Aalto University

UNIVERSITY OF WATERLOO

Hardware-assisted Run-time Protection

On balancing security and deployability

N. Asokan

https://asokan.org/asokan/

@nasokan

Acknowledgements: Thomas Nyman, Hans Liljestrand, Lachlan J Gunn, Jan-Erik Ekberg

How to thwart run-time attacks?

Run-time attacks are now routine

Software defenses incur security vs. cost tradeoffs

Hardware-assisted defenses are attractive

- **Two case studies: HardScope and PARTS/PACStack**

Current/future directions

Memory-related run-time attacks

Software written in memory unsafe languages such as C/C++

- Suffer from various memory-related errors

Memory errors may allow run-time attacks to compromise program behaviour

- *Control-flow hijacking* / *code injection*
- *Return-Oriented Programming* (ROP)
- *Non-control-data attacks*
- *Data-Oriented Programming* (DOP)

Run-time attacks compromise program behaviour

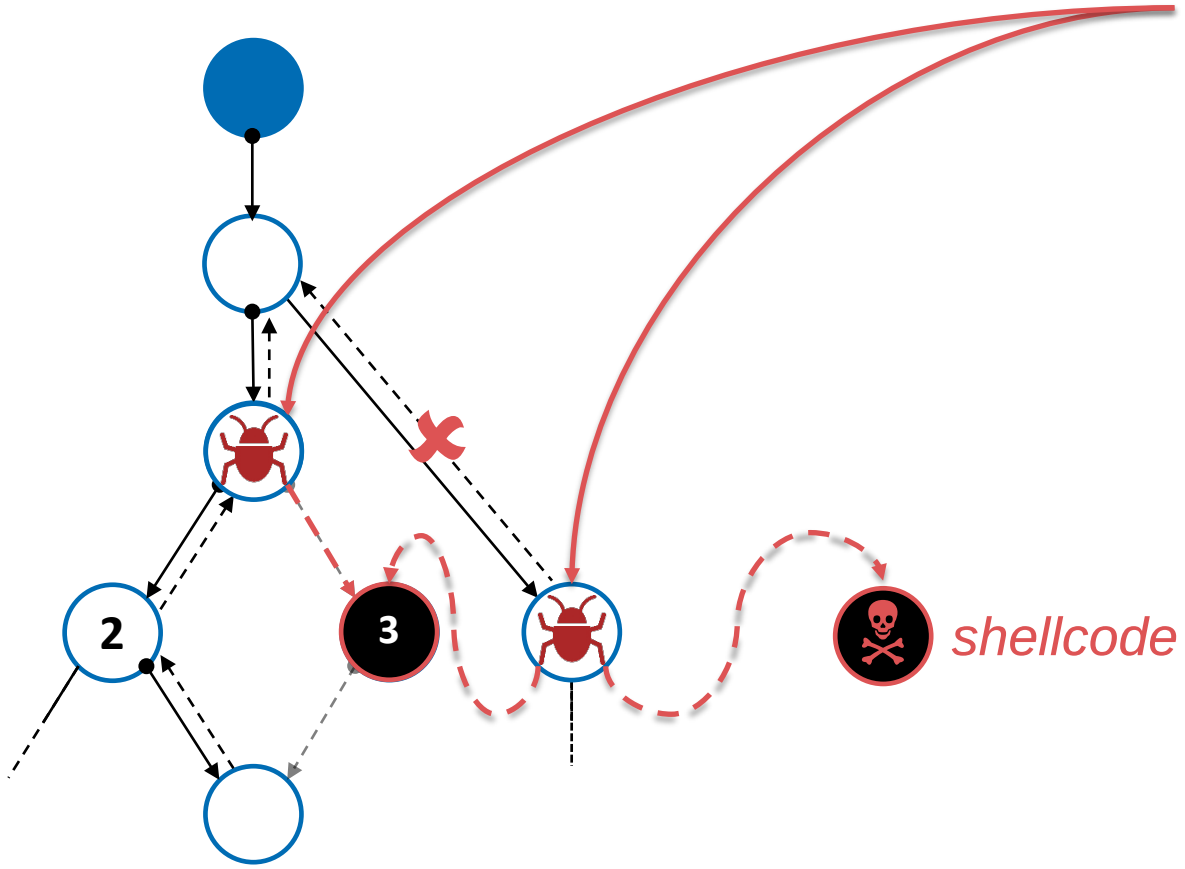
(i) Code-injection attack

(ii) Code-reuse attack

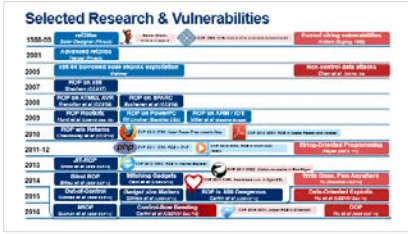
(iii) Non-control-data attack

```
① if (authenticated != true)
     then: call unprivileged()
     else: call   privileged()
…

② unprivileged() { … }
③   privileged() { … }
…
```



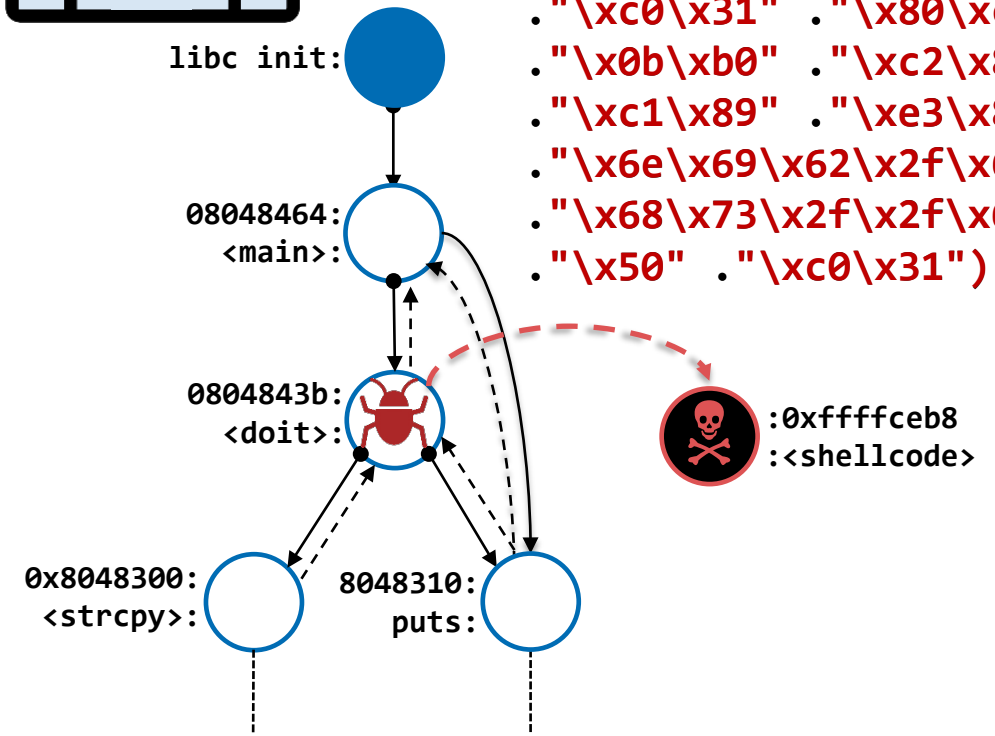
Adversary exploits bug



Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013)

Classic code-injection

```
$ ./a.out $(perl -e 'print "A"x8 \
        ."\x??\x??\x04\x08" \
        ."A"x4 \
        ."\xb8\xce\xff\xfb" \
        ."\x80\xcd" ."\x40" \
        ."\xc0\x31" ."\x80\xcd" \
        ."\x0b\xb0" ."\xc2\x89" \
        ."\xc1\x89" ."\xe3\x89" \
        ."\x6e\x69\x62\x2f\x68" \
        ."\x68\x73\x2f\x2f\x68" \
        ."\x50" ."\xc0\x31")
```

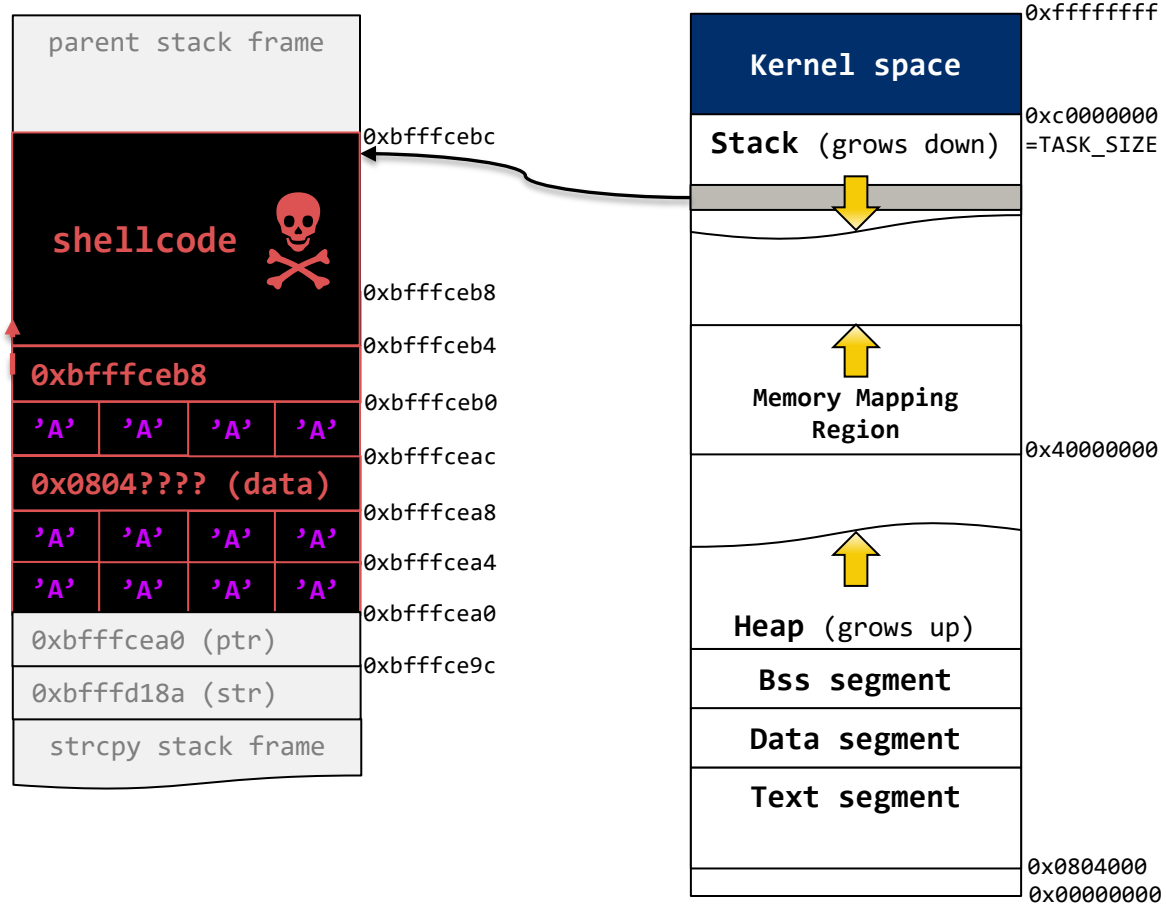



```c
int main(int argc, char *argv[])
{
    puts("So... The End...");
    doit(argv[1]);
    puts("or... maybe not?");

    return 0;
}
```

