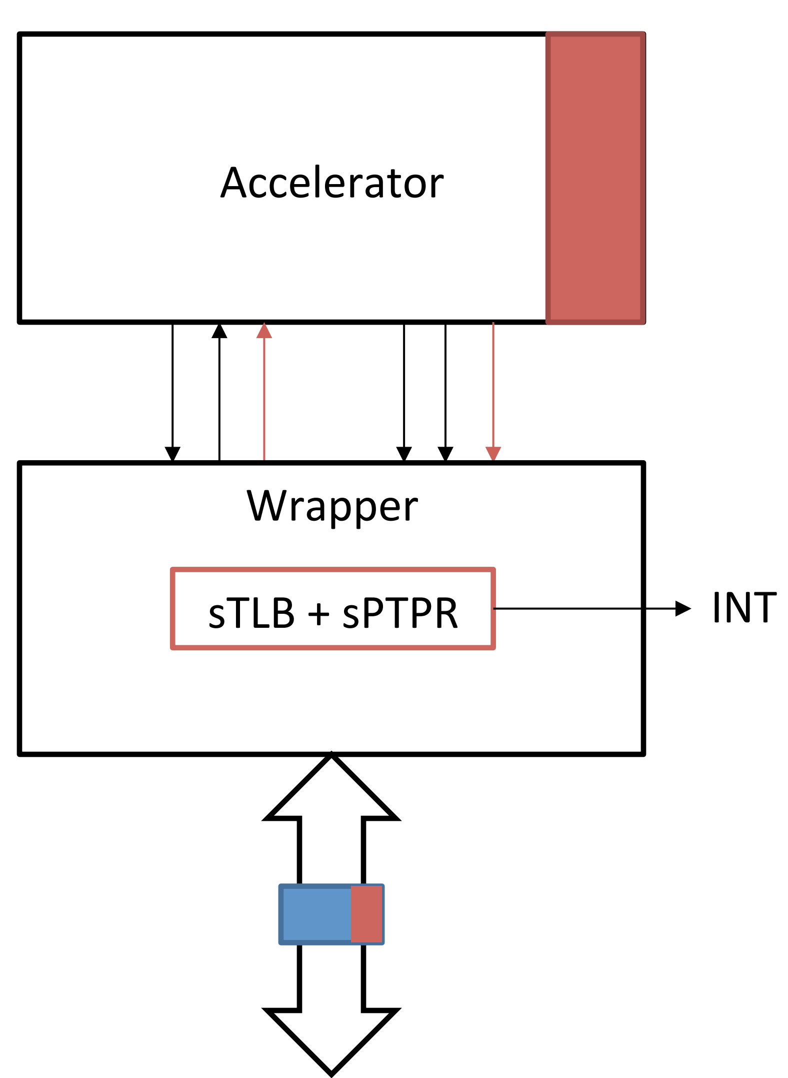
WHISK – Tag management (2)

- Processors
- sTLBs
  - Exploit page granularity
  - Handle page refinement
- Separate tag caches
  - Lower area
- Communication protocol
  - NONE, WITH, ONLY



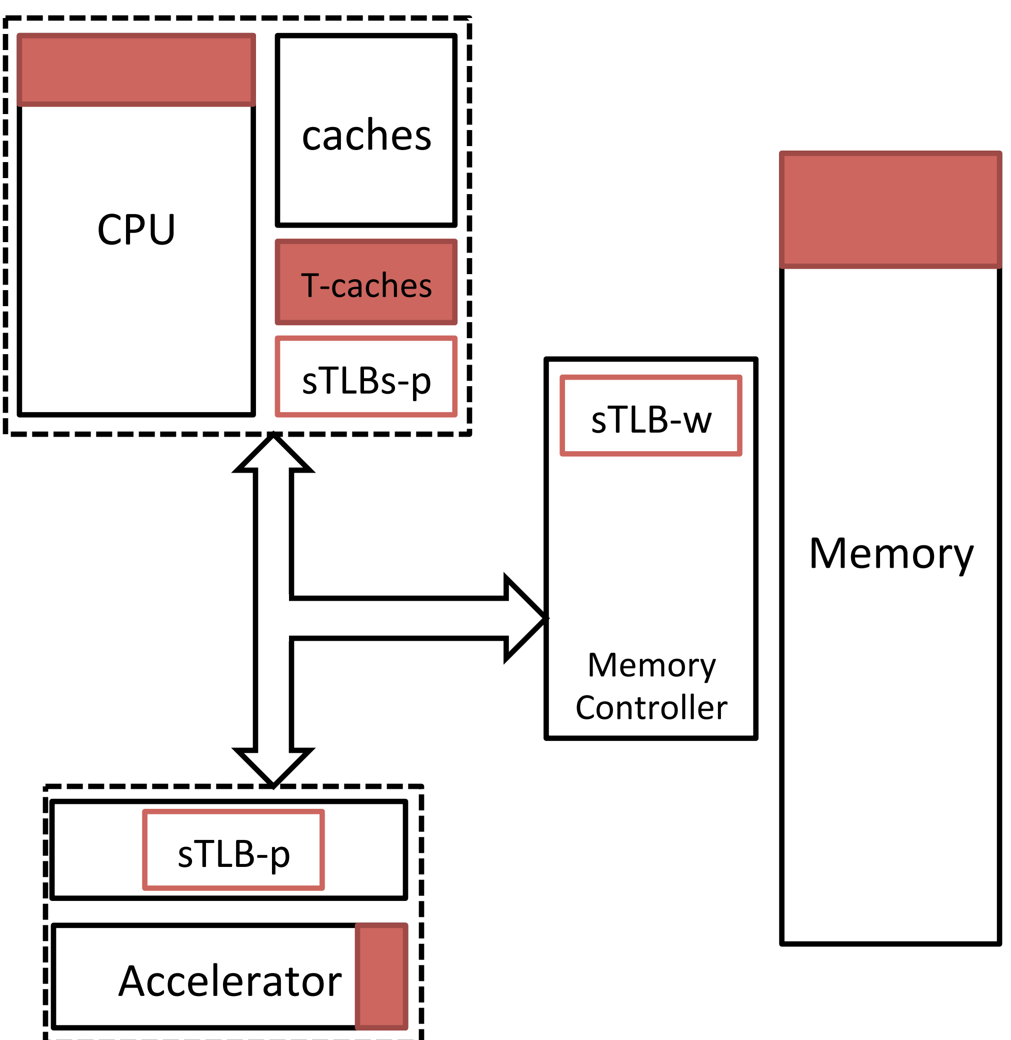
WHISK – Tag management (3)

- Accelerators
- sTLB
- Serializer/ Deserializer
- Page refinement



WHISK – Tag management (4)

- sTLBs
  - sTLB-p
  - sTLB-w
- Software support
  - Page table
  - PTPRs
  - Page refinement
  - Tag policies



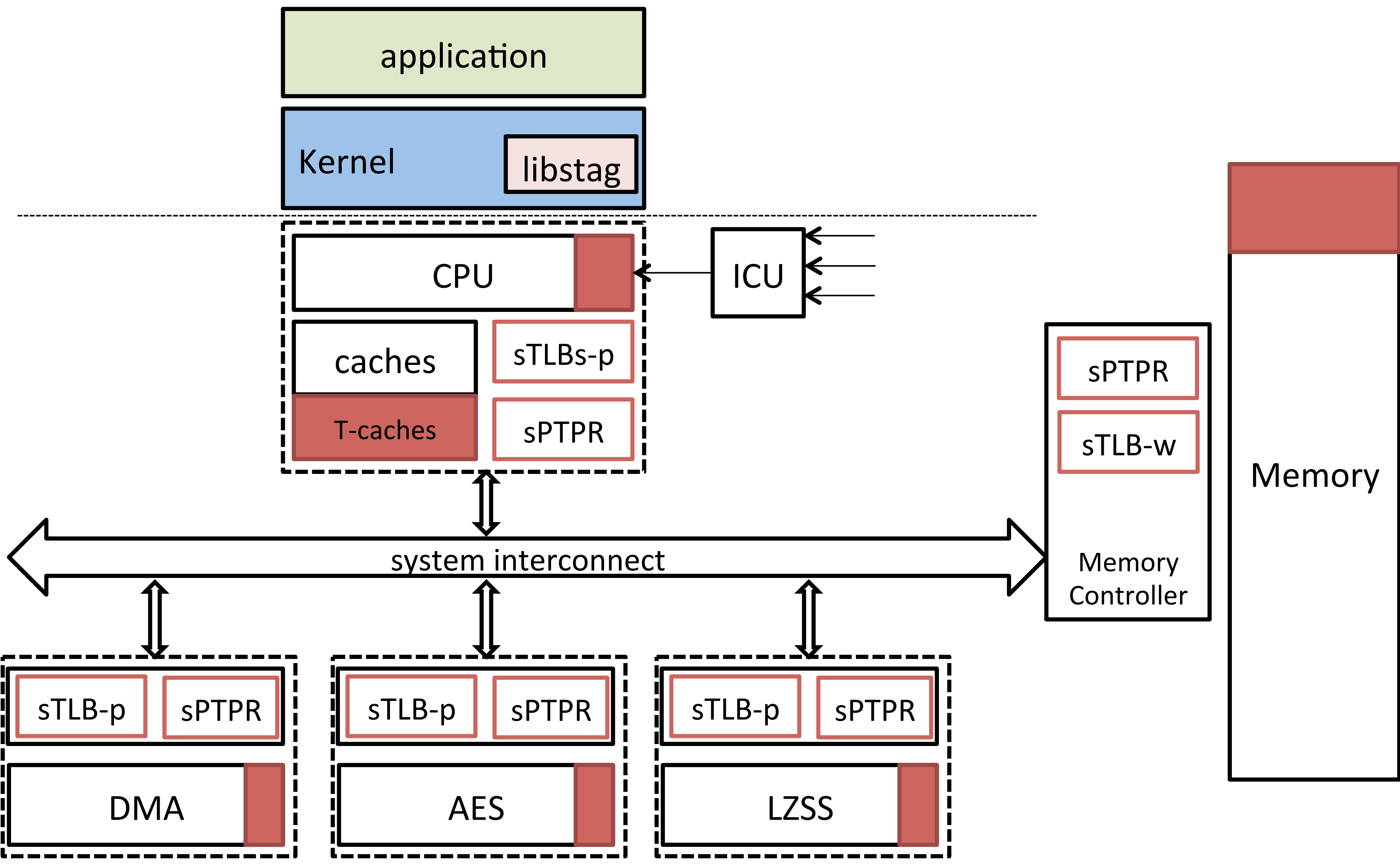
Implementation

- Hardware
  - Based on SoCLib simulation framework
  - MIPS processor (single-issue pipeline)
  - Write-back caches
  - Directory-based coherence protocol
  - No virtual memory
  - Crossbar interconnect