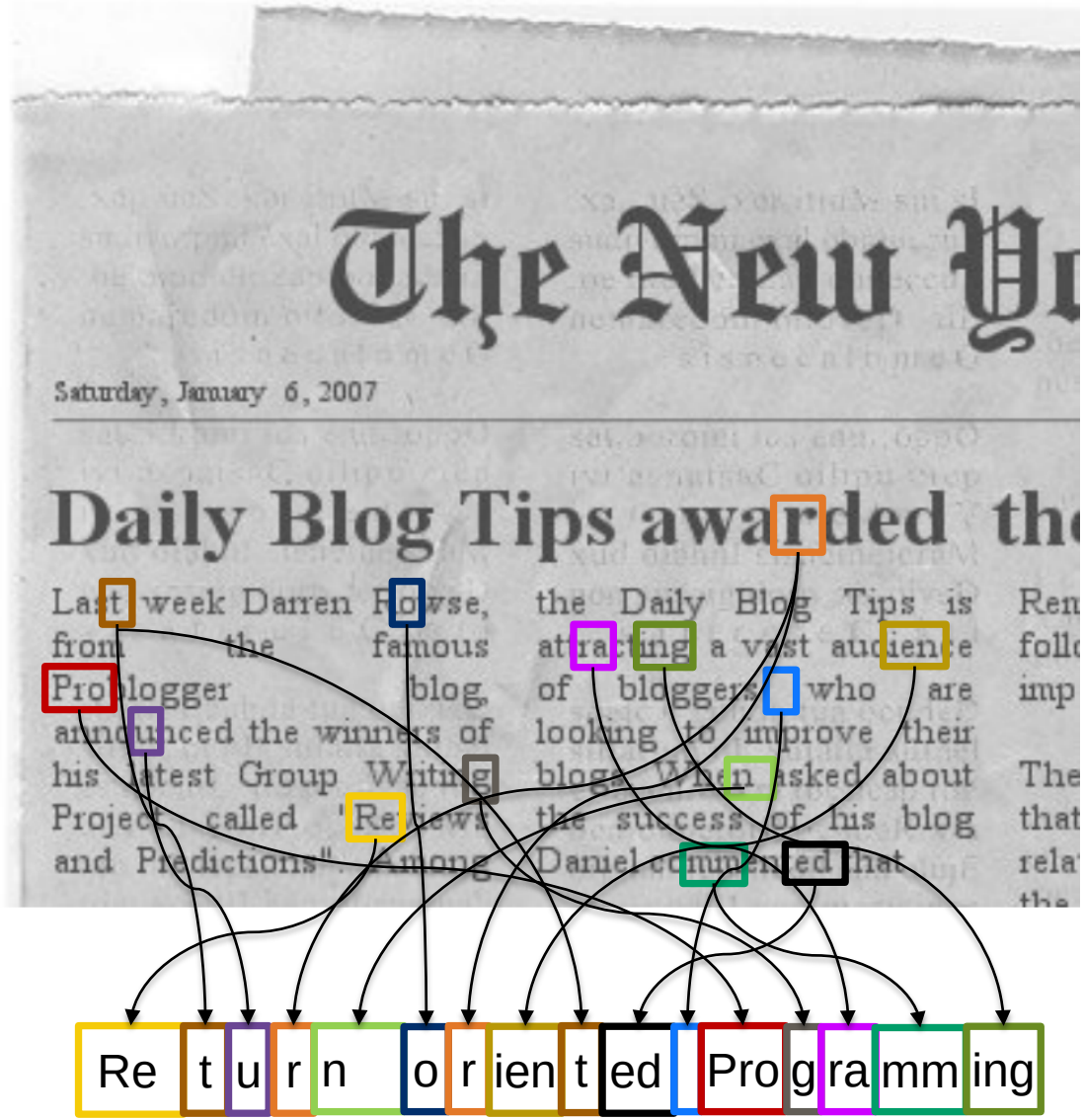
```c
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```



Elias Levy (as *Aleph One*), Smashing the stack for fun and profit, Phrack 7 (1996)

Return-oriented programming (high-level idea)

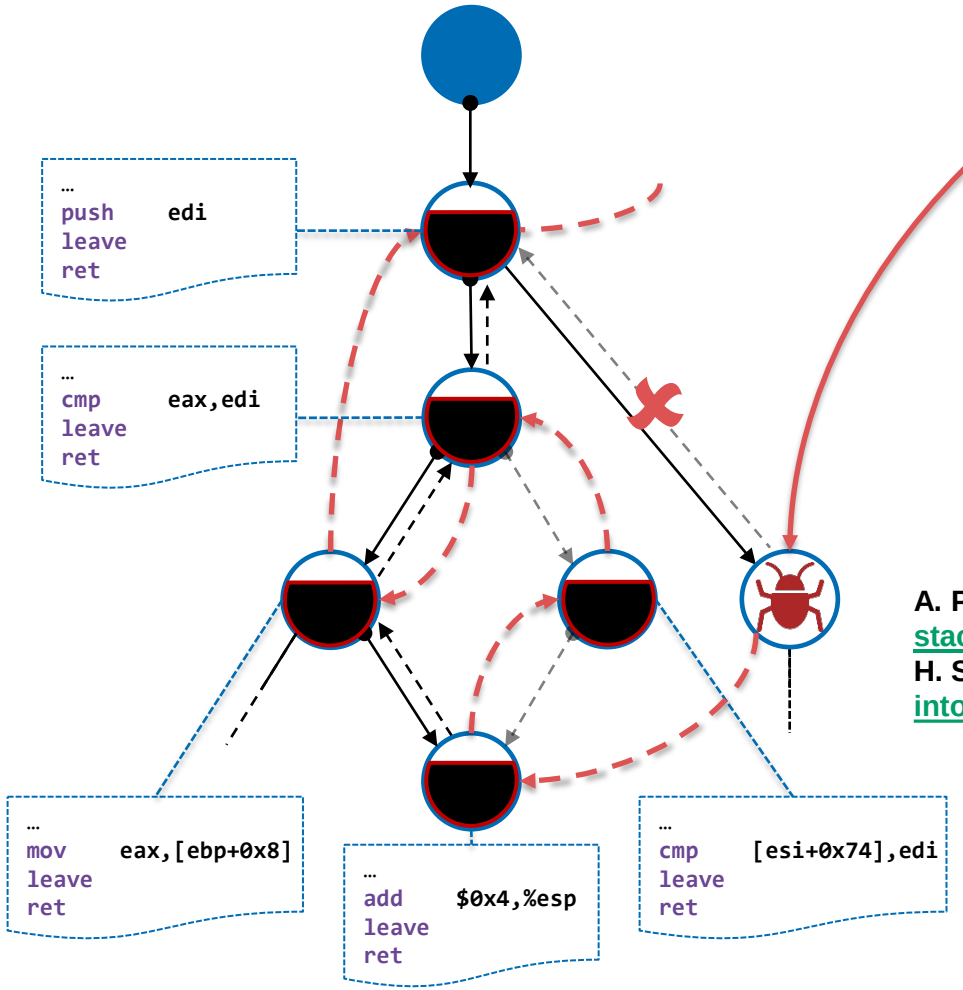


Return-oriented programming

Attacker arranges call stack with code pointers to existing code sequences (“*gadgets*”)

- **Given a suitable gadget set, arbitrary *return-oriented programs* can be constructed**

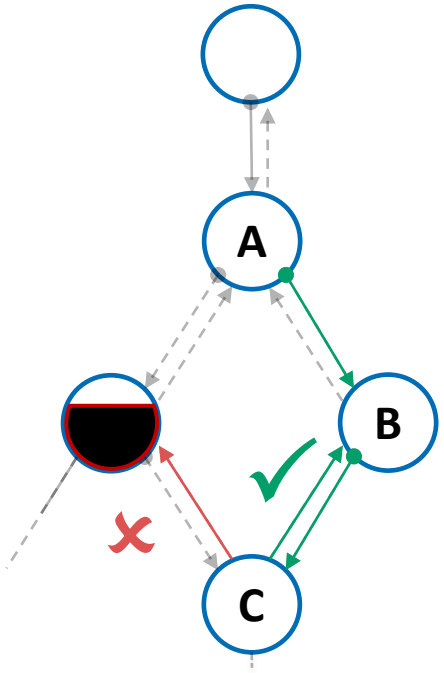
Adversary exploits bug

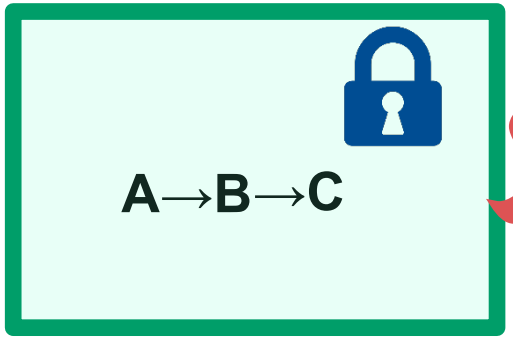


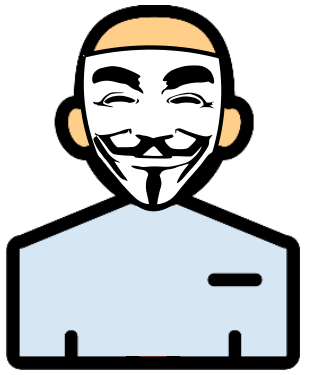
A. Peslyak (as *Solar Designer*), Getting around non-executable stack (and fix), Bugtraq (1997)

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

Shadow Stack: High-level idea







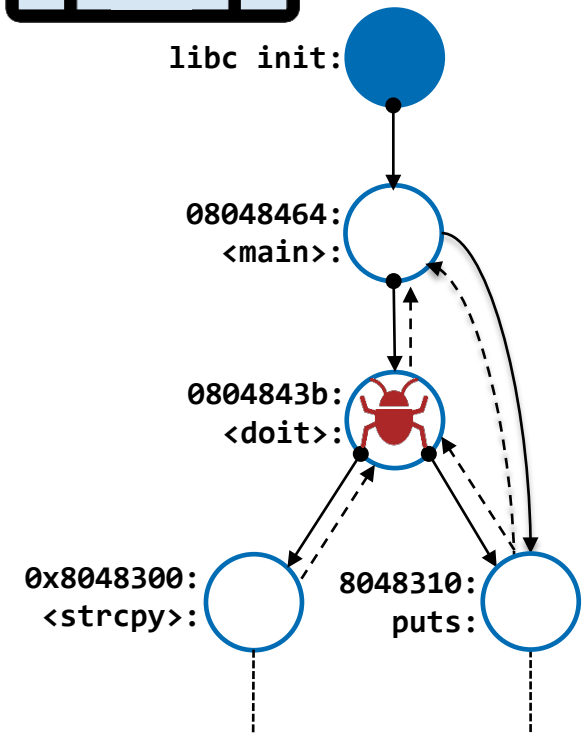
Adversary tampers with shadow stack

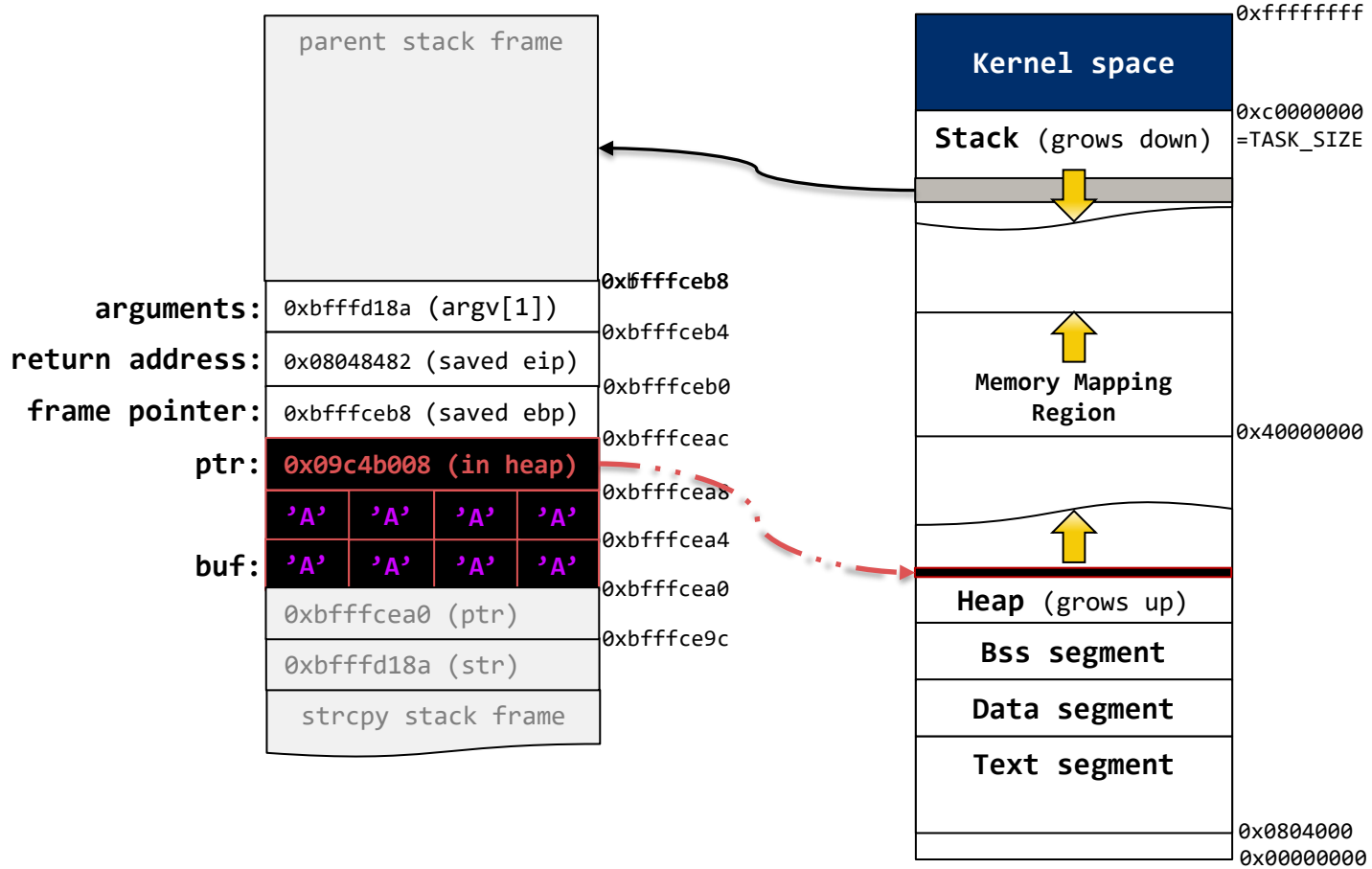
Non-control data attack