  - SystemC – BCA

- Software
  - MutekH: dedicated kernel for embedded systems

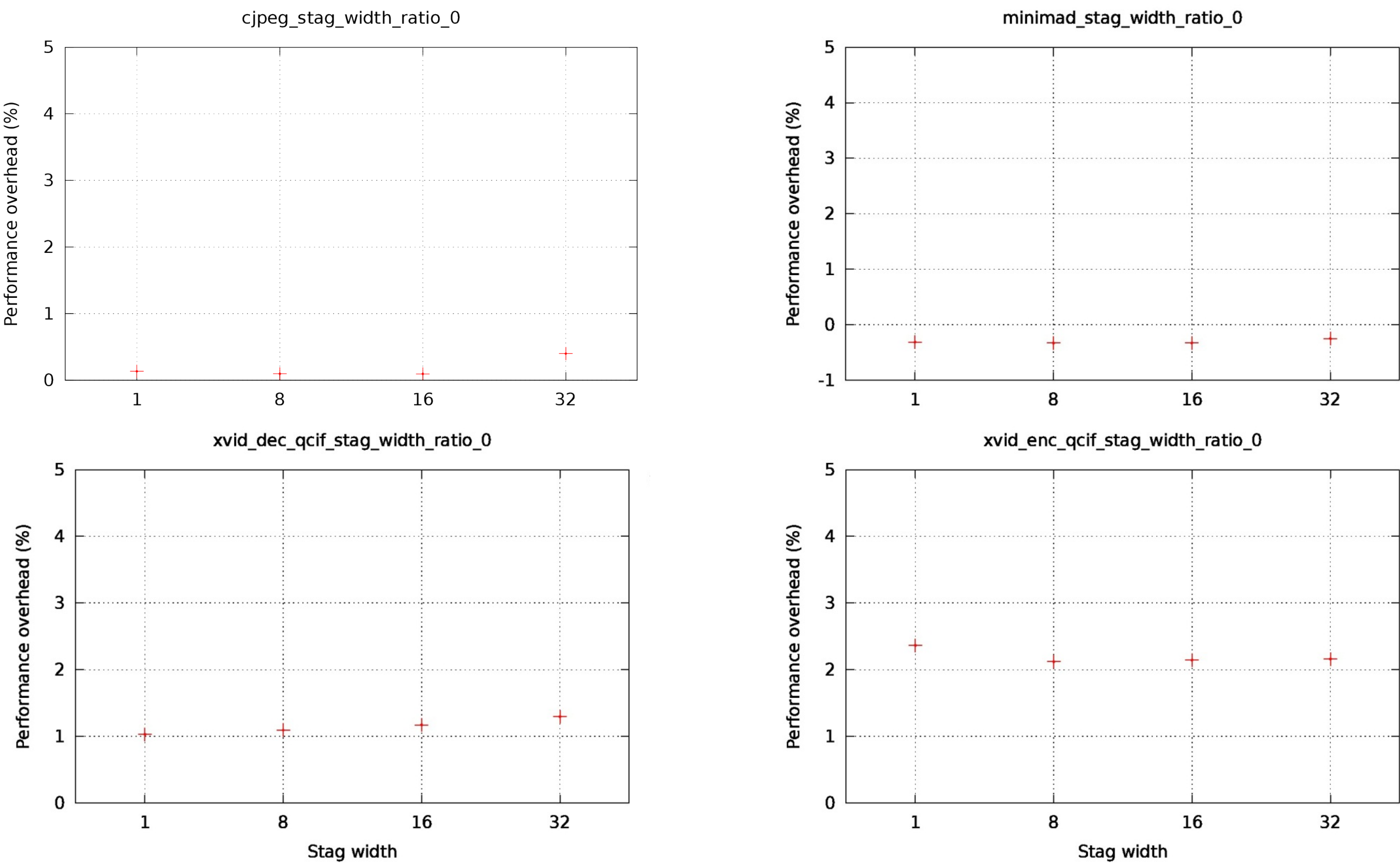
DIFT full system



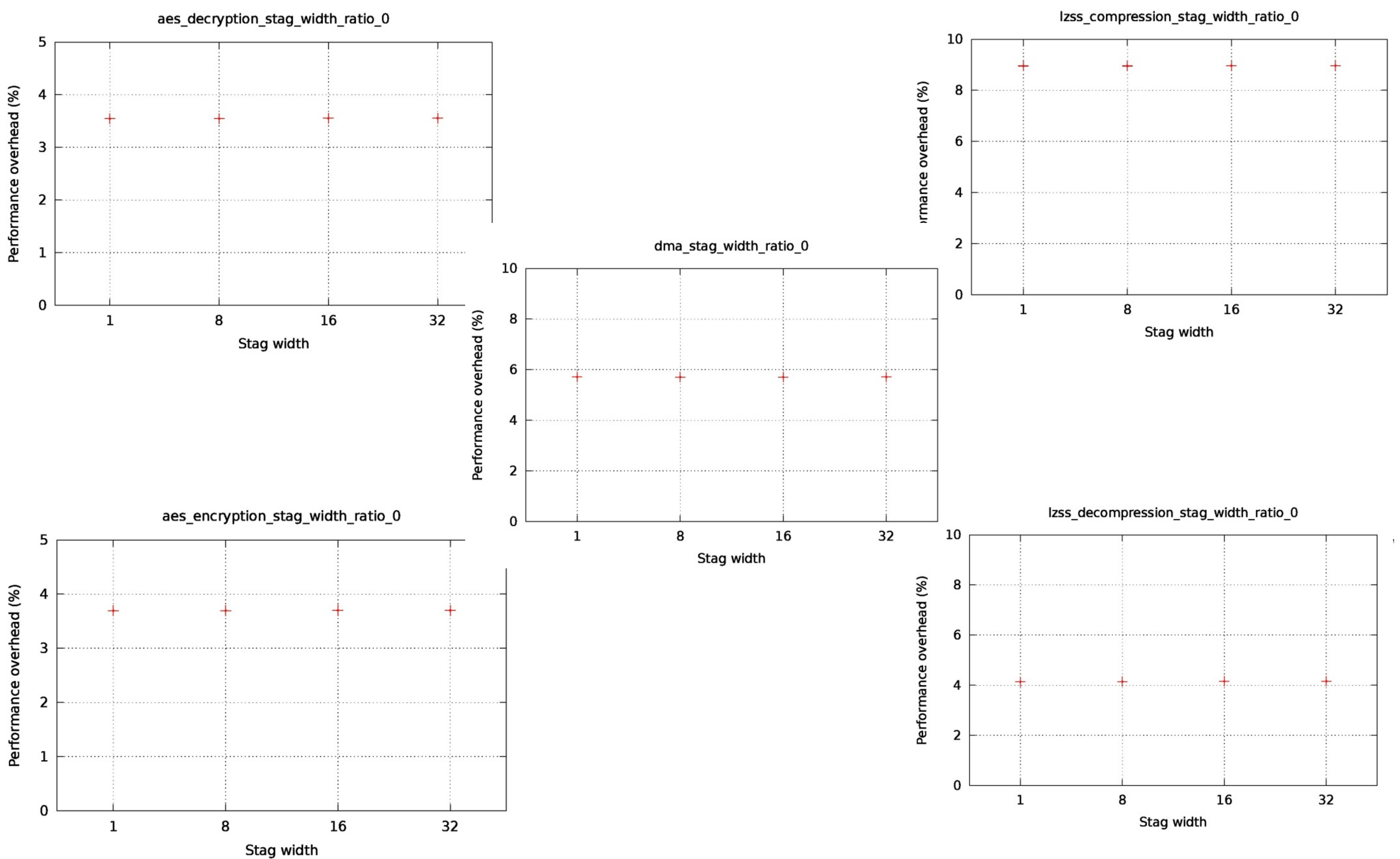
Evaluations

- Set of software benchmarks
  - Multimedia-oriented, data-intensive
  - cjpeg, minimad, xvid_enc, xvid_dec
- Set of hardware micro-benchmarks
  - Input buffer (256 pages) -> accelerator -> output buffer
- Baseline performance loss
  - DIFT infrastructure is active but unused
- Price of security vs. “degree” of security

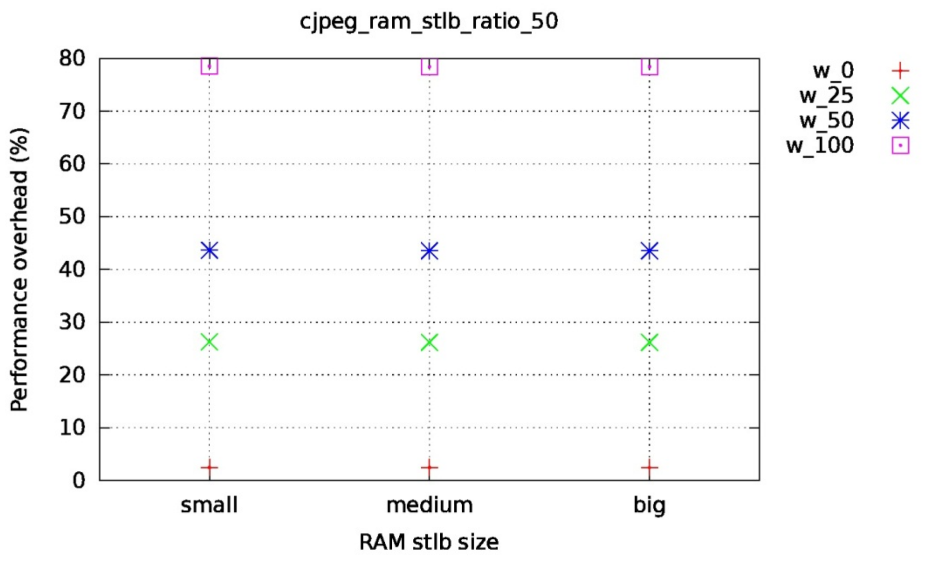
Baseline – software applications

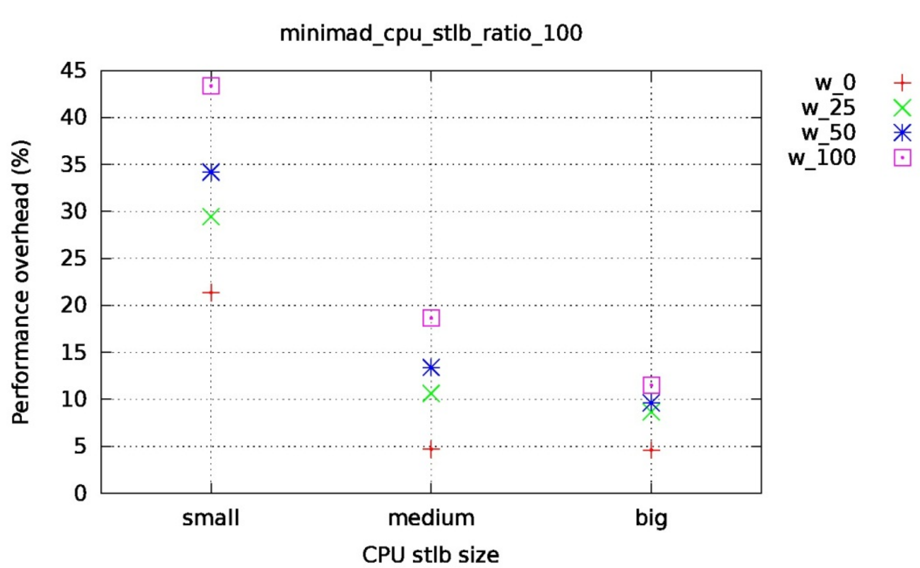


Baseline – accelerators

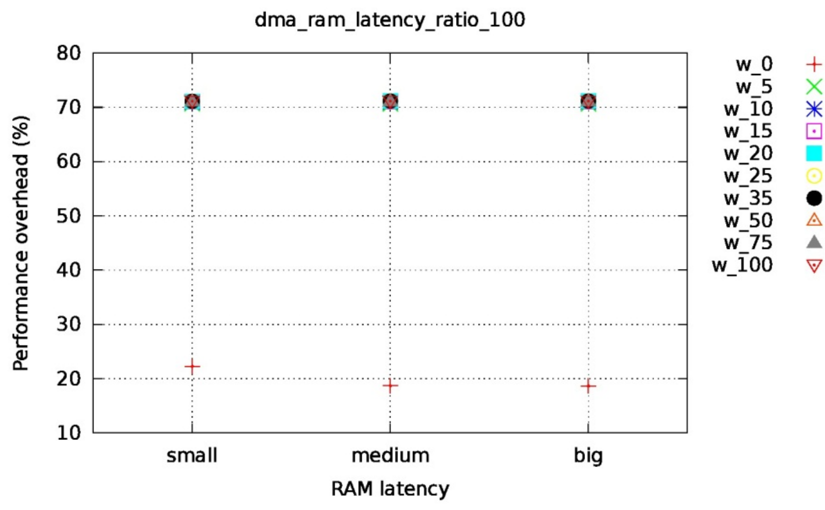


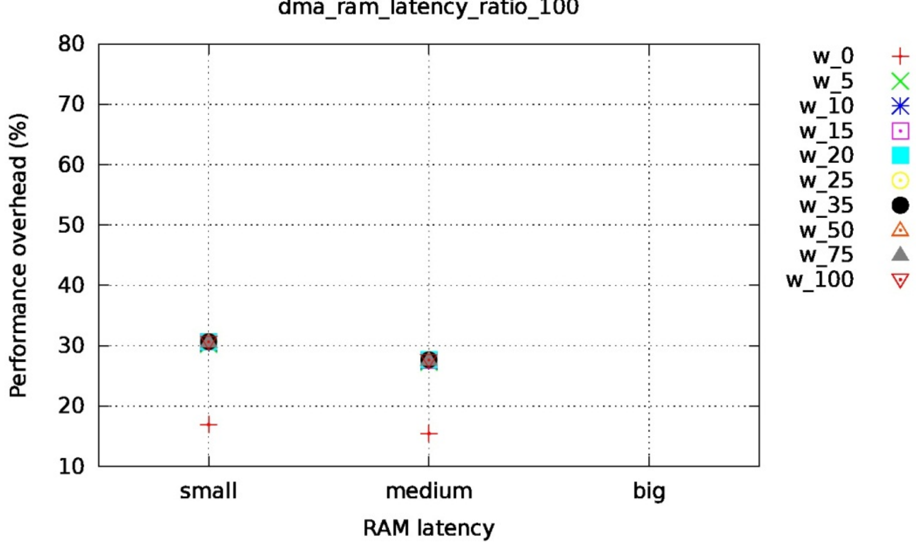
DIFT – software applications





DIFT – accelerators





Conclusion

- DIFT as a hardware security primitive
  - Full-system DIFT
  - Seamless integration of our DIFT platform backbone with past research
- Negligible performance loss when not utilizing tagging
- (Expected) linear correlation between the price of the security and the amount of tagging
- Short-term future work:
  - Bugs fixing
  - Finish simulations
  - Multi-processor system
- Long-term future work:
  - Externally managed DIFT system

Embedded System Exploitation and Defense
CRASH Site Visit
September 6, 2012

Ang Cui
Columbia IDS Lab
ang@cs.columbia.edu

Salvatore J. Stolfo
Columbia IDS Lab
sal@cs.columbia.edu

Autotomic Binary Structure Randomization (ABSR)

Lessons Learned From HP RFU Vulnerability

- Legit Features can be serious vulnerabilities
- Legit Features can't always be disabled

So…

- "disable" all unused "features" to reduce attack surface
  - Turn unused code into dead-code
- Dead-code can be used for defense
  - Binary randomization, re-structuring
  - ROP/Return-to-Lib detection

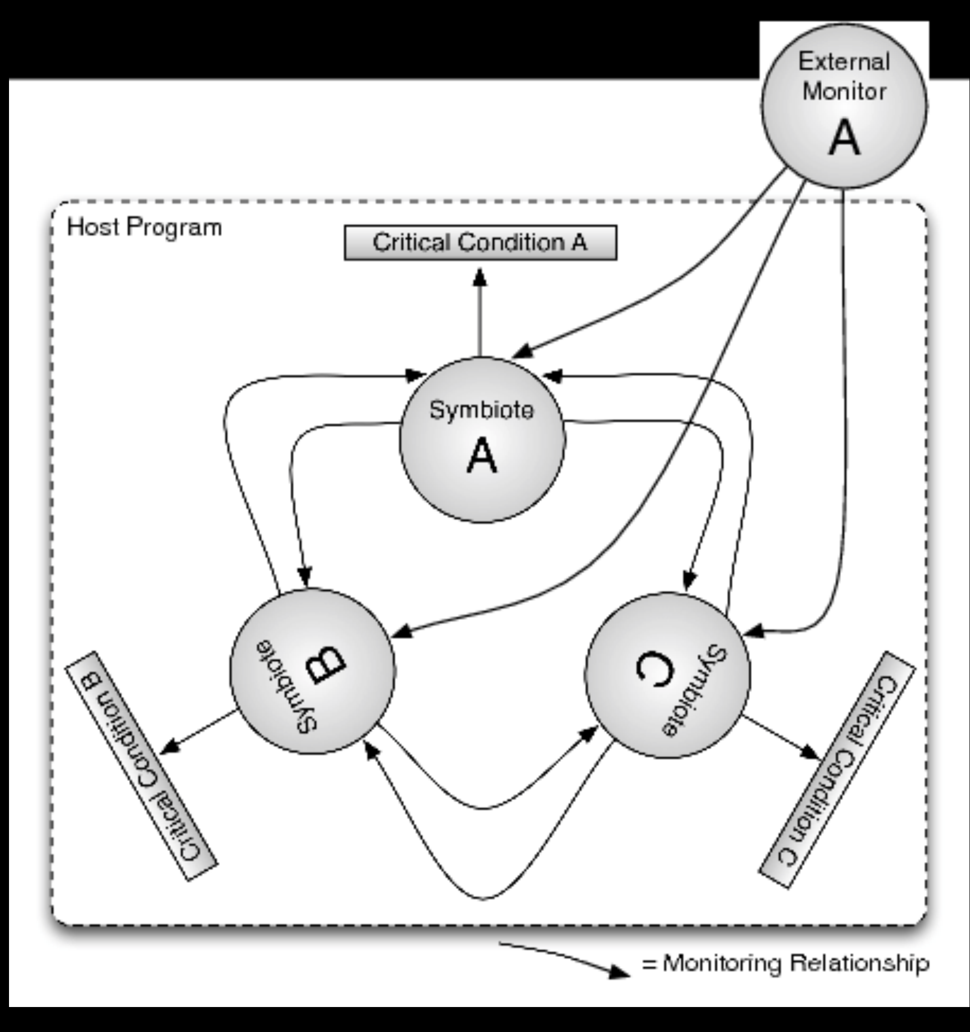
Autotomic Binary Structure Randomization (ABSR)