```
$ ./a.out $(perl -e 'print "A"x8 \
        ."\x08\xb0\xc4\x09" )
```

```
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```





Program logic that can be influenced as result of memory vulnerability constitute "*data-oriented gadgets*"

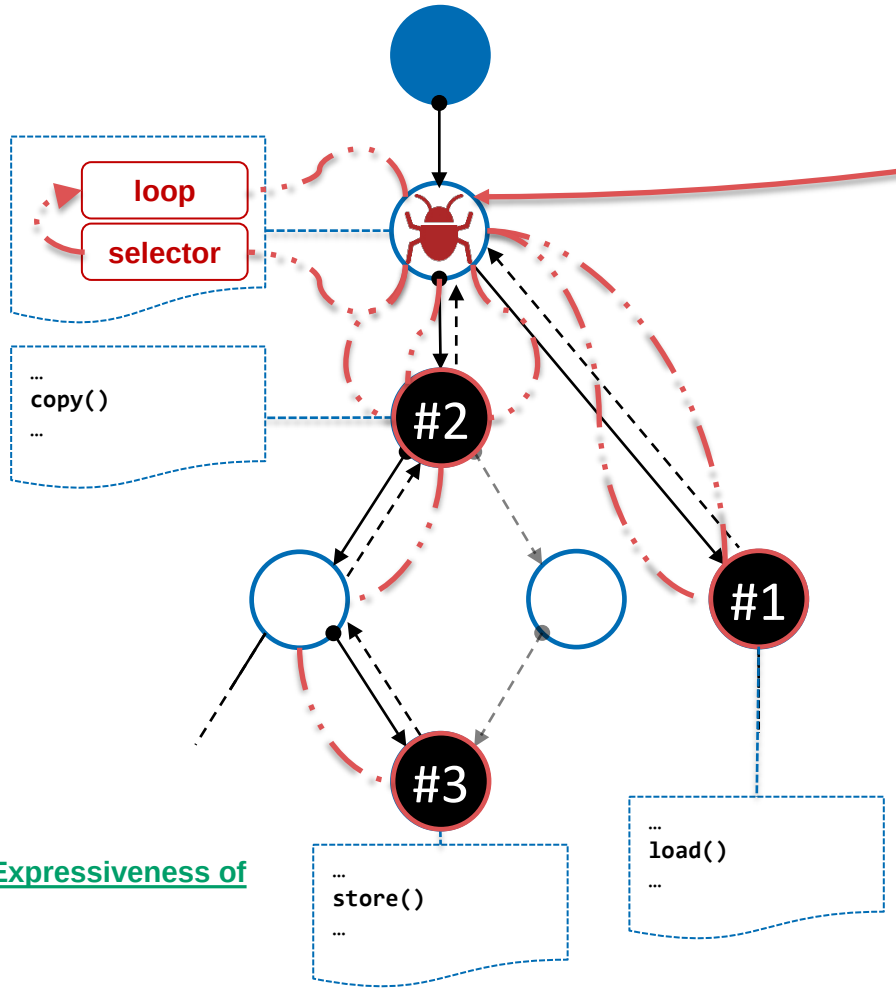
Attacker influences the behavior of benign program code without breaking control-flow integrity

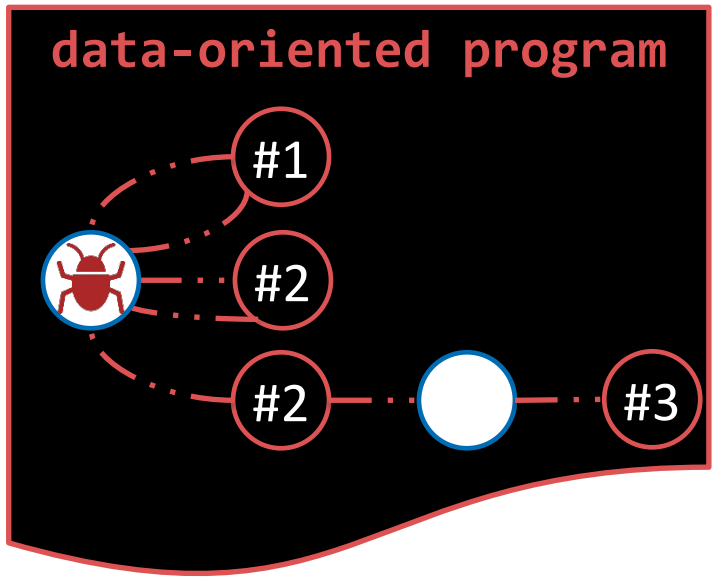
Data-oriented Programming

Enables expressive computation via use of "*data-oriented gadgets*" without diverging from program's benign control-flow

- **Requires a "gadget dispatch" that allows chaining together gadgets at will**

Adversary exploits bug





Hu et.al. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. IEEE S&P, 2016

Selected Research & Vulnerabilities

1988-99
- **ret2libc** — *Solar Designer (Phrack)*
- **Morris Worm:** RCE in `fingerd`
- **CVE-1999-1416:** DoS & RCE in Solaris Answerbook2
- **Format string vulnerabilities** — *Anders (Bugtraq. 1999)*

2001
- **Advanced ret2libc** — *Nergal (Phrack)*

2005
- **x86-64 borrowed code chunks exploitation** — *Krahmer*
- **Non-control-data attacks** — *Chen et al (SSYM. '05)*

2007
- **ROP on x86** — *Shacham (CCS'07)*

2008
- **ROP on ATMEL AVR** — *Francillon et al (CCS'08)*
- **ROP on SPARC** — *Buchanan et al (CCS'08)*

2009
- **ROP Rootkits** — *Hund et al (USENIX Sec. '09)*
- **ROP on PowerPC** — *FX Lindner (BlackHat USA)*
- **ROP on ARM / iOS** — *Miller et al (BlackHat Europe)*

2010
- **ROP w/o Returns** — *Checkoway et al (CCS'10)*
- **CVE-2010-3765**: *Nobel Peace Price website 0day*
- **CVE-2010-2883:** *RCE in Adobe Reader and Acrobat*

2011-12
- **CVE-2011-1938:** *RCE in PHP*
- **CVE-2012-0003:** *RCE in WMP MIDI library*
- **String-Oriented Programming** — *Payer (28C3. '11)*