So…

- “disable” all unused “features” to reduce attack surface
  - AUTOTOMIC
- Dead-code can be used for defense
  - BINARY STRUCTURE RANDOMIZATION

Post-ABSR Symbiote Organization

Symbiotes in Self-Monitoring-Monitors Configuration



Autotomic Binary Structure Randomization (ABSR) STATUS

Initial Proof of Concept implementation

Autotomic Binary Structure Randomization (ABSR) STATUS

Presentation to Symantec, HP

Several Provisional patent filing

Michael Costello hired as FTE

Paper under review

BlackHat/Defcon presentations

Red Balloon Security Inc. founded
www.redballoonsecurity.com

PEREGRINE: Efficient Deterministic Multithreading through Schedule Relaxation

Heming Cui,
Jingyue Wu,
John Gallagher,
Huayang Guo,
Junfeng Yang

Software Systems Lab
Columbia University

Nondeterminism in Multithreading

- Different runs ➔ different behaviors, depending on thread schedules
- Cause lots of problems
  - Data race
  - Security exploit [HotPar '12]
  - …

Nondeterministic Synchronization

```
Thread 0                    Thread 1
mutex_lock(M)
*obj = …
mutex_unlock(M)
                            mutex_lock(M)
                            free(obj)
                            mutex_unlock(M)
```

```
Thread 0                    Thread 1
                            mutex_lock(M)
                            free(obj)
                            mutex_unlock(M)
mutex_lock(M)
*obj = …
mutex_unlock(M)
```

Apache Bug #21287

Data Race

```
Thread 0                    Thread 1
……                          ……
barrier_wait(B)             barrier_wait(B)
                            result += …
print(result)
```

```
Thread 0                    Thread 1
……                          ……
barrier_wait(B)             barrier_wait(B)
print(result)
                            result += …
```

FFT in SPLASH2

Deterministic Multithreading (DMT)

- Same input → same schedule
  - Addresses many problems due to nondeterminism
- Existing DMT systems enforce either of
  - ***Sync-schedule***: deterministic total order of synch operations (e.g., lock()/unlock())
  - ***Mem-schedule***: deterministic order of shared memory accesses (e.g., load/store)

Sync-schedule

- [TERN OSDI '10], [Kendo ASPLOS '09], etc
- Pros: efficient (16% overhead in Kendo)
- Cons: deterministic only when no races
  - Many programs contain races [Lu ASPLOS '08]

```
Thread 0              Thread 1          |  Thread 0              Thread 1
mutex_lock(M)                           |                        mutex_lock(M)
*obj = …                                |                        free(obj)
mutex_unlock(M)                         |                        mutex_unlock(M)
                      mutex_lock(M)     |  mutex_lock(M)
                      free(obj)         |  *obj = …
                      mutex_unlock(M)   |  mutex_unlock(M)
```

Apache Bug #21287

```
Thread 0              Thread 1          |  Thread 0              Thread 1
……                    ……                |  ……                    ……
barrier_wait(B)       barrier_wait(B)   |  barrier_wait(B)       barrier_wait(B)
                      result += …       |  print(result)
print(result)                           |                        result += …
```

FFT in SPLASH2

Mem-schedule

- [COREDET ASPLOS '10], [dOS OSDI '10], etc
- Pros: deterministic despite of data races
- Cons: high overhead (e.g., 1.2~10.1X slowdown in dOS)

Thread 0	Thread 1	Thread 0	Thread 1
……	……	……	……
barrier_wait(B)	barrier_wait(B)	barrier_wait(B)	barrier_wait(B)
	result += …	print(result)	
print(result)			result += …

FFT in SPLASH2

Open Challenge [WODET '11]

- Either determinism or efficiency, but not both

Type of Schedule	Determinism	Efficiency
Sync	☹	☺
Mem	☺	☹

Can we get both?

Yes, we can!

Type of Schedule	Determinism	Efficiency
Sync	☹	☺
Mem	☺	☹
PEREGRINE	☺	☺

PEREGRINE Insight

- ***Races rarely occur***
  - Intuitively, many races ➔ already detected
  - Empirically, six real apps ➔ up to 10 races occured
- ***Hybrid schedule***
  - Sync-schedule in race-free portion (major)
  - Mem-schedule in racy portion (minor)

PEREGRINE: Efficient DMT

- ***Schedule Relaxation***
  - Record execution trace for new input
  - Relax trace into ***hybrid schedule***
  - Reuse on many inputs: deterministic + efficient
    - Reuse rate is high (e.g., 90.3% for Apache, [TERN OSDI '10])
    - Automatic using new program analysis techniques

- Run in Linux, user space
- Handle Pthread synchronization operations
- Work with server programs [TERN OSDI '10]

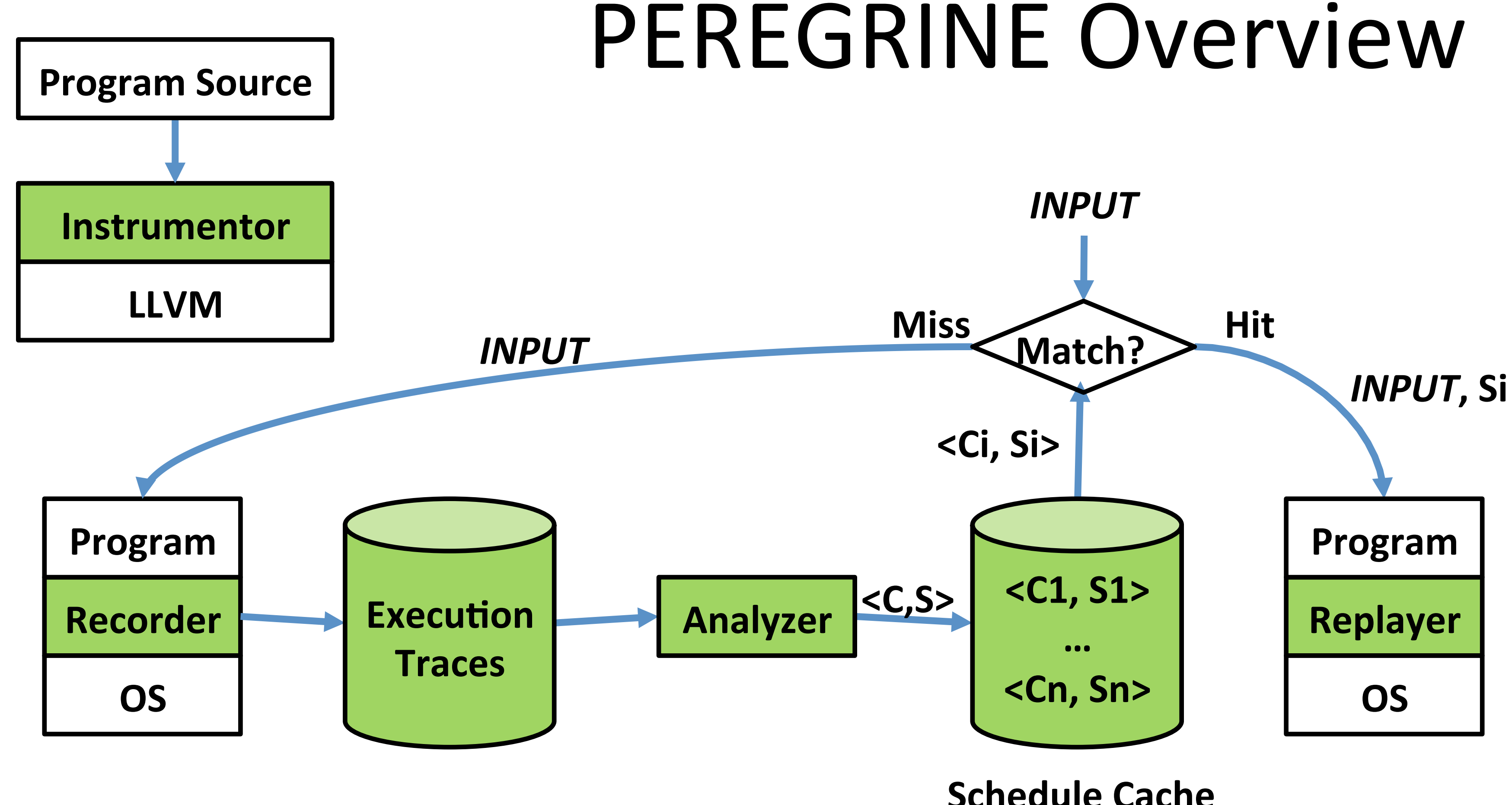
Summary of Results

- Evaluated on a diverse set of 18 programs
  - 4 real applications: Apache, PBZip2, aget, pfscan
  - 13 scientific programs (10 from SPLASH2, 3 from PARSEC)
  - Racey (popular stress testing tool for DMT)
- Deterministically resolve all races
- Efficient: 54% faster to 49% slower
- Stable: frequently reuse schedules for 9 programs
  - Many benefits: e.g., reuse good schedules [TERN OSDI '10]

Outline

- **PEREGRINE overview**
- An example
- Evaluation
- Conclusion
- Future work

PEREGRINE Overview



Outline

- PEREGRINE overview
- **An example**
- Evaluation
- Conclusion
- Future work

An Example

```
main(argc, char *argv[]) {
  nthread = atoi(argv[1]);              // Read input.
  size = atoi(argv[2]);
  for(i=1; i<nthread; ++i)
    pthread_create(worker);             // Create children threads.
  worker();                             // Work.
  // Missing pthread_join()
  if ((flag=atoi(argv[3]))==1)          // if “flag” is 1, update “result”.
    result += …;
  printf(“%d\n”, result);               // Read from “result”.
}
worker() {
  char *data;
  data = malloc(size/nthread);          // Allocate data with “size/nthread”.
  for(i=0; i<size/nthread; ++i)
    data[i] = myRead(i);                // Read data from disk and compute.
  pthread_mutex_lock(&mutex);           // Grab mutex.
  result += …;                          // Write to “result”.
  pthread_mutex_unlock(&mutex);
}
```

Instrumentor

```
main(argc, char *argv[]) {
  nthread = atoi(argv[1]);        // Instrument command line arguments.
  size = atoi(argv[2]);
  for(i=1; i<nthread; ++i)
    pthread_create(worker);
  worker();
  // Missing pthread_join()
  if ((flag=atoi(argv[3]))==1)
    result += …;
  printf("%d\n", result);
}
worker() {
  char *data;
  data = malloc(size/nthread);
  for(i=0; i<size/nthread; ++i)
    data[i] = myRead(i);          // Instrument read() function within myRead().
  pthread_mutex_lock(&mutex);
  result += …;
  pthread_mutex_unlock(&mutex);
}
```

Instrumentor

```
main(argc, char *argv[]) {
  nthread = atoi(argv[1]);        // Instrument command line arguments.
  size = atoi(argv[2]);
  for(i=1; i<nthread; ++i)
    pthread_create(worker);       // Instrument synchronization operation.
  worker();
  // Missing pthread_join()
  if ((flag=atoi(argv[3]))==1)
    result += …;
  printf("%d\n", result);
}
worker() {
  char *data;
  data = malloc(size/nthread);
  for(i=0; i<size/nthread; ++i)
    data[i] = myRead(i);          // Instrument read() function.
  pthread_mutex_lock(&mutex);     // Instrument synchronization operation.
  result += …;
  pthread_mutex_unlock(&mutex);   // Instrument synchronization operation.
}
```

$./a.out 2 2 0

Recorder

```
main(argc, char *argv[]) {
  nthread = atoi(argv[1]);
  size = atoi(argv[2]);
  for(i=1; i<nthread; ++i)
    pthread_create(worker);
  worker();
  // Missing pthread_join()
  if ((flag=atoi(argv[3]))==1)
    result += …;
  printf("%d\n", result);
}
worker() {
  char *data;
  data = malloc(size/nthread);
  for(i=0; i<size/nthread; ++i)
    data[i] = myRead(i);
  pthread_mutex_lock(&mutex);
  result += …;
  pthread_mutex_unlock(&mutex);
}
```

```
Thread 0                          Thread 1

main()
  nthread=atoi()
  size=atoi()
  (1<nthread)==1
  pthread_create()
  (2<nthread)==0
  worker()                        worker()
    data=malloc()                   data=malloc()
    (0<size/nthread)==1             (0<size/nthread)==1
    data[i]=myRead()                data[i]=myRead()
    (1<size/nthread)==0             (1<size/nthread)==0
    lock()
    result+=…;
    unlock()
                                    lock()
                                    result+=…;
                                    unlock()
  (flag==1)==0
  printf(…,result)
```

$./a.out 2 2 0 Recorder

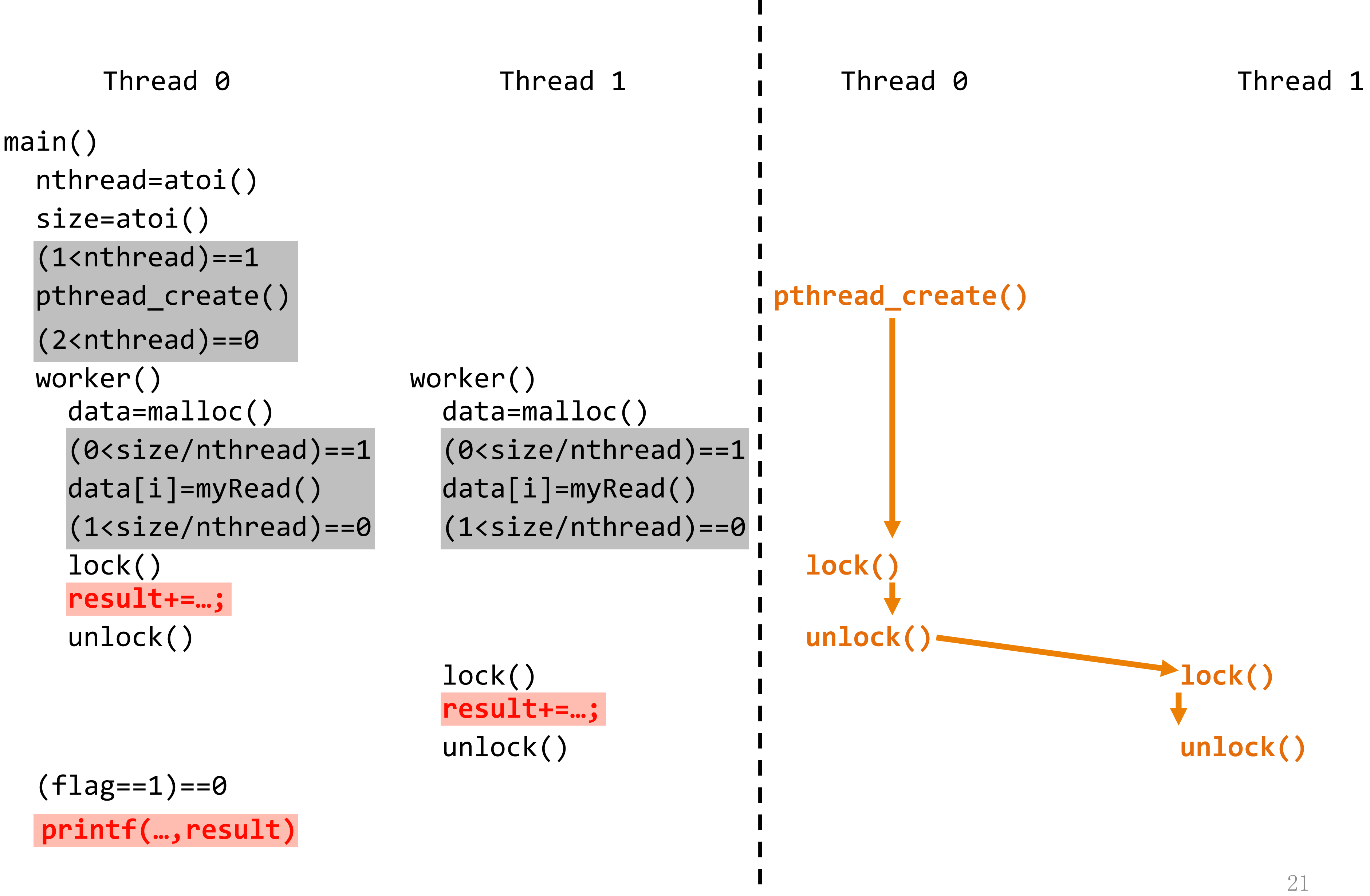
```
Thread 0                          Thread 1

main()
  nthread=atoi()
  size=atoi()
  (1<nthread)==1
  pthread_create()
  (2<nthread)==0
  worker()                        worker()
    data=malloc()                   data=malloc()
    (0<size/nthread)==1             (0<size/nthread)==1
    data[i]=myRead()                data[i]=myRead()
    (1<size/nthread)==0             (1<size/nthread)==0
    lock()
    result+=…;
    unlock()
                                    lock()
                                    result+=…;
                                    unlock()
  (flag==1)==0
  printf(…,result)
```

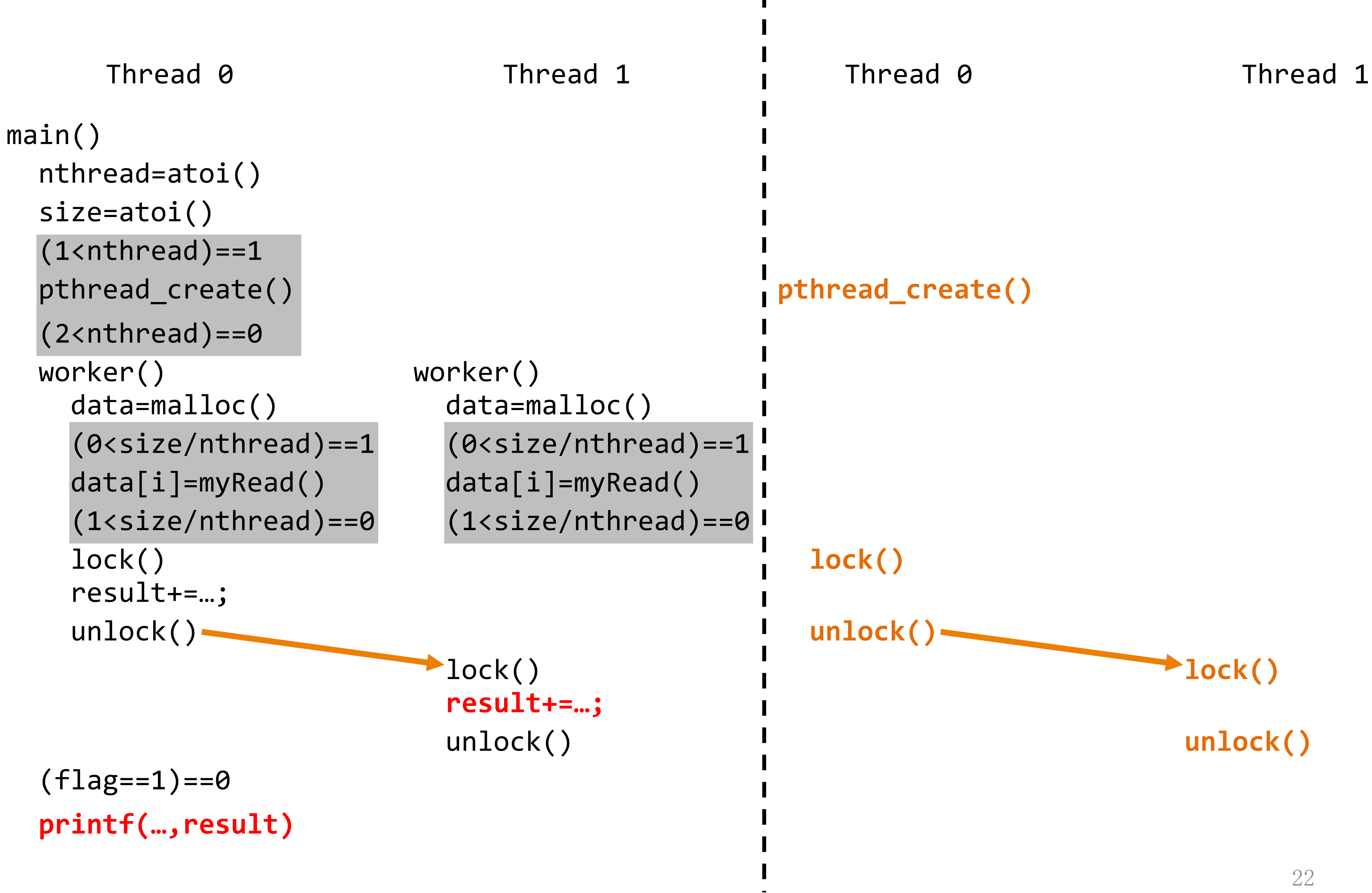
Analyzer: Hybrid Schedule

Thread 0	Thread 1
main()	
nthread=atoi()	
size=atoi()	
(1<nthread)==1	
pthread_create()	
(2<nthread)==0	
worker()	worker()
data=malloc()	data=malloc()
(0<size/nthread)==1	(0<size/nthread)==1
data[i]=myRead()	data[i]=myRead()
(1<size/nthread)==0	(1<size/nthread)==0
lock()	
result+=…;	
unlock()	
	lock()
	result+=…;
	unlock()
(flag==1)==0	
printf(…,result)	

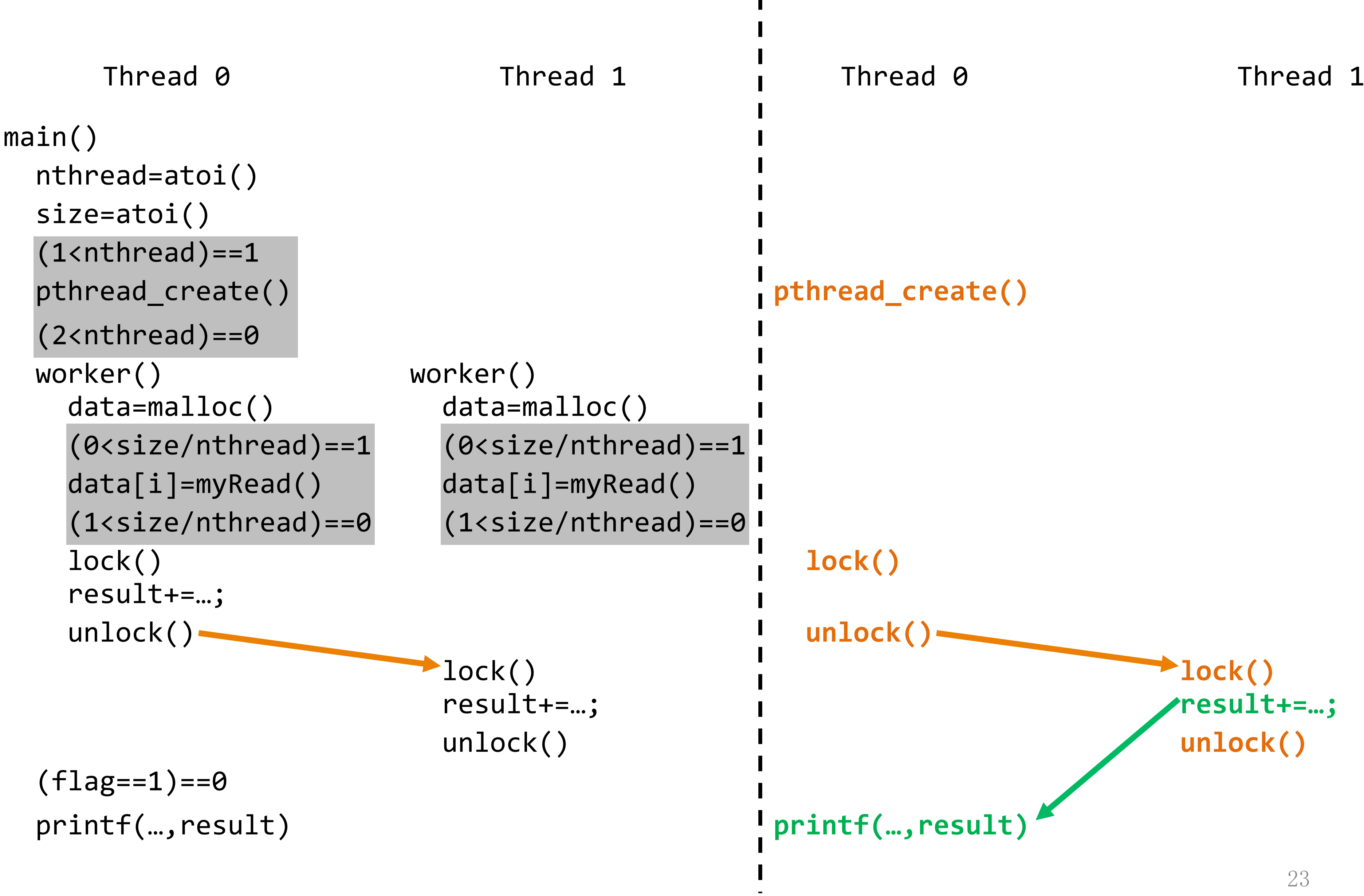
Analyzer: Hybrid Schedule



Analyzer: Hybrid Schedule



Analyzer: Hybrid Schedule



./a.out 2 2 0

Analyzer: Precondition

Thread 0 Thread 1

main()
nthr
size
(1<n
pthr
(2<n
work
da
(0
da
(1
lo
re
un

```
main(argc, char *argv[]) {
  nthread = atoi(argv[1]);
  size = atoi(argv[2]);
  for(i=1; i<nthread; ++i)
    pthread_create(worker);
  worker();
  // Missing pthread_join()
  if ((flag=atoi(argv[3]))==1)
    result += …;
  printf("%d\n", result);
}
……
```

d)==1
()
d)==0

unlock()

(flag==1)==0
printf(…,result)

Challenges

- Ensure schedule is feasible
- Ensure no new races

Hybrid Schedule

Thread 0 Thread 1

pthread_create()
lock()
unlock() → lock()
result+=…;
unlock()
printf(…,result)

Naïve Approach to Computing Preconditions

```
      Thread 0                    Thread 1

main()
  nthread=atoi()
  size=atoi()
  (1<nthread)==1
  pthread_create()
  (2<nthread)==0
  worker()                  worker()
    data=malloc()             data=malloc()
    (0<size/nthread)==1       (0<size/nthread)==1
    data[i]=myRead()          data[i]=myRead()
    (1<size/nthread)==0       (1<size/nthread)==0
    lock()
    result+=…;
    unlock()
                              lock()
                              result+=…;
                              unlock()
  (flag==1)==0
  printf(…,result)
```

```
nthread==2
size==2
flag!=1
```

Analyzer: Preconditions (a Naïve Way)

```
      Thread 0                    Thread 1

main()
  nthread=atoi()
  size=atoi()
  (1<nthread)==1
  pthread_create()
  (2<nthread)==0
  worker()                   worker()
    data=malloc()              data=malloc()
    (0<size/nthread)==1        (0<size/nthread)==1
    data[i]=myRead()           data[i]=myRead()
    (1<size/nthread)==0        (1<size/nthread)==0
    lock()
    result+=…;
    unlock()
                               lock()
                               result+=…;
                               unlock()
  (flag==1)==0
  printf(…,result)
```

```
nthread==2
size==2
flag!=1
```

- Problem: over-constraining!
  - *size* must be 2 to reuse
- Absorbed most of our brain power in this paper!
- Solution: two new program analysis techniques; see paper

Analyzer: Preconditions

Thread 0

main()
nthread=atoi()
size=atoi()
(1<nthread)==1
pthread_create()
(2<nthread)==0
worker()
~~data=malloc()~~
~~(0<size/nthread)==1~~
~~data[i]=myRead()~~
~~(1<size/nthread)==0~~
lock()
~~result+=…;~~
unlock()

Thread 1

worker()
~~data=malloc()~~
~~(0<size/nthread)==1~~
~~data[i]=myRead()~~
~~(1<size/nthread)==0~~
lock()
result+=…;
unlock()

Thread 0

(flag==1)==0
printf(…,result)

nthread==2

flag!=1

./a.out 2 1000 3

Replayer

Thread 0	Thread 1
main()	
nthread=atoi()	
size=atoi()	
(1<nthread)==1	
pthread_create()	
(2<nthread)==0	
worker()	worker()
~~data=malloc()~~	~~data=malloc()~~
~~(0<size/nthread)==1~~	~~(0<size/nthread)==1~~
~~data[i]=myRead()~~	~~data[i]=myRead()~~
~~(1<size/nthread)==0~~	~~(1<size/nthread)==0~~
lock()	
~~result+=…;~~	
unlock()	
	lock()
	result+=…;
	unlock()
(flag==1)==0	
printf(…,result)	

Preconditions

nthread==2

flag!=1

Hybrid Schedule

Thread 0	Thread 1
pthread_create()	
lock()	
unlock() →	
	lock()
	result+=…;
	unlock()
printf(…,result) ←	

Benefits of PEREGRINE

```
     Thread 0                      Thread 1

main()
  nthread=atoi()
  size=atoi()
  (1<nthread)==1
  pthread_create()
  (2<nthread)==0
  worker()                      worker()
    data=malloc()                 data=malloc()
    (0<size/nthread)==1           (0<size/nthread)==1
    data[i]=myRead()              data[i]=myRead()
    (1<size/nthread)==0           (1<size/nthread)==0
    lock()
    result+=…;
    unlock()
                                lock()
                                result+=…;
                                unlock()
  (flag==1)==0
  printf(…,result)
```

- **Deterministic:** resolve race on ***result***; no new data races
- **Efficient**: loops on ***data[]*** run in parallel
- **Stable** [TERN OSDI '10]: can reuse on any data size or contents
- Other applications possible; talk to us!