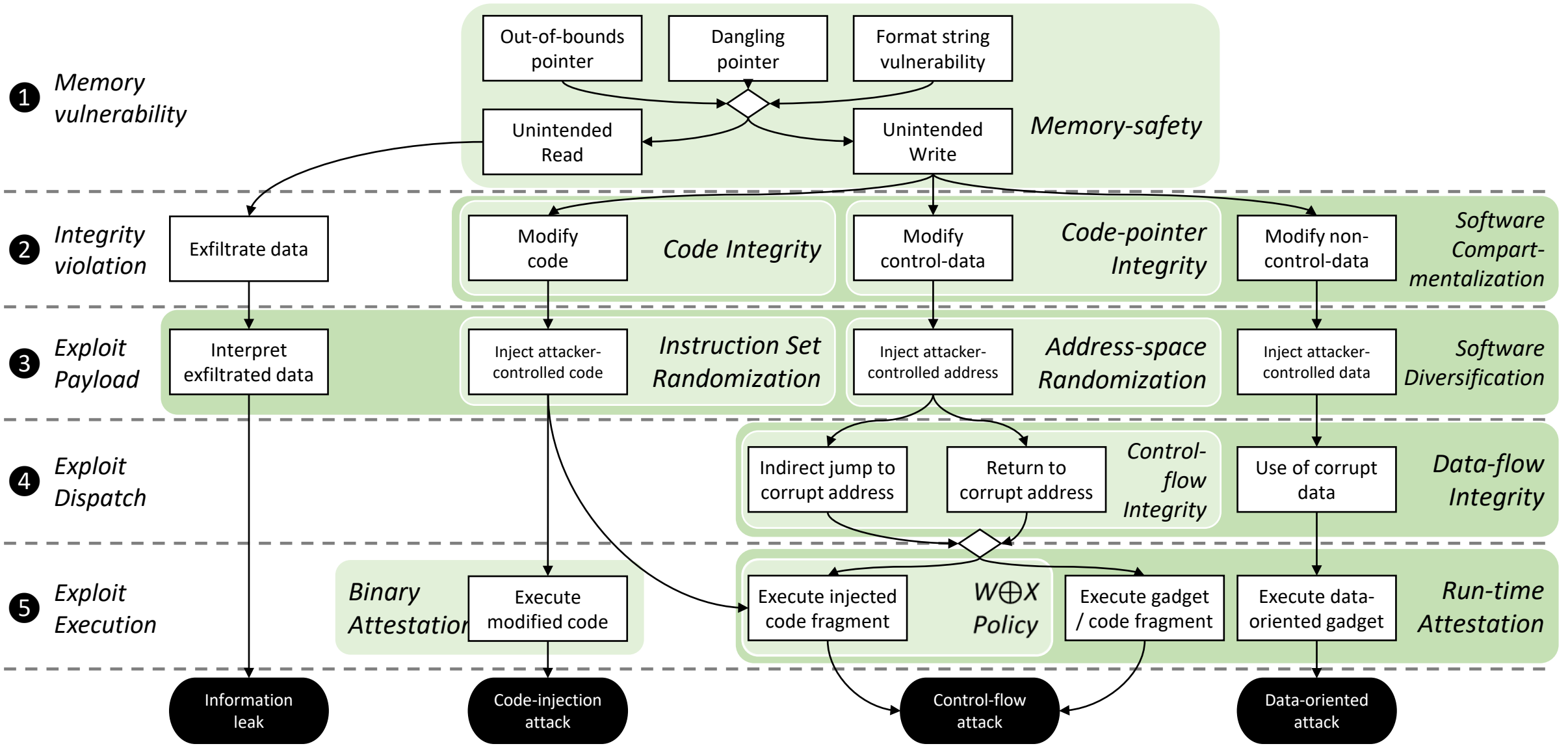
2013
- **JIT-ROP** — *Snow et al (IEEE S&P'13)*
- **CVE-2013-3893:** *RCE in Internet Explorer*
- **CVE-2014-9222:** *Misfortune cookie in RomPager*

2014
- **Blind ROP** — *Bittau et al (IEEE S&P'14)*
- **Stitching Gadgets** — *Davi et al (USENIX'14)*
- **CVE-2014-0160:** *Heartbleed vuln. in OpenSSL*
- **Write Once, Pwn Anywhere** — *Yu (BlackHat USA'14)*

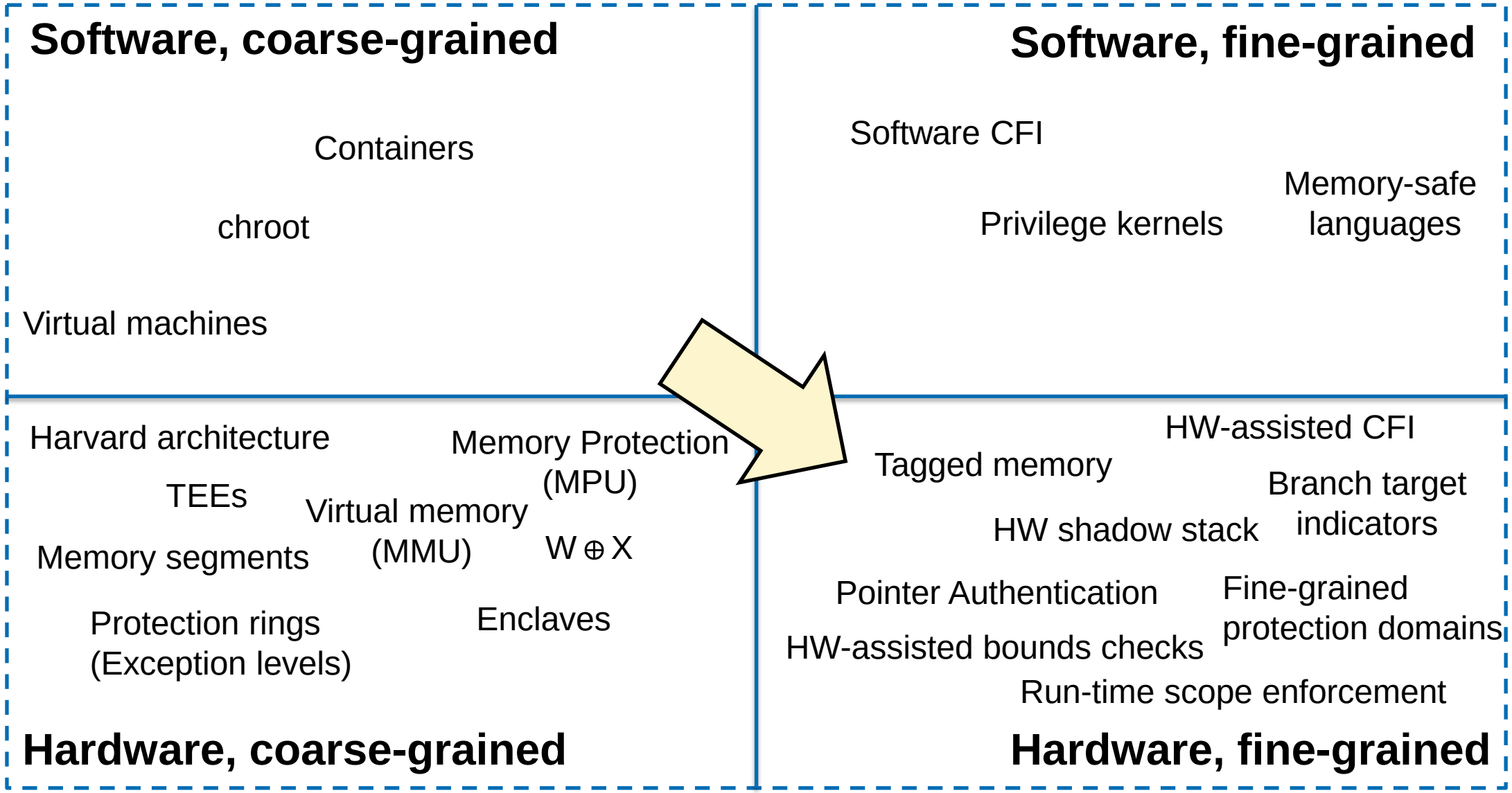
2015
- **Out-of-Control** — *Göktas et al (IEEE S&P'14)*
- ***Gadget size Matters*** — *Göktas et al (USENIX'14)*
- **ROP is Still Dangerous** — *Carlini et al (USENIX'14)*
- **Data-Oriented Exploits** — *Hu et al (USENIX Sec.'15)*

2016
- **SROP** — *Bosman et al (IEEE S&P'14)*
- **Control-flow Bending** — *Carlini et al (USENIX Sec.'16)*
- **CVE-2016-0034:** *Angler RCE in Silverlight*
- **DOP** — *Hu et al (IEEE S&P '16)*

Taxonomy of Defenses



From **Thomas Nyman's doctoral dissertation, Towards Hardware-assisted Run-time Protection, 2020** (Figure Adapted from Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013))


Software, coarse-grained
Containers
chroot
Virtual machines
Software, fine-grained
Software CFI
Memory-safe languages
Privilege kernels
Harvard architecture
TEEs
Memory segments
Protection rings (Exception levels)
Virtual memory (MMU)
Memory Protection (MPU)
W ⊕ X
Enclaves
Hardware, coarse-grained
HW-assisted CFI
Tagged memory
Branch target indicators
HW shadow stack
Pointer Authentication
Fine-grained protection domains
HW-assisted bounds checks
Run-time scope enforcement
Hardware, fine-grained

Protect against run-time attacks without incurring a significant performance penalty

Hardware-assisted run-time protection

Two case studies:

- **HardScope: minimal CPU extensions for hardware-assisted scope enforcement**
- **PARTS and PACStack: Run-time safety using ARM Pointer Authentication**

HardScope

Hardware-assisted Run-time Scope Enforcement

Joint work with TU Darmstadt

Motivation: Run-time Attacks

Memory corruption vulnerabilities in C / C++ can allow an attacker to access to:

- Control-data, e.g. return address stored on call stack (control-flow hijacking)
- Decision-making data, e.g. user id used for authorization decisions (data-oriented attack)
- Sensitive data, e.g. cryptographic keys (information leakage)

Access to unintended data

Compile-time variable visibility rules make references to unintended variables less likely

➔ **Enforcing variable scope also at run-time would reduce potential of memory attacks**

Challenges

Lexical scope only known at compile-time

- In C / C++, variable visibility information not available at run-time

Granularity of enforcement

- Effective compartmentalization requires fine granularity for subjects (code) and objects (data)

Context-sensitive access

- Same code may operate under different set of access rules depending on caller

Pervasiveness

- Efficiently mediate all memory accesses

Design

HardScope: High-level Idea

Instrument program during compilation to:

- Split code up into distinct *execution contexts* (common environment for function or block)
- Associate each execution context with *storage regions* (data memory accessed)

Modify underlying hardware with HardScope instructions to:

- Accumulate rules for storage regions [new storage region instructions]
- Track changes in execution context [new scope block instructions]
- Track dynamic data flows [new data delegation instructions]
- Enforce accesses to storage regions [modified load / store instructions]

New Instructions

During run-time, 7 new instructions configure HardScope-hardware with access rules

- Scope Block instructions mark points of *domain transitions*, e.g. function call / return
- Storage Region (SR) instructions *whitelist memory regions* for current domain, e.g. stack frame
- Delegation instructions gives callee/caller access to SRs e.g. arguments, return values

Mnemonic	Name	Description
sbent	Scope Block ENter	Mark transition into new domain
sbxit	Scope Block eXIT	Mark transition ouf of domain
sradd	Storage Region ADD	Set base and limit for new storage region
srdda	Storage Region ADD (reverse operands)	
srdel	Storage Region DELete	Revoke access to storage region
srdlg	Storage Region DeLeGate	Delegate existing SR to callee / caller
srdsub	Storage Region Delegate SUBregion	Delegate subregion to an existing SR

Storage Region Stack

Stack-oriented storage for accumulated access rules

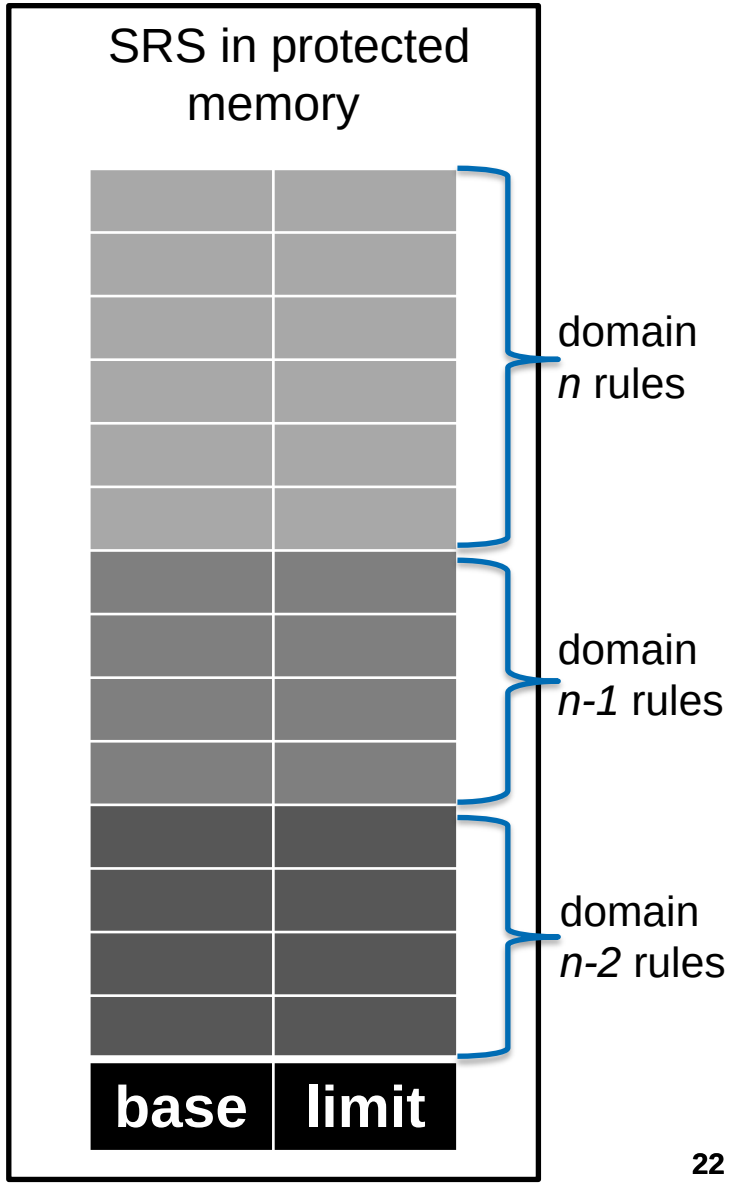
- Stores the bounds of each used *storage region* (e.g. stack variable, heap object, global variable)
- Frames created upon domain entry (`sbent` → push)
- Frames discarded upon domain exit (`sbxit` → pop)

Actively enforced rules in topmost frame

- Memory accesses matched only against active rules
- Subsequent frame store inactive rules for inactive domains
- Function-level enforcement mirrors structure of call stack

Maintained in protected memory

- Rules only modifiable by HardScope instructions

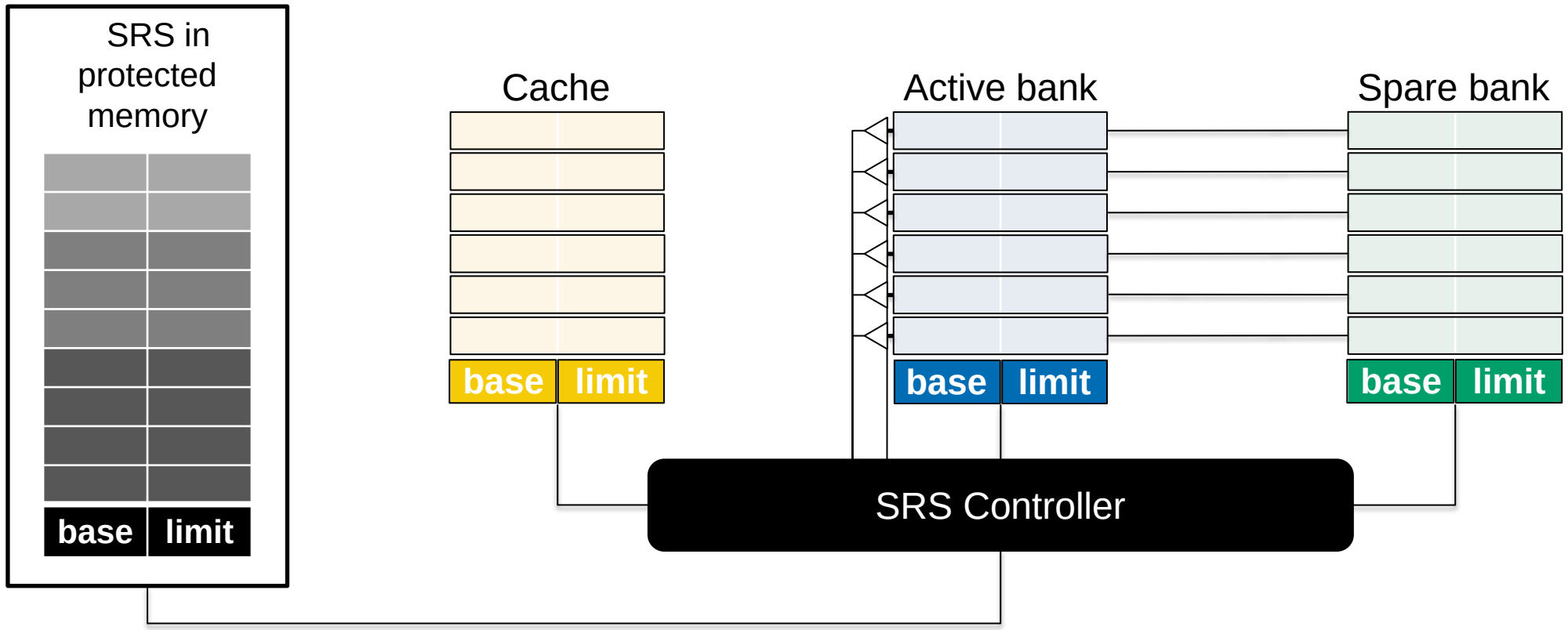


Storage Region Stack Hardware Design

Implementation

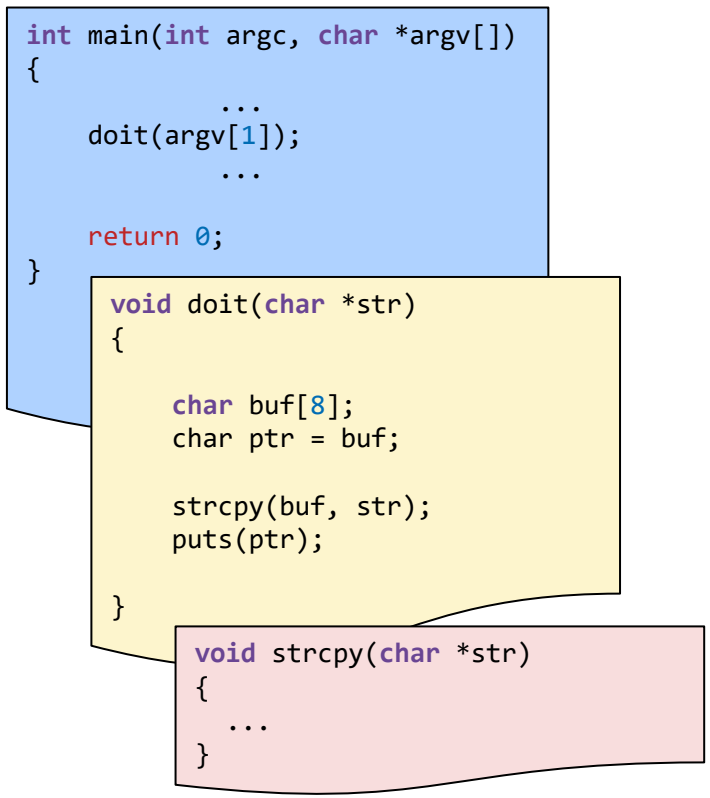
Active and delegated storage region rules stored in register banks