Outline

- PEREGRINE overview
- An example
- **Evaluation**
- Conclusion
- Future work

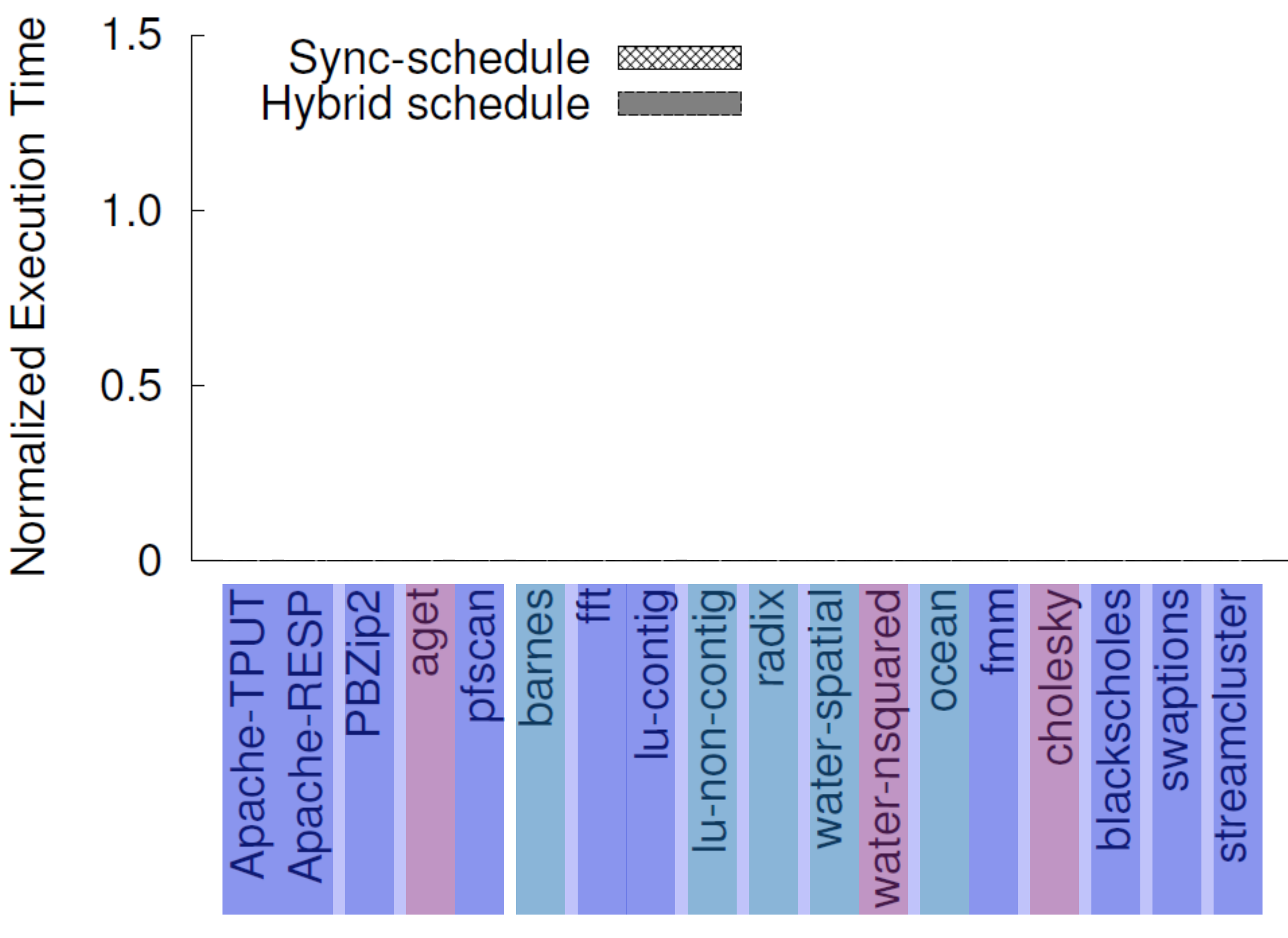
General Experiment Setup

- Program-workload
  - **Apache**: download a 100KB html page using ApacheBench
  - **PBZip2**: compress a 10MB file
  - **Aget**: download linux-3.0.1.tar.bz2, 77MB.
  - **Pfscan**: scan keyword “return” 100 files from gcc project
  - **13 scientific benchmarks** (10 from SPLASH2, 3 from PARSEC): run for 1-100 ms
  - **Racey**: default workload
- Machine: 2.67GHz dual-socket quad-core Intel Xeon machine (eight cores) with 24GB memory
- Concurrency: eight threads for all experiments

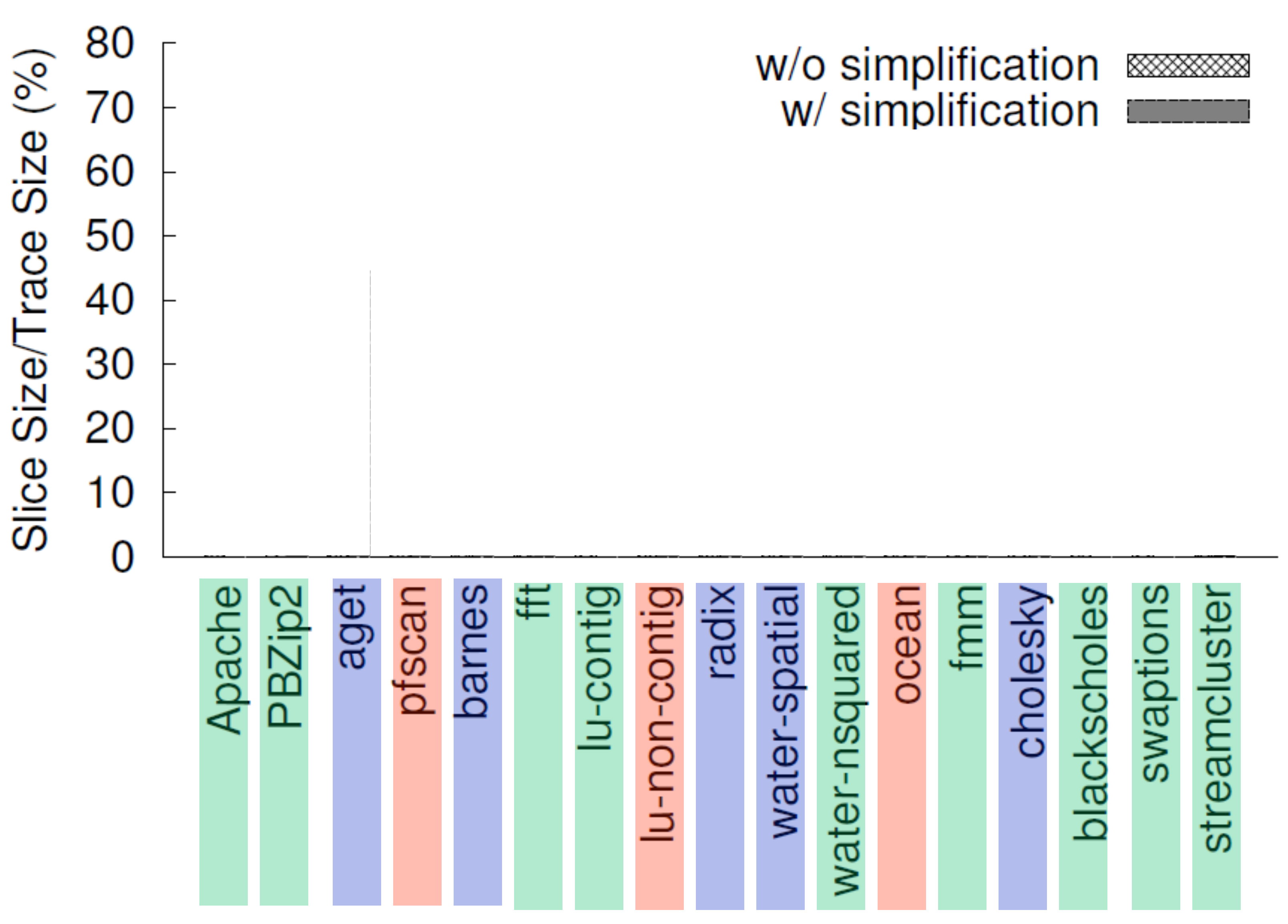
Determinism

Program	# Races	Sync-schedule	Hybrid schedule
Apache	0	☺	☺
PBZip2	4	☹	☺
barnes	5	☹	☺
fft	10	☹	☺
lu-non-contig	10	☹	☺
streamcluster	0	☺	☺
racey	167974	☹	☺

Overhead in Reusing Schedules



% of Instructions Left in the Trace



Conclusion

- Hybrid schedule: combine the best of both sync-schedule and mem-schedules
- PEREGRINE
  - Schedule relaxation to compute hybrid schedules
  - Deterministic (make all 7 racy programs deterministic)
  - Efficient (54% faster to 49% slower)
  - Stable (frequently reuse schedule for 9 out of 17)

Future Work

- Check critical system rules
  - Detected 113 system rule violations from widely used Linux utilities
  - 10 serious data loss errors in widely used utilities with 2 confirmed by developers
- Speedup distribution systems model checking
- Support OpenMP

Related Work

- Deterministic Execution
  - [Grace OOPSLA '09], [Kendo ASPLOS '09], [DMP ASPLOS '09], [COREDET ASPLOS '10], [dOS OSDI '10], [Determinator OSDI '10], [dthreads SOSP '11]
- Deterministic Replay
  - [ReVirt OSDI '02] , [SMP-ReVirt VEE '08], [Capo ASPLOS '09], [PRES SOSP '09], [ODR SOSP '09], [Scribe SIGMETRICS '10]
- Concurrency Errors
  - [Eraser TOCS '97], [Racex SOSP '03], [RaceTrack SOSP '05], [Avio ASPLOS '06], [Lu el ASPLOS '08], [CTrigger ASPLOS '09]
- Symbolic Execution
  - [CUTE FSE-13], [EXE CCS '06], [Yang el SP '06], [Bouncer SOSP '07], [KLEE OSDI '08], [Castro el ASPLOS '08]

Thank you!

Questions?

Liberty Architecture

Jordan Fix

Soumyadeep Ghosh

Advisor: David I. August

THE LIBERTY RESEARCH GROUP

Princeton University

	Traditional ISA	Virtual Machine Code
Execution Type	Native	Interpreted
Execution Speed	Fast	Slow
Analyzability	Low	High
Optimizability	Low	High
Safety	Unsafe	Safe

- Native machine code enables fast execution
- Richer expression of the program means easier and better analysis for security, optimization, parallelization, etc.

The Liberty Architecture combines the best of both

GOAL: Rethink architecture design to achieve security, longevity, performance using the minimal set of features

Security

- Control flow and data integrity (this talk)

Longevity

- Restores the abstraction broken by Multicore and GPUs
- System is provided all useful program information
- Program information can be used in new architecture paradigms (i.e. reconfigurable)

Performance

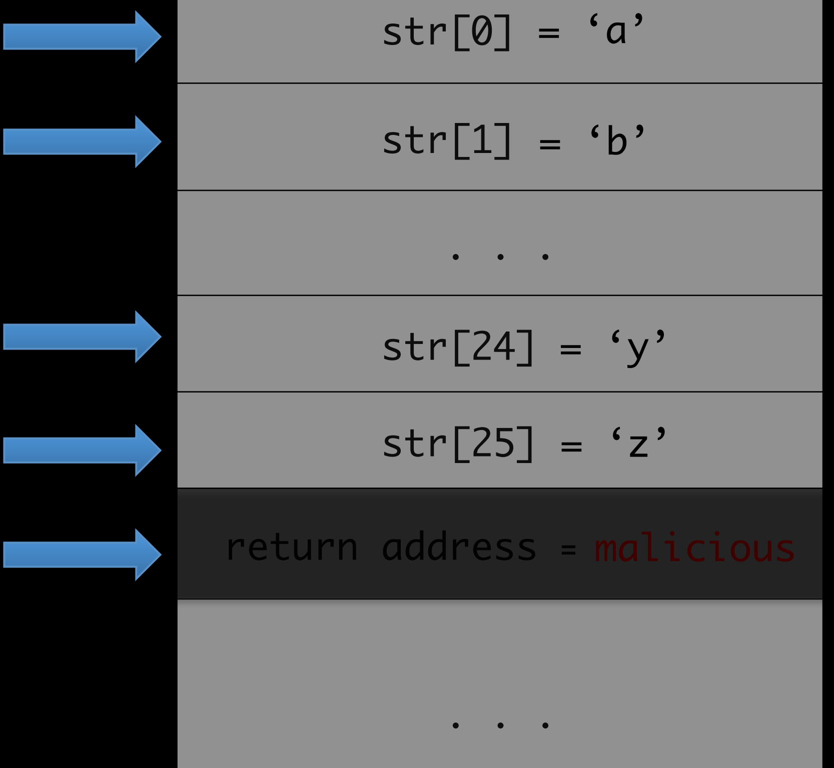
- Dynamic optimization
- Smart response to dynamically changing user requirements* and to input set variation

*Arun Raman, Ayal Zaks, Jae W. Lee, David I. August. Parcae: a system for flexible parallel execution. PLDI `12.

- Explicit Control Flow

- Explicit Data Dependences

EXPLICIT CONTROL FLOW



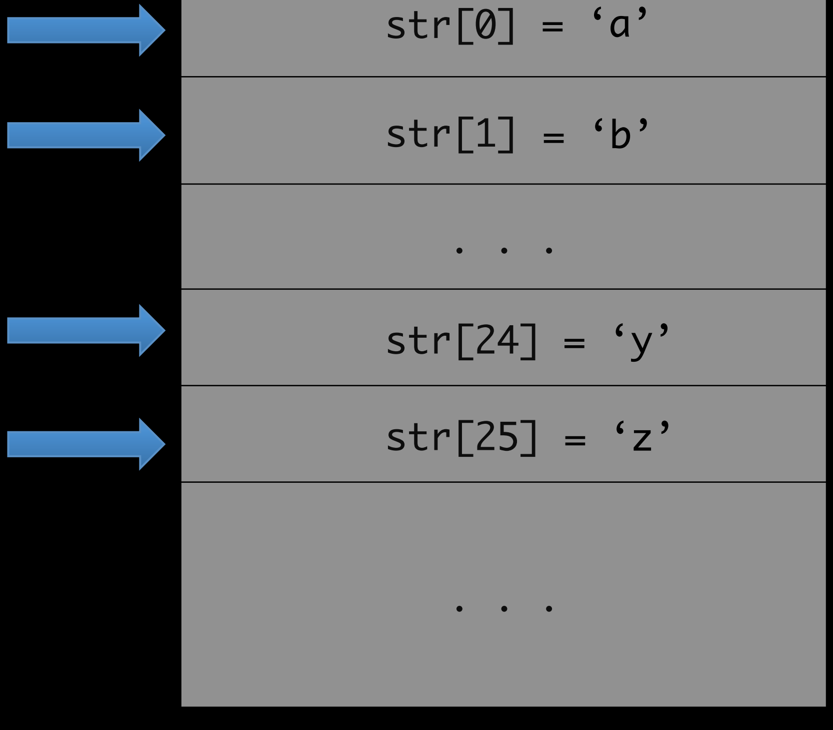

```
void Callee(){
    . . .
    char str[26];
    gets(str);
    . . .
    return;
}
```

Caller: . . .
call Callee

Callee: . . .
return

Ret: . . .
. . malicious .

mal: . . .
. . .




```
void Callee(){
    . . .
    char str[26];
    gets(str);
    . . .
    return;
}
```

Caller: . . .
cblk Callee
. . .
. . malicious .

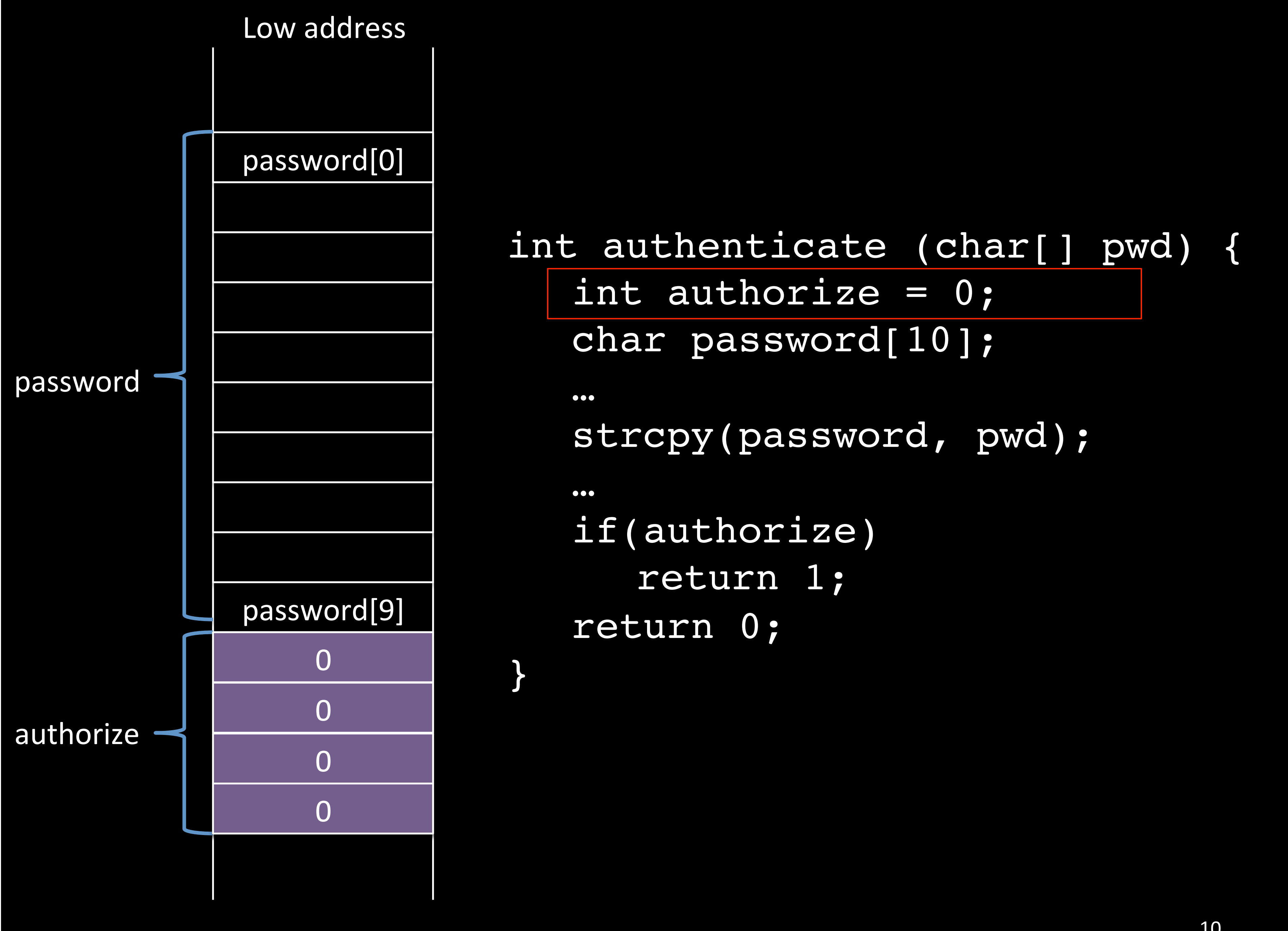
Callee: . . .
. . .

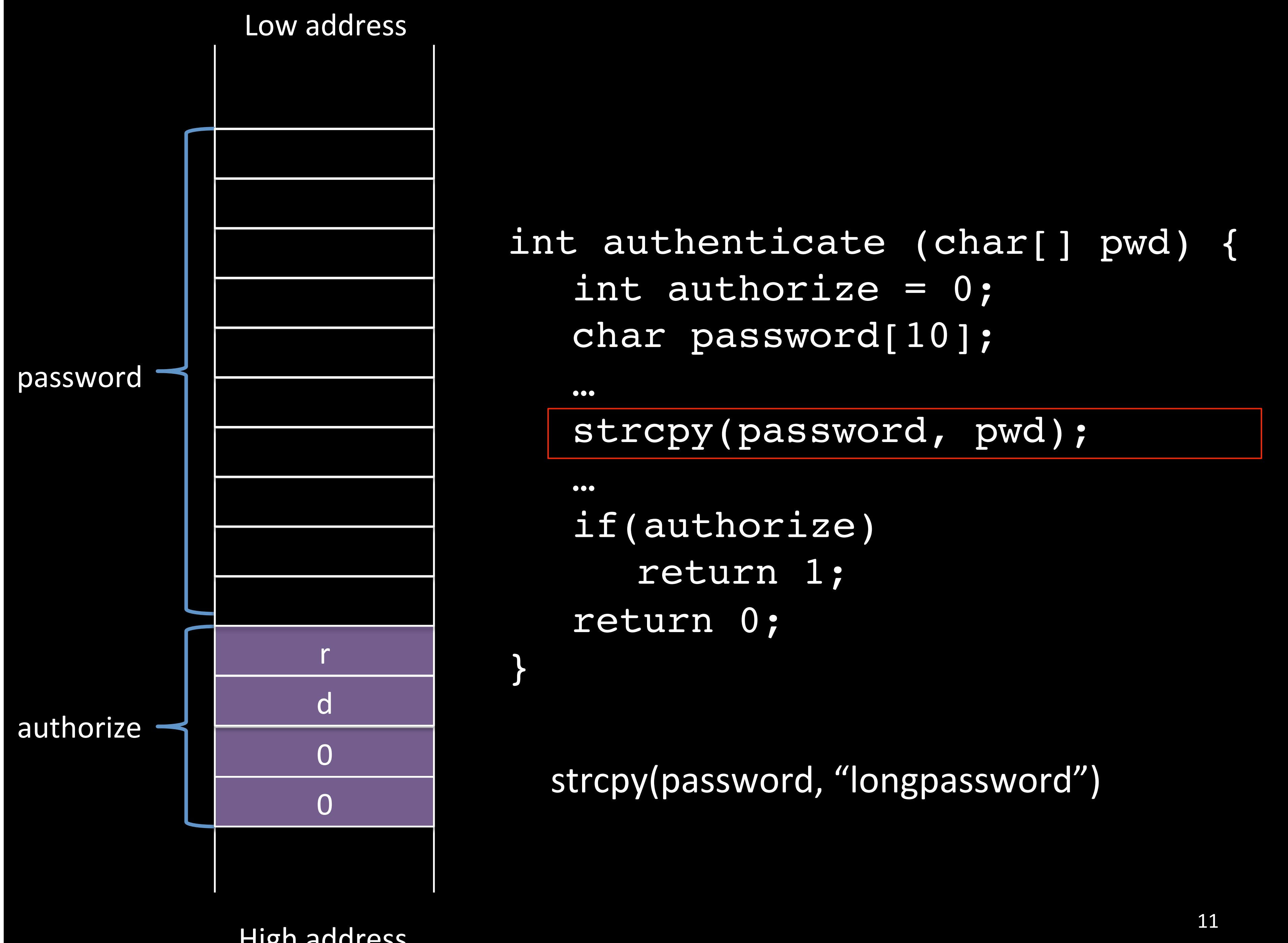
mal: . . .
. . .

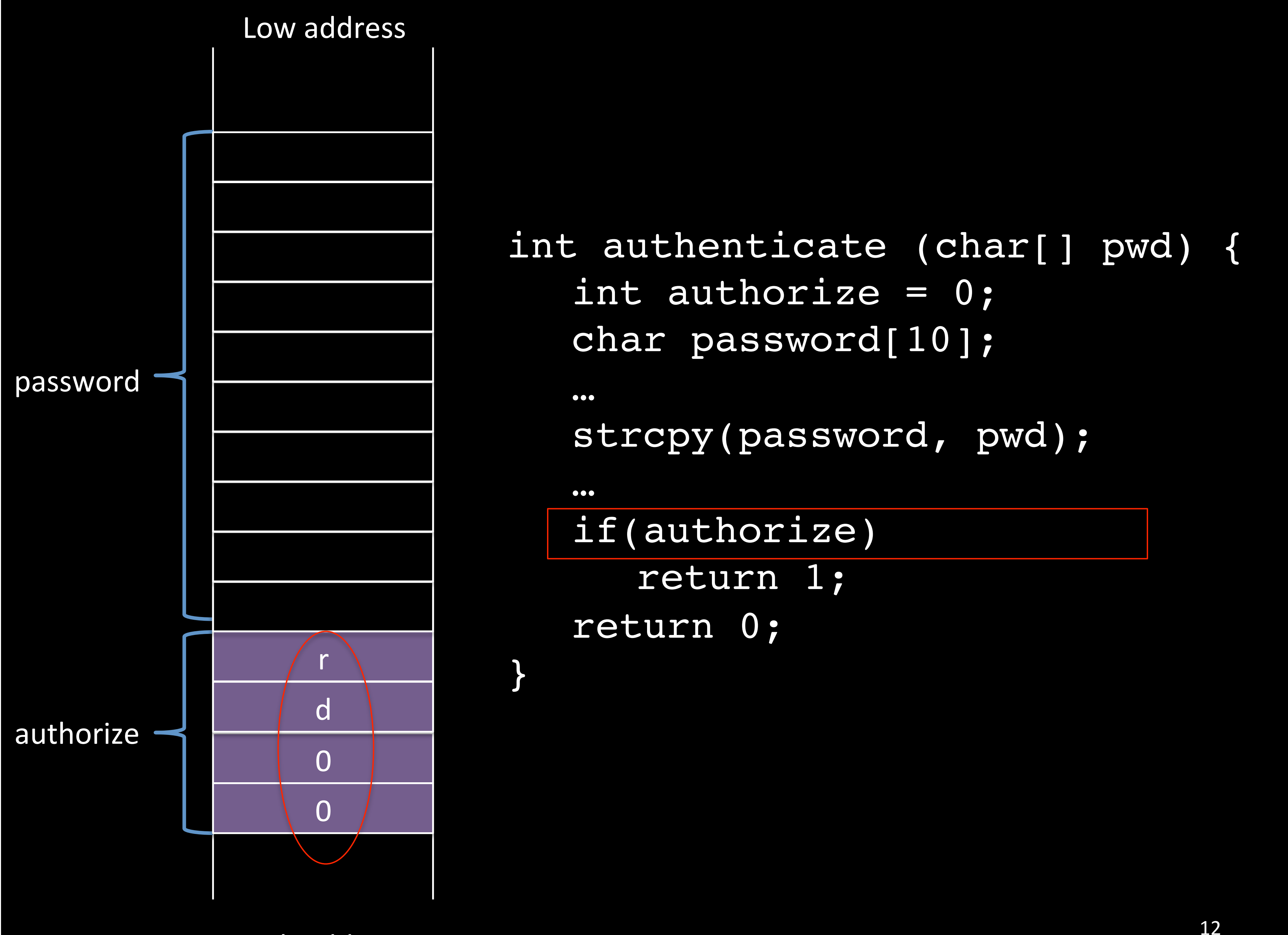
	Constrains Control Flow to CFG	Protects Data Integrity	Separate Protected Return Stack	No Software Runtime Overhead	Code Organized Into Parallelizable Blocks
Liberty	Yes	Yes	Yes	Yes	Yes
Mondrix [1]	No	Yes	Yes	No	No
CFI [2]	Yes	No	No	No	No
SFI [3]	Yes	Yes	No	No	No
XFI [4]	Yes	Yes	Yes	No	No
XFI-HW [5]	Yes	Yes	Yes	Yes	No
Minos [6]	Yes	Yes	No	Yes	No
TRIPS [7]	No	No	No	Yes	Yes

1. Emmett Witchel, et al. Mondrix: memory isolation for linux using mondriaan memory protection. SOSP '05.
2. Martin Abadi, et al. Control-flow integrity. CCS '05.
3. Mihai Budiu, et al. Architectural support for software-based protection. ASID '06.
4. Úlfar Erlingsson, et al. XFI: software guards for system address spaces. OSDI '06.
5. David Sehr, et al. Adapting software fault isolation to contemporary CPU architectures. In Proceedings of the 19th USENIX conference on Security. USENIX Security'10.
6. Jedidiah R. Crandall, et al. Minos: Control Data Attack Prevention Orthogonal to Memory Model. MICRO '04.
7. Aaron Smith, et al. Compiling for EDGE Architectures. CGO '06.