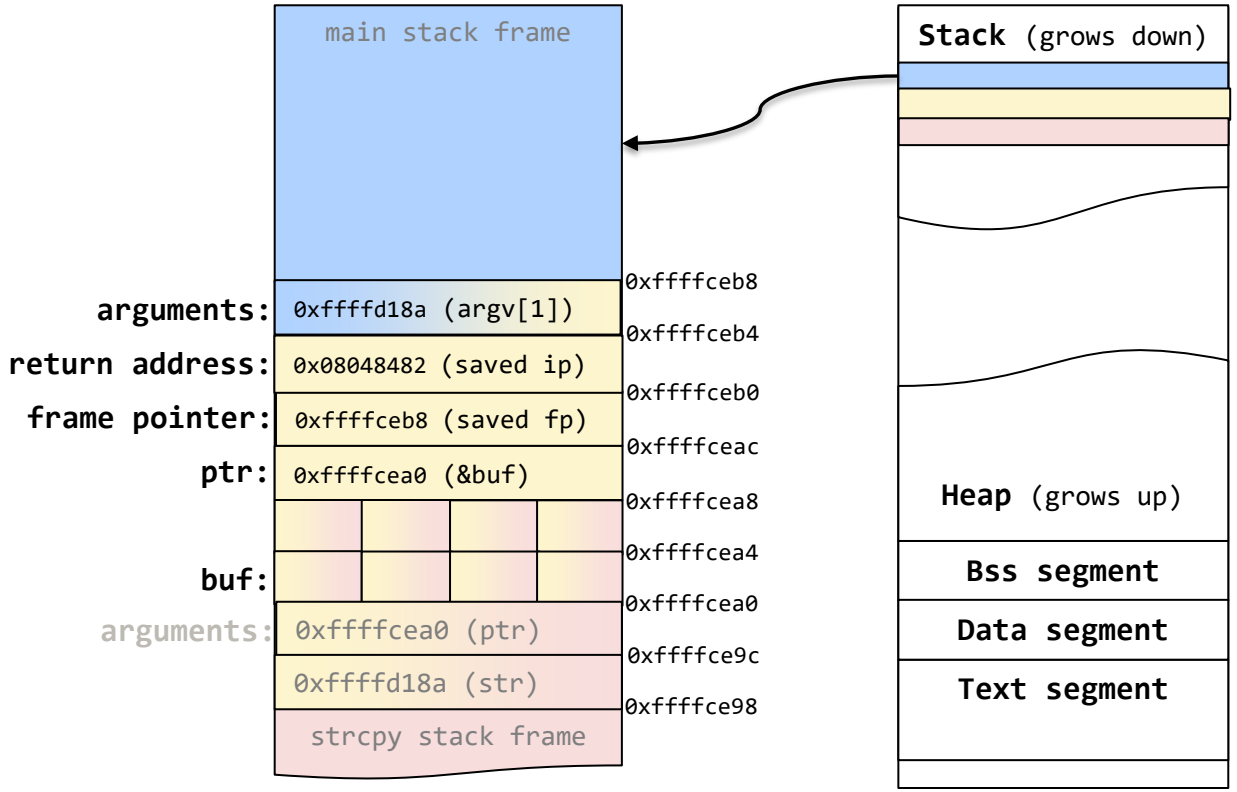
- Allows enforcement without slowing down loads / stores as active rules cached for fast access
- Cache management amortized over several instructions on execution context change



Function-granularity compartmentalization

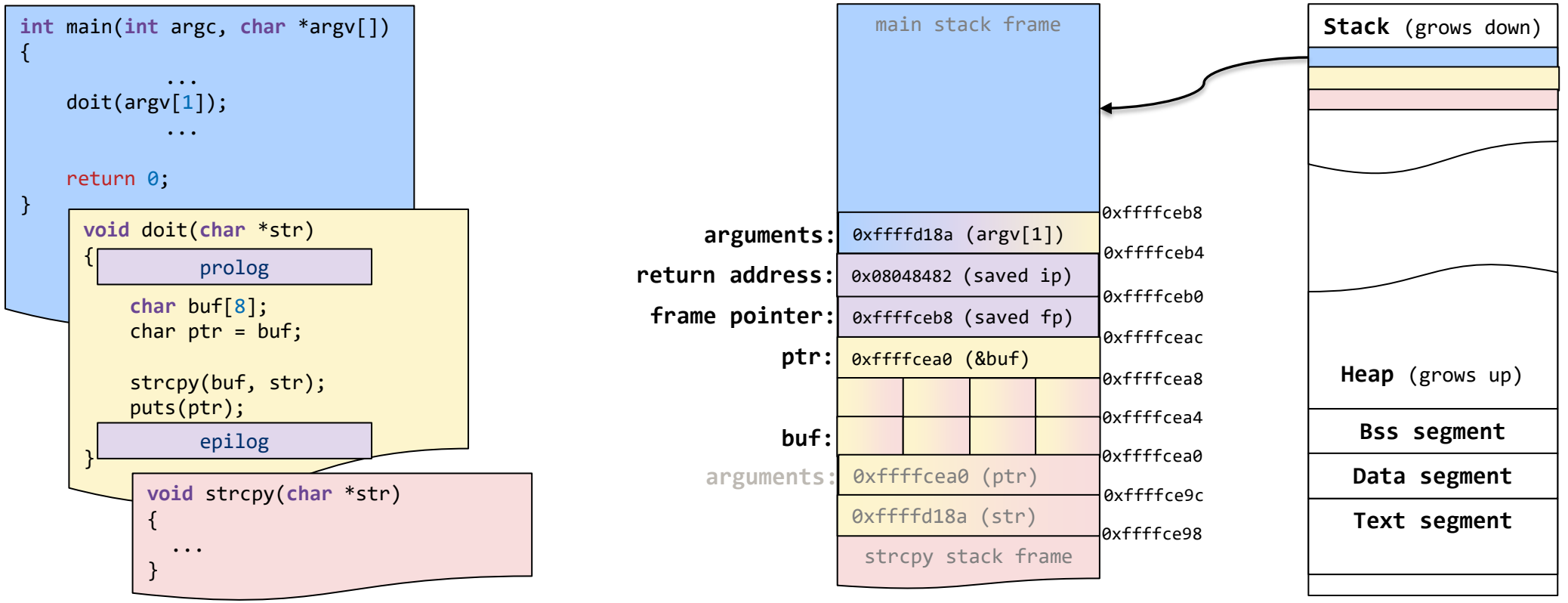
Functions separated into distinct execution contexts





Return-state compartmentalization

Function prologue and epilogue separated into own execution context

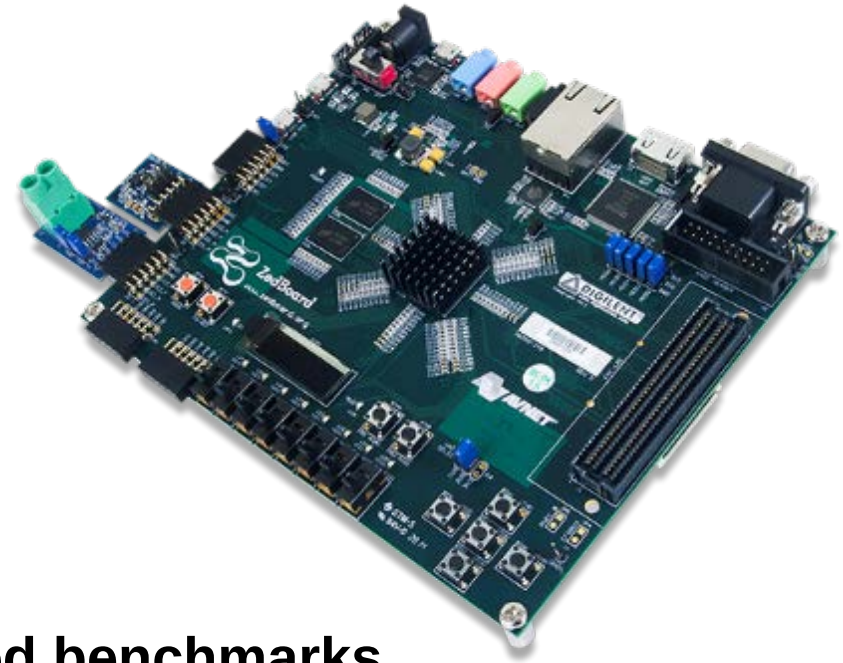


Implementation

Proof-of-Concept Implementation

Proof-of-Concept ISA extension for RISC-V processor

- Software simulation implemented for Spike ISA simulator
- Integrated with PULPino SoC on ZedBoard FPGA



GCC Plug-in for automatic HardScope instrumentation

- Function granularity enforcement for local, global, and static variables, function arguments and return values

Only 3.2% performance overhead in CoreMark embedded benchmarks

- 11% binary size increase due to instrumentation
- 32 entries in register banks (CoreMark used only up to 23)
- 574 byte memory overhead*

*) Maximum SRS depth: 71 entries over 11 frames encoded using 64 bits per SR entry + 4 bits per frame for the number of entries

Nyman et al. HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement, **DAC (2019)**

HardScope benefits

+ Adjustable granularity of enforcement
  e.g. module-, function-, code-block- compartmentalization
+ Can provide resilience against multiple classes of attacks
  e.g. ROP, DOP

Nyman et al. HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement, **DAC (2019)**

HardScope limitations

- Currently only supports single-threaded C programs
  Additions to hardware design needed to support concurrency
- Currently manual annotations needed to instrument dynamic data structures
  Coarse-granularity enforcement can be provided via wrappers
- Assumes programs minimize variable scope and module interdependence
  Programs without logical structure benefit less and consume more SRS resources
- Secure memory size (needed for storing info about storage regions) fixed at synthesis time
  Optimal size may be difficult to determine

- Hardware changes seen as too invasive

Nyman et al. HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement, **DAC (2019)**

Paper & source code

HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement

DAC 2019 (phew!)

Research report version available at https://arxiv.org/abs/1705.10295

Toolchain, emulator and code samples:

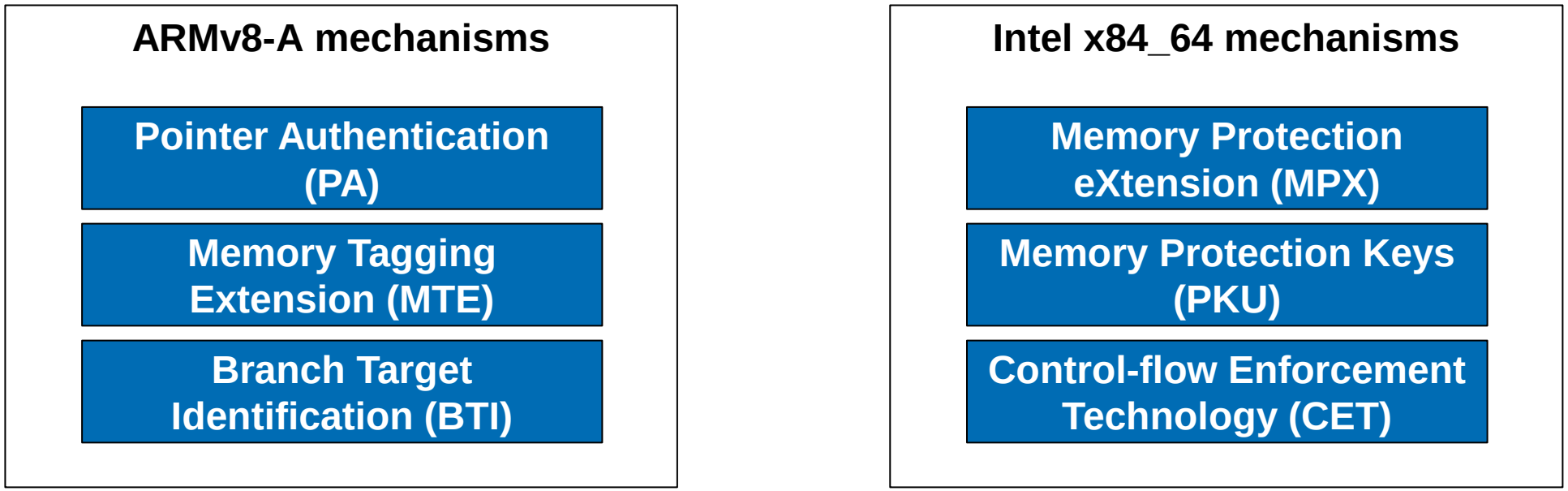
https://github.com/runtime-scope-enforcement/

https://arxiv.org/abs/1705.10295

https://github.com/runtime-scope-enforcement

Protect against run-time attacks without incurring a significant performance penalty or major deployment hurdles

Hardware assisted defenses in CotS processors



PARTS and PACStack

Run-time safety using ARM Pointer Authentication
Joint work with Huawei Helsinki Platform Security Team

Deploying new hardware extensions is difficult

But CPU vendors are rolling out several different hardware security mechanisms

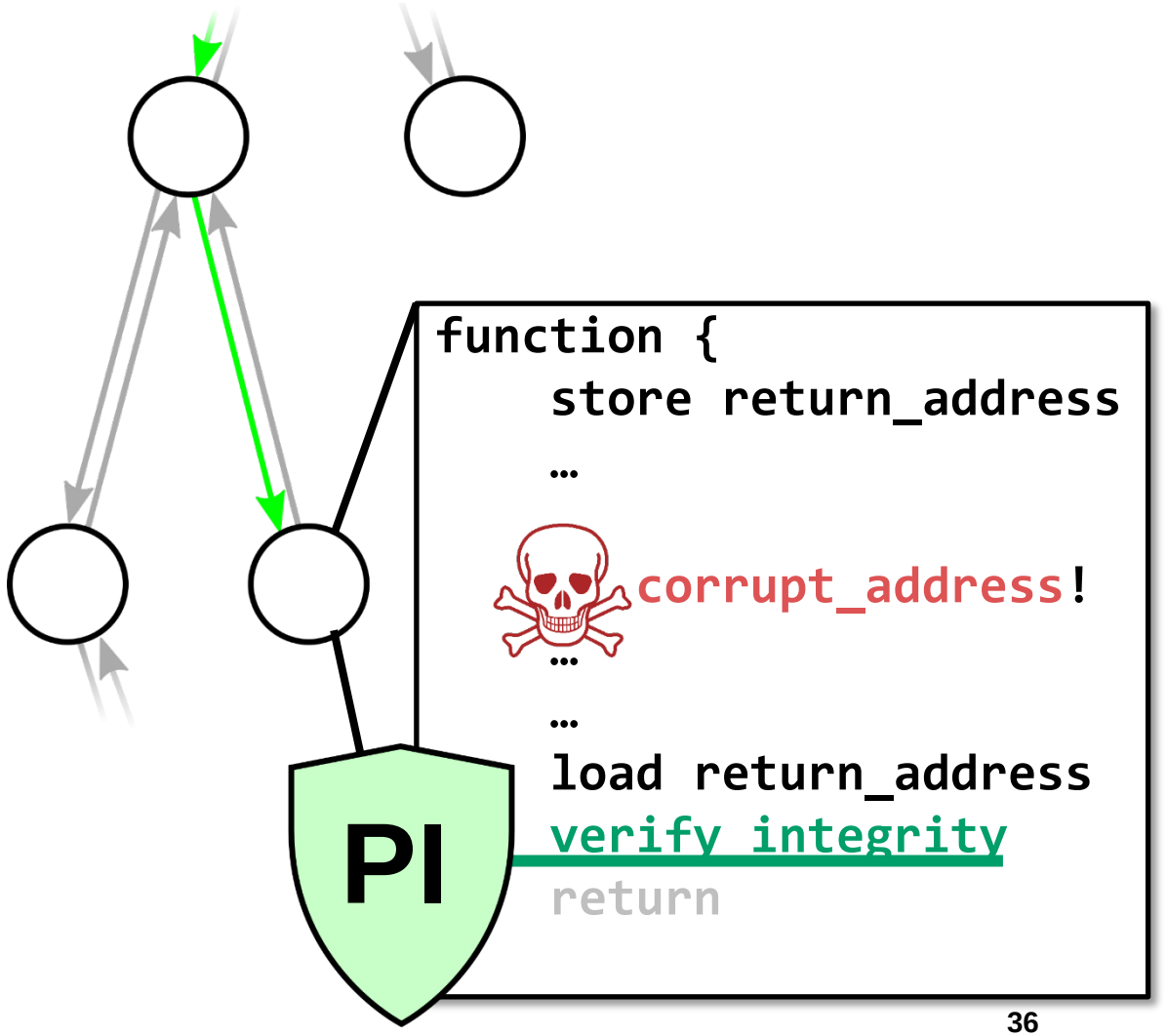
How can we use these mechanisms for run-time protection?

ARMv8-A mechanisms

Pointer Integrity: memory safety for pointers

Ensure pointers in memory remain unchanged

- **Code pointer integrity implies CFI**
  - Control-flow attacks manipulate code pointers
- **Data pointer integrity**
  - Reduces data-only attack surface



Kuznetsov et al. "Code-Pointer Integrity", USENIX OSDI 2014

ARMv8.3-A Pointer Authentication

General purpose hardware primitive approximating pointer integrity

- Ensure pointers in memory remain unchanged

Introduced in ARMv8.3-A specification (2016) **to be improved in ARM-8.6-A** (2020)

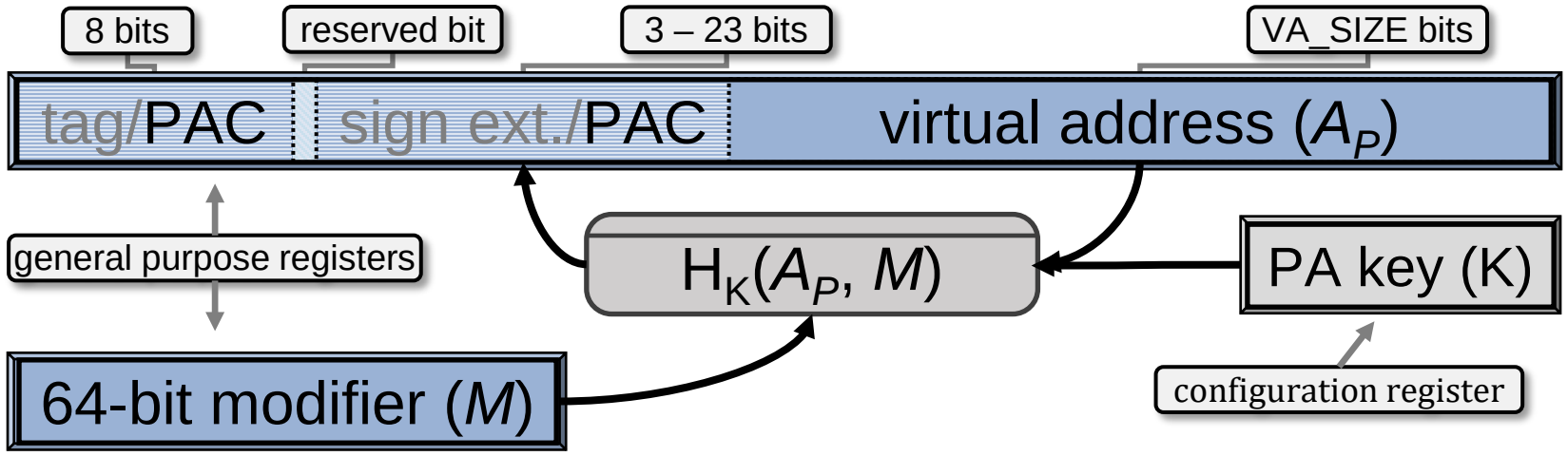
- First compatible processors 2018 (Apple A12 / iOS12)
- Support in Linux 5.0
- Instrumentation support in GCC 7.0 (-msign-return address, deprecated in GCC 9.0 -mbranch-protection=pac-ret[+leaf] GCC 9.0 and newer)

ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019
ARM. Developments in the Arm A-Profile Architecture: Armv8.6-A. September 2019

ARMv8.3-A PA – PAC Generation

Adds Pointer Authentication Code (PAC) into unused bits of pointer

- Keyed, tweakable MAC from pointer address and 64-bit modifier
- PA keys protected by hardware, modifier decided where pointer created and used



ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019

PA-based protection schemes

PA instructions are primitives, assembled to form protection schemes

Two main components:

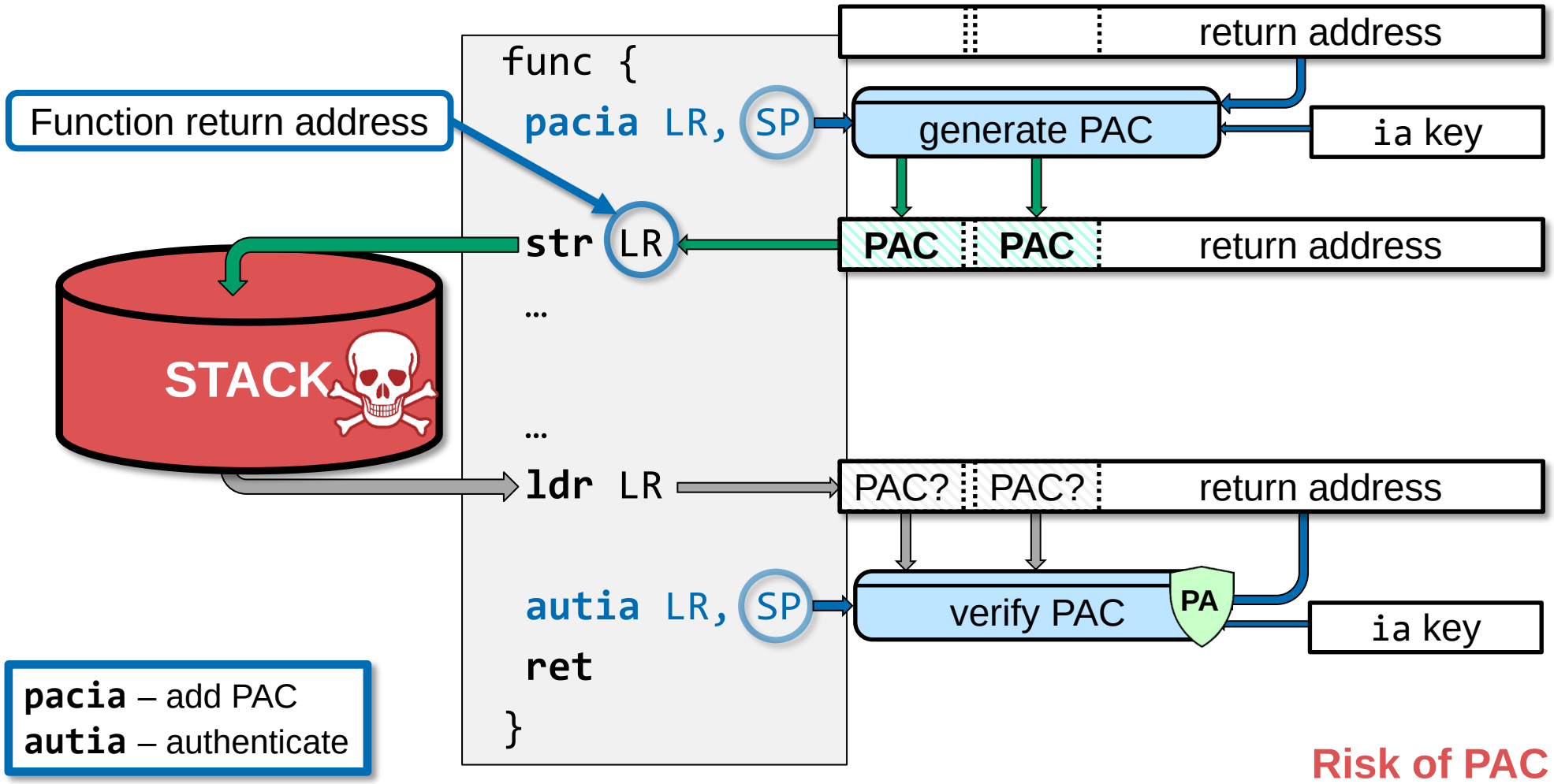
- When are pointers “PACed” and “unPACed”?
- Which modifier is used at a given point?