NON-CONTROL DATA ATTACKS AND WRITE INTEGRITY


Low address
password[0]
password[9]
password
0
0
0
0
authorize
High address
int authenticate (char[] pwd) {
int authorize = 0;
char password[10];
…
strcpy(password, pwd);
…
if(authorize)
return 1;
return 0;
}
10


Low address
password
authorize
r
d
0
0
High address
int authenticate (char[] pwd) {
int authorize = 0;
char password[10];
…
strcpy(password, pwd);
…
if(authorize)
return 1;
return 0;
}
strcpy(password, “longpassword”)


Low address
password
authorize
r
d
0
0
High address
int authenticate (char[] pwd) {
int authorize = 0;
char password[10];
…
strcpy(password, pwd);
…
if(authorize)
return 1;
return 0;
}

C Standard

The C standard specifies that the following behaviors are undefined:

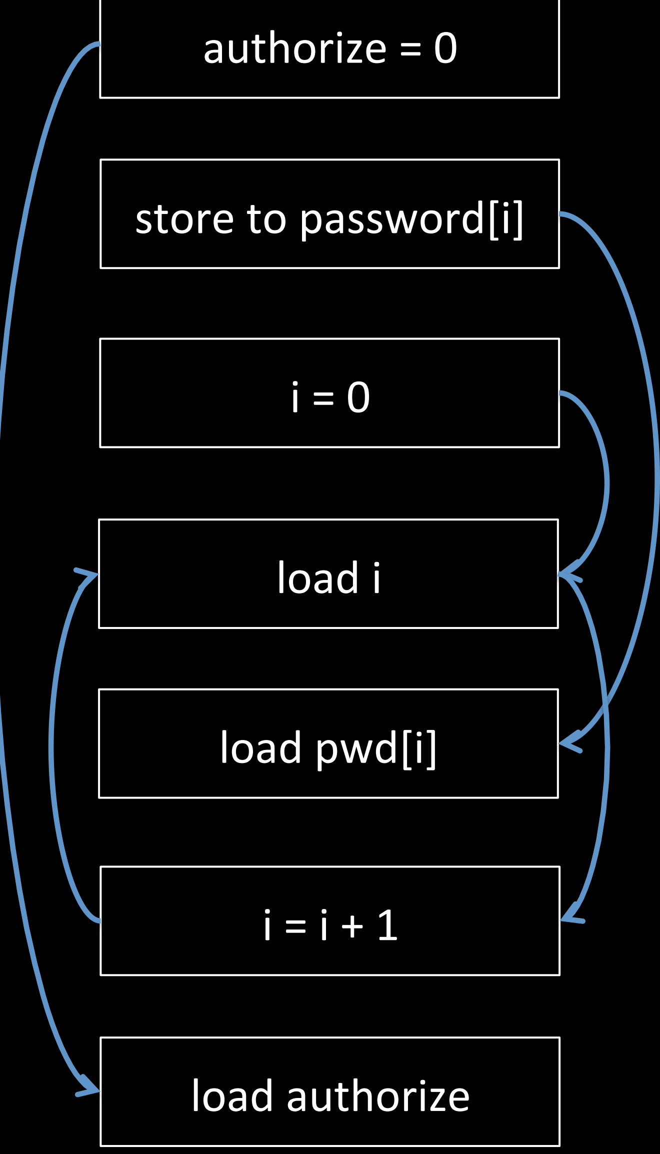
- Pointer arithmetic more than one byte beyond allocation unit bounds
- Accessing beyond bounds of allocation units
- Pointers before first byte of allocation unit
- Dereferencing undefined pointer values
- Accessing the value of an uninitialized variable

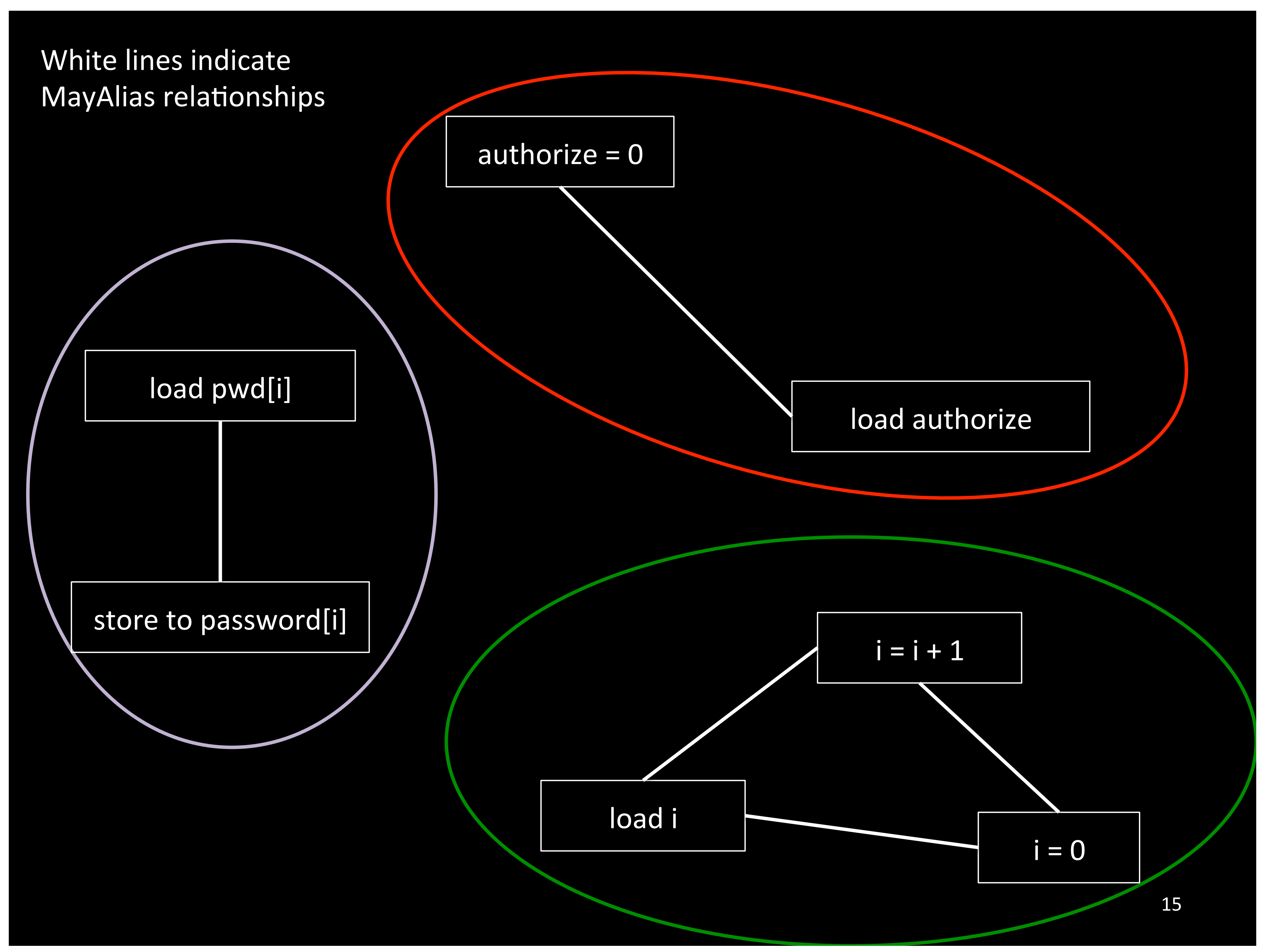
* includes all C standards since C89

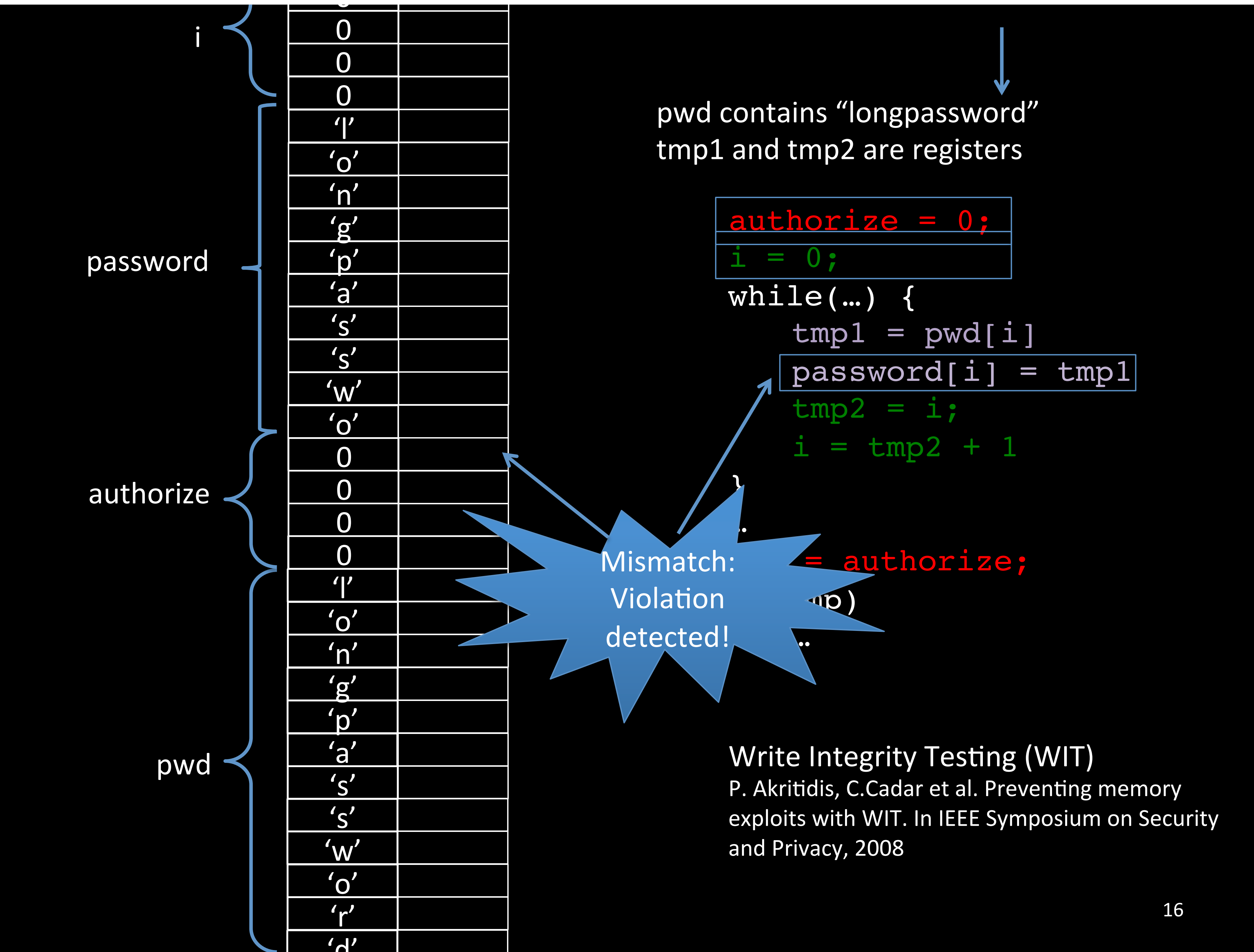
```
int authenticate (char[] pwd) {
    int authorize = 0;
    char password[10];
    …
    int i=0;
    while(password[i] = pwd[i])
        i++;
    …
    if(authorize)
        return 1;
    return 0;
}
```

Blue lines indicate memory dependences

Encode dependence information into the ISA. Hardware only allows statically detected dependences to manifest.




White lines indicate
MayAlias relationships
authorize = 0
load authorize
load pwd[i]
store to password[i]
i = i + 1
load i
i = 0


i
0
0
0
‘l’
‘o’
‘n’
‘g’
‘p’
‘a’
‘s’
‘s’
‘w’
‘o’
password
0
0
0
0
authorize
‘l’
‘o’
‘n’
‘g’
‘p’
‘a’
‘s’
‘s’
‘w’
‘o’
‘r’
‘d’
pwd
pwd contains “longpassword”
tmp1 and tmp2 are registers
authorize = 0;
i = 0;
while(…) {
tmp1 = pwd[i]
password[i] = tmp1
tmp2 = i;
i = tmp2 + 1
= authorize;
Mismatch: Violation detected!
Write Integrity Testing (WIT)
P. Akritidis, C.Cadar et al. Preventing memory exploits with WIT. In IEEE Symposium on Security and Privacy, 2008

Vulnerability Coverage

- Provides partial memory safety
- Memory safety errors detected include:
  - Buffer overruns
  - Dangling pointer references (Use after free)
  - Wild pointer accesses
- Store + Load checks also prevent leakage of confidential information
- Liberty Architecture implements WIT in hardware

Load Instruction Semantics

load addr, region

Page Table Entry (PTE) Check

- pte ← Page Table entry for addr
- Is pte→read true?

Region Check

- (Instruction.region== addr.region) ? load : fail

Load

- currReg ← Mem[addr]

Store Instruction Semantics

store val, addr, region

Page Table Entry (PTE) Check

- pte ← Page Table Entry for addr
- Is pte→write true?

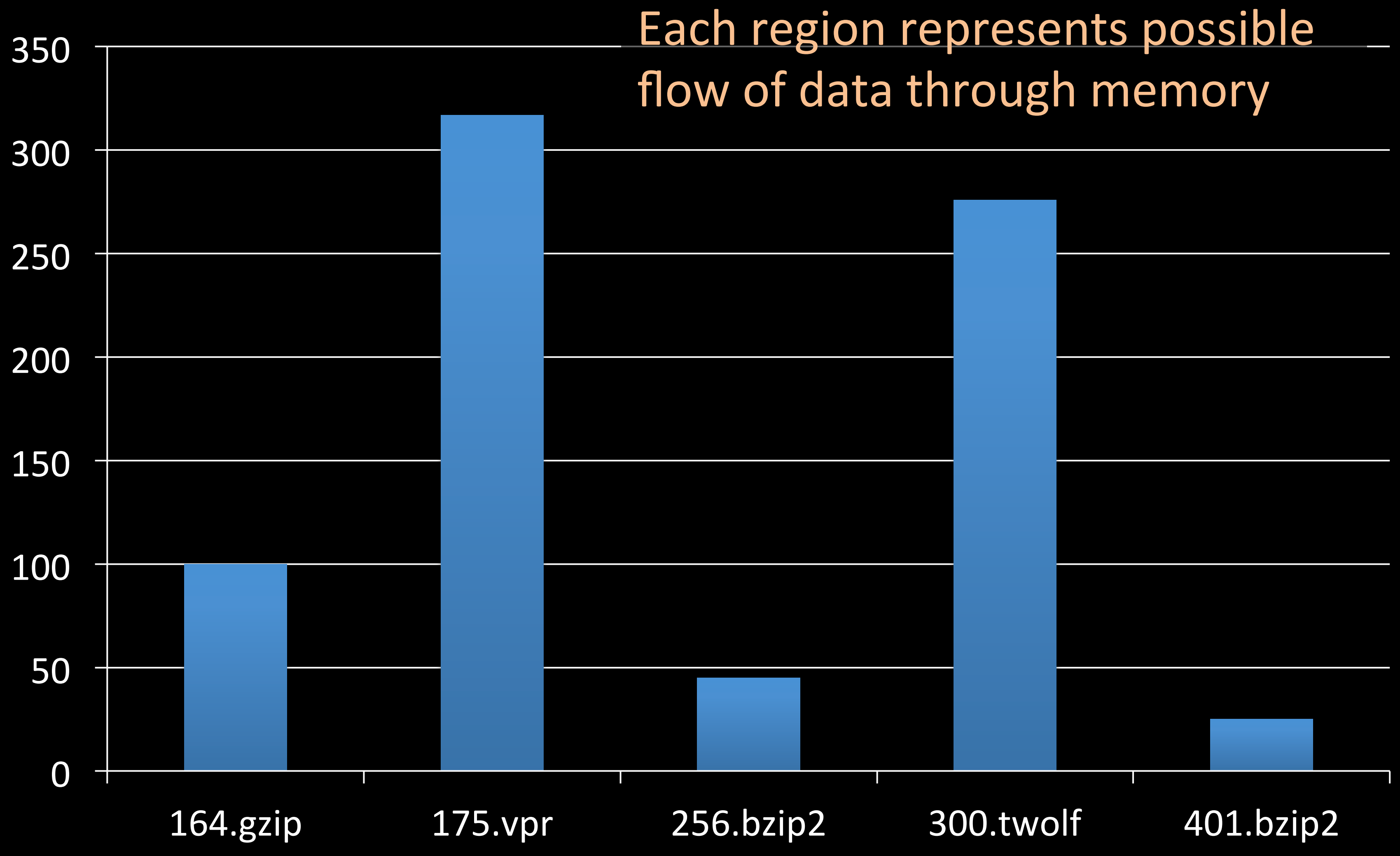
Region Check

- (Instruction.region == addr.region) ? store: fail

Store

- Mem[addr] ← value

Number of Regions Detected



Liberty Architecture Status

- Design
  - Initial ISA complete
  - Second iteration of design in progress
- Technology
  - Compiler
    - Analysis pass complete
    - Code Generation in progress
  - Assembler complete
  - Linker complete
  - ISA Functional Simulator complete
  - ISA Timing Simulator in progress
  - Dynamic Optimizer in progress

Questions?

SPARCHS Architecture

Revision 0.1

Principles

- Symbiosis
- Diversity
- Unpredictable execution
- Continuous learning
- Repair and Recovery

Talks today

- Diversity
  - Instruction Set Randomization
- Unpredictability
  - Kbouncer
- Learning
  - Heterogeneous Information Flow Tracking
  - Explicit dependence encoding
- Repair
  - Autotomic Binary Structure Randomization

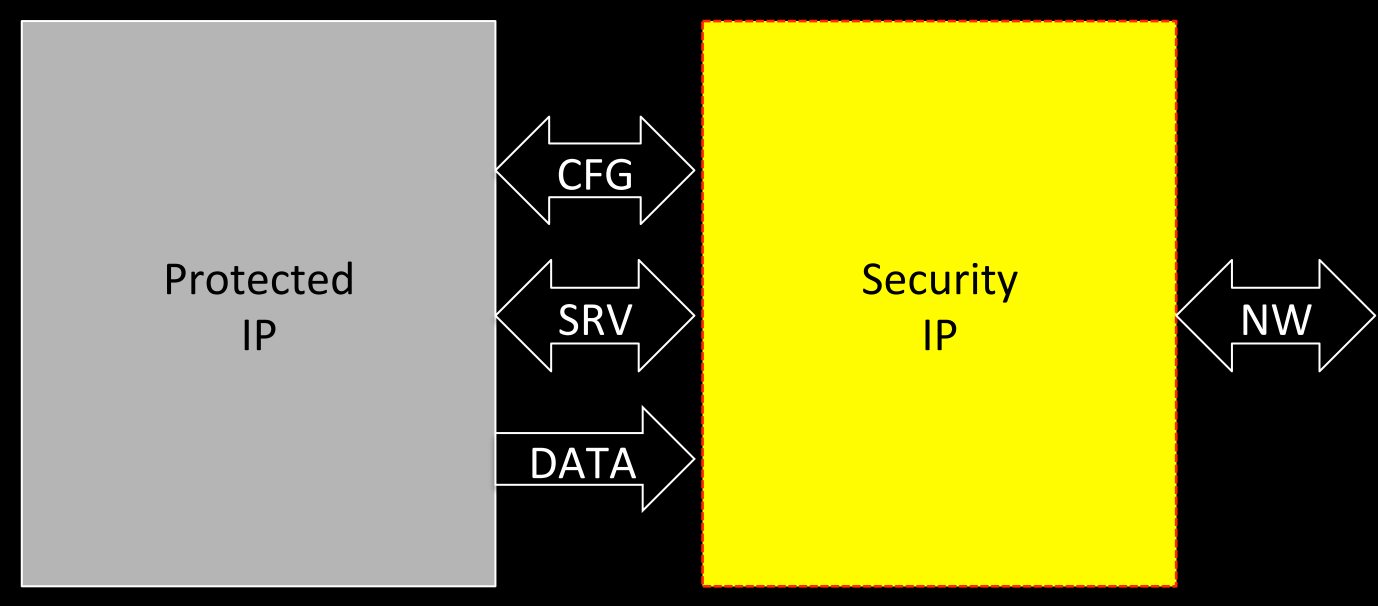
Prior Results

- Diversity
  - Instruction Set Randomization
  - *Symbiotes*
- Unpredictability
  - Kbouncer, *Time Warp*
- Learning
  - Heterogeneous/*Regular* Information Flow Tracking,
  - Explicit dependence encoding
  - *Enhancing Hardware Performance Counters*
- Repair
  - Autotomic Binary Structure Randomization
  - *REASSURE*

Technology Transition

- Industry prefers:
  - Microarchitectural changes
  - Simple architectural extensions
  - Clean system organization

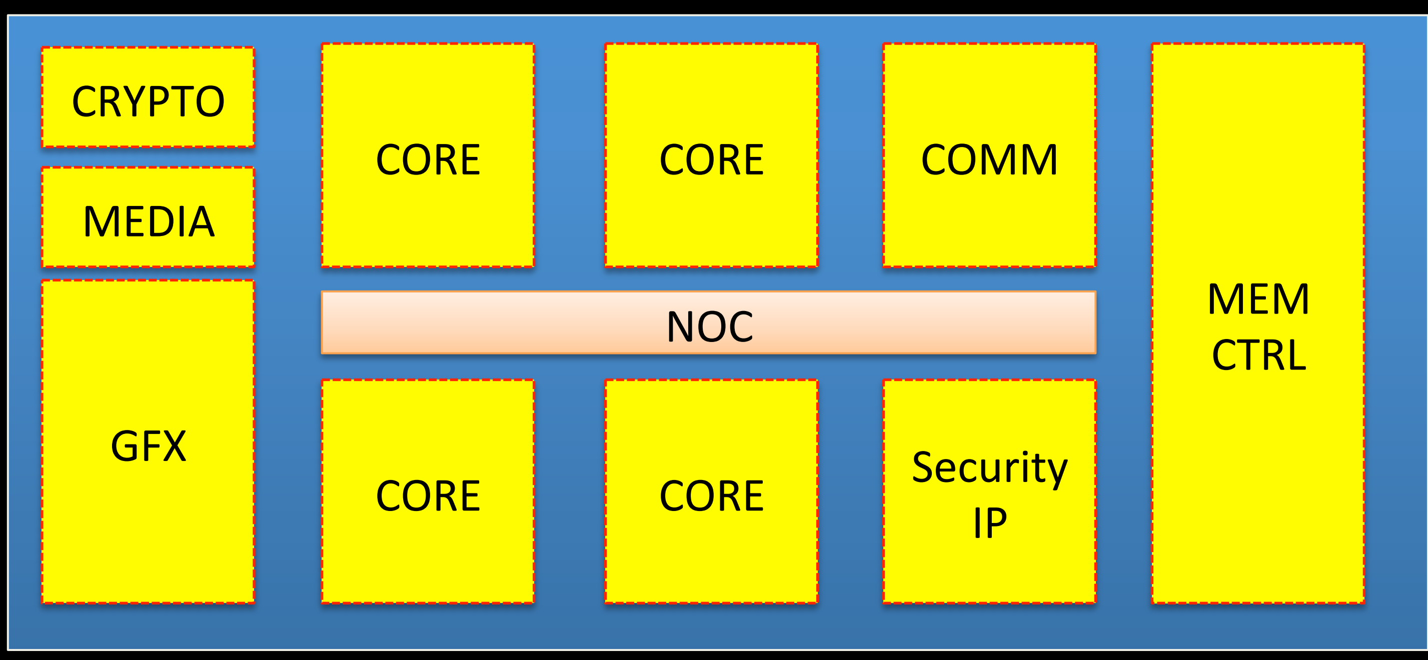
Architecture Security Standard



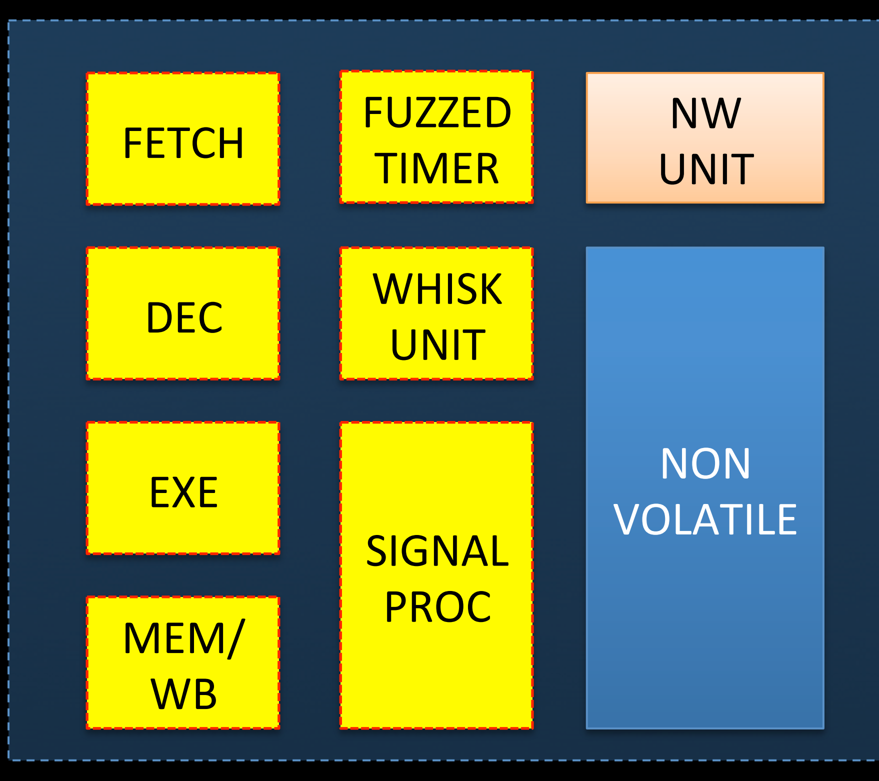
Security Services

Name	Functionality	Interface
sTimer	Strong Isolation (mitigates side channels)	REQ
ISR	Key storage for ISR	CFG
WHISK	Tracking information flow	CFG, Data
Malware	Anomaly detection, Learning	CFG, Data, NW
Watch	Transfer CTRL to Secure IP (dynamic checking)	CFG, Data
Recovery	Checkpoint, recovery and restore	Data

System Architecture



Security IP Microarchitecture



Observations so far

- Lack of (strong) security architecture
  - HP RFU
- Bad implementations are prevalent
  - Sony PS
- Systems are configured incorrectly
  - Router
- System update functions are always suspect!
  - Convenient updating is also convenient attack surface

Tech Transfer

- Red Balloon Security
- MS Prize
- Talks at IBM, AMD
  - Planned visit

Status: Y1 and Y2 Tasks

#	Task	Completion
1	Initial design of SPARCHS	Comlplete
2	Develop concurrency Attacks	Complete
3	Polymorphic Microarchitectures	90%
4	OS-based self-healing	Complete
6	Optimized Information Flow Tracking (Compiler)	30%
7	Injection-based Symbiote	Complete
8	ISR extensions	Complete
9	Dynamic Diversified Replicas - Initial Design	Complete
10	Hardware support for ISR	Complete
11	Emulation evaluation OS-based self healing	90%
12	Initial hardware support for healing	Forthcoming
13	ISR supported Symbiote	Forthcoming
14	Defenses against concurrency attacks	50%
15	SPARCHS architecture and design	25%

Publications (Y2)

- [1] Side-Channel Vulnerability Factor: A Metric for Measuring Information Leakage
- John Demme, Robert Martin, Adam Waksman and Simha Sethumadhavan,
- Proceedings of the 39th ACM/IEEE International Symposium
- on Computer Architecture (ISCA), 2012, Portland, OR, USA. (Acceptance
- rate: 18%)

- [2] Time Warp: Rethinking Timekeeping and Performance Measurement Mechanisms to Mitigate Side Channels
- Robert Martin, John Demme and Simha Sethumadhavan,
- Proceedings of the 39th ACM/IEEE International Symposium on Computer
- Architecture (ISCA), 2012, Portland, OR, USA. (Acceptance rate: 18\%)

- [3] Rapid Identification of Architectural Bottlenecks via Precise Event Counting
- John Demme and Simha Sethumadhavan,
- Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture (ISCA)}, 2011, San Jose, CA, USA. (Acceptance rate: 19\%)

- [4] kGuard: Lightweight Kernel Protection against Return-to-user Attacks
- Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. To appear in the Proceedings of the 21st USENIX Security Symposium. August 2012, Bellevue, WA. (Acceptance rate: 19.4%)

- [5] Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization"
- Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. In Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P), pp. 601 - 615. May 2012, San Francisco, CA. (Acceptance rate: 13%)

Publications (Y2)

- [6] A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware
- Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. In Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed Systems Security (SNDSS). February 2012, San Diego, CA. (Acceptance rate: 17.8%)

- [7] ROP Payload Detection Using Speculative Code Execution
- Michalis Polychronakis and Angelos D. Keromytis. In Proceedings of the 6th International Conference on Malicious and Unwanted Software (MALWARE), pp. 58 - 65. October 2011, Fajardo, PR. (Best Paper Award)

- [8] Junfeng Yang, Ang Cui, Salvatore J. Stolfo, Simha Sethumadhavan; "Concurrency Attacks;" the Fourth USENIX Workshop on Hot Topics in Parallelism; 2012/06/07.

- [9] Ang Cui, Jatin Kataria, Salvatore J. Stolfo; "From Prey To Hunter: Transforming Legacy Embedded Devices Into Exploitation Sensor Grids;" The 27th Annual Computer Security Applications Conference (ACSAC); 2011/12/05.

- [10] Ang Cui, Salvatore J. Stolfo; "Defending Legacy Embedded Systems with Software Symbiotes;" The 14th International Symposium on Recent Advances in Intrusion Detection (RAID); 2011/09/20

- [11] Ang Cui, Salvatore J. Stolfo, Jatin Kataria; "Killing the Myth of Cisco IOS Diversity: Towards Reliable, Large-Scale Exploitation of Cisco IOS;" 5th USENIX Workshop on Offensive Technologies (WOOT); 2011/08/08

- [12] Runtime Asynchronous Fault Tolerance via Speculation Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Proceedings of the 2012 International Symposium on Code Generation and Optimization (CGO), April 2012.

- [13] A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed Systems Security (NDSS), February 2012.

- [14] Speculative Separation for Privatization and Reductions, Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2012.