What should the modifier be for a given pointer?

- For security: using many different modifiers makes replay attacks harder
- For functionality: large numbers of modifiers are hard to keep track of

Example: `-msign-return-address`

Deployed in GCC 5.0 and LLVM/Clang 7.0



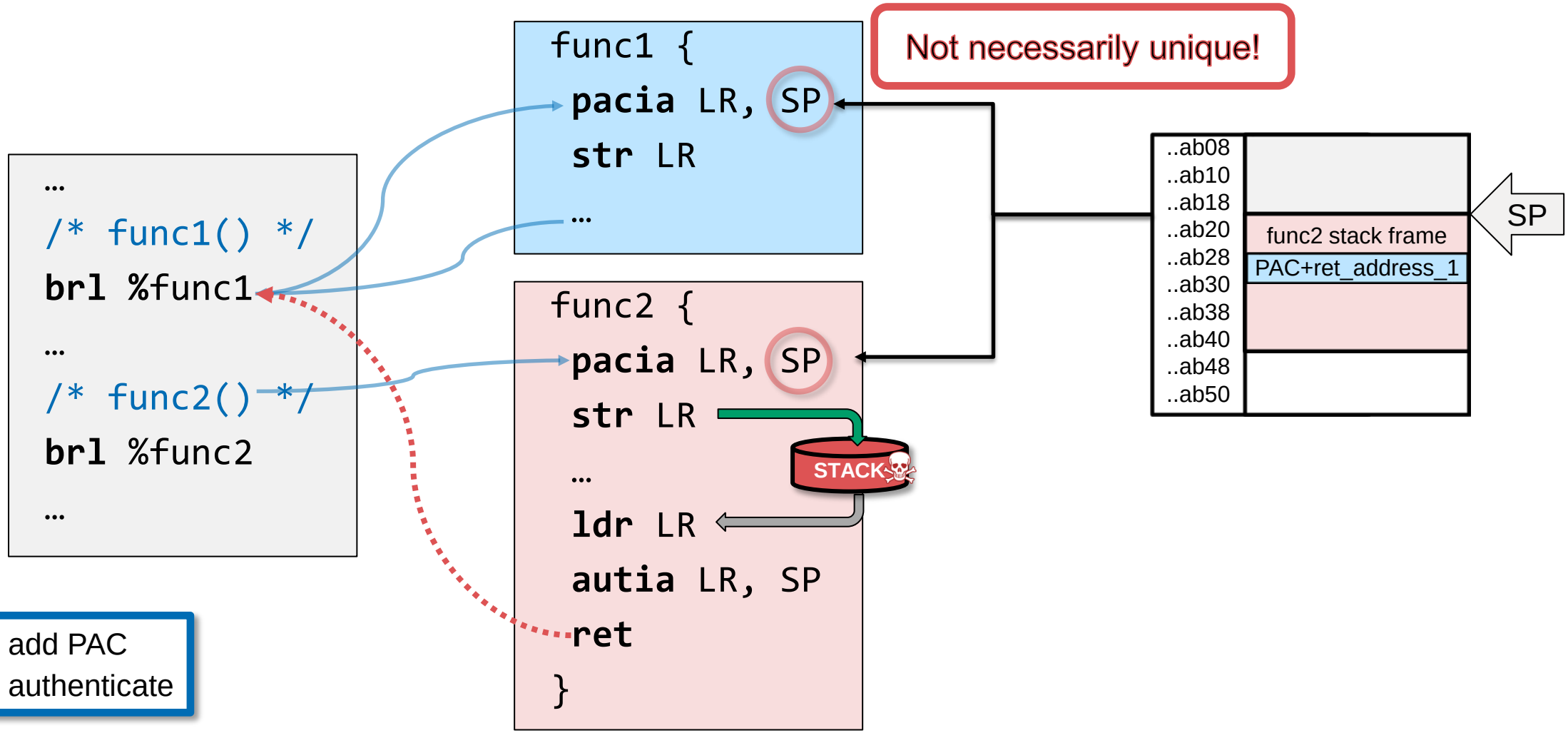
`pacia` – add PAC
`autia` – authenticate

Risk of PAC reuse!

Qualcomm "Pointer Authentication on ARMv8.3", whitepaper 2017

PA only approximates fully-precise pointer integrity

Adversary may reuse PACs



pacia – add PAC
autia – authenticate

PA-assisted Run-time Safety (PARTS)

Expands scope of PA protection

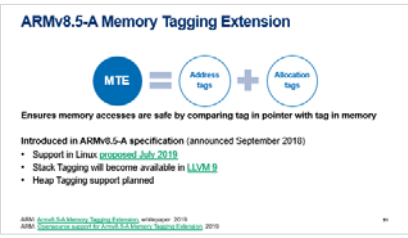
- Return address signing
- Code pointer signing
- Data pointer signing

Mitigates pointer reuse by binding

- return addresses to the function definition
- code and data pointers to the pointer type

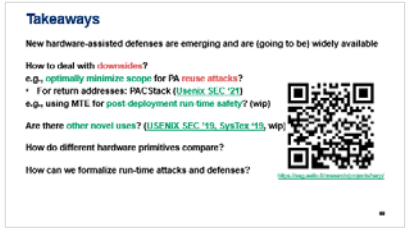
Liljestrand et al. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication USENIX Security (2019)

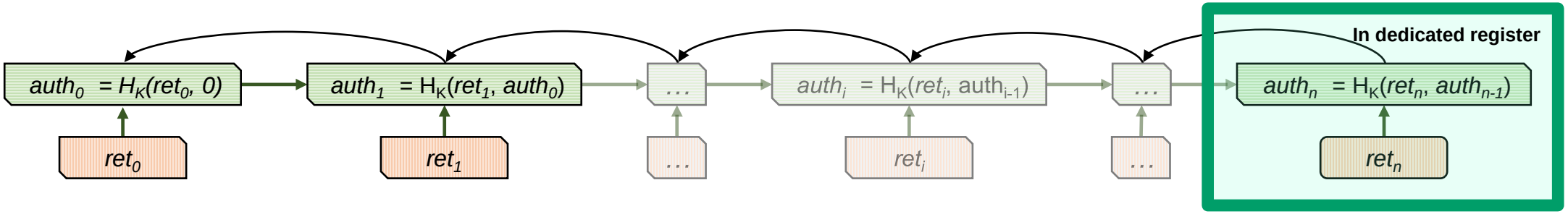
Authenticated Call Stack: high-level idea



Chained MAC of authentications tokens cryptographically bound to return addresses

- Provides modifier (*auth*) bound to all previous return addresses on the call stack
- Statistically unique to control-flow path
  - prevents reuse
  - allows precise verification of returns





$auth_i$, $i \in [0, n-1]$ bound to corresponding return addresses, ret_i , $i \in [0, n]$, and $auth_n$

Liljestrand et al. PACStack: an Authenticated Call Stack. Usenix Security (2021).

Mitigation of hash-collisions: PAC masking

- **Challenge: PAC collisions occur on average after $1.253*2^{b/2}$ return addresses**
  - For b=16 n= 321 addresses

- **Solution: Prevent *recognizing* collisions by masking each *auth***
  - pseudo-random mask generated using `pacib`(`0x0`, $auth_{i-1}$)

Attack	w/o Masking	w/ Masking
Reuse previous auth collision	1	2^{-b}
Guess auth to existing call-site	2^{-b}	2^{-b}
Guess auth to arbitrary address	2^{-2b}	2^{-2b}

Maximum probability of success for different attacks

Liljestrand et al. PACStack: an Authenticated Call Stack. Usenix Security (2021).

ARMv8.5-A Memory Tagging Extension

Ensures memory accesses are safe by comparing tag in pointer with tag in memory

Introduced in ARMv8.5-A specification (announced September 2018)

- Support in Linux proposed July 2019
- Stack Tagging will become available in LLVM 9
- Heap Tagging support planned

ARM. Armv8.5-A Memory Tagging Extension, whitepaper 2019
ARM. Opensource support for Armv8.5-A Memory Tagging Extension. 2019

ARMv8.5-A MTE

Address tags stored in top 4-bits of a pointer

- uses existing top-byte ignore (TBI) feature

Allocation tags stored transparently by hardware

- 4-bit tag per 16-byte granule of memory

Mismatch between tags reported either:

- synchronously (precise check during testing, leads to an MTE fault), or
- asynchronously (imprecise checks, kept track of in a status register, checked by kernel)

ARM, Armv8.5-A Memory Tagging Extension, whitepaper 2019
ARM, Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile, Version F.c. July 2020

LLVM MemTagSanitizer

Random base tag for each stack frame

- Slots sequentially tagged to minimize tag book-keeping
- Uses Stack Safety Analysis to optimize instrumentation

Globals tagging requires loader support to assign initial tags

Heap tagging via the new secure Scudo allocator

Provides:

- Deterministic prevention of sequential overflows
- Probabilistic detection of use-after-free and non-sequential out-of-bounds
  - In the general case: $1-2^{-4} \approx 0.94$ chance of detection

E. Stepanov *et al.*, Memory tagging in LLVM and Android, LLVM Developers' Meeting (2020)
LLVM, MemTagSanitizer, online documentation
LLVM, Stack Safety Analysis, online documentation

MTE (and MemTagSanitizer) challenges

Tags are corruptible

- Random tags prevent hard-coding
- Adversary can inject tagged pointers
  - Wild-card tags can be used to circumvent MTE checking!
  - Guessing probability 2^{-4} with short 4-bit tags

Used with generic LLVM optimization (to avoid unnecessary tagging)

but analysis not MTE-aware: assumes pointers in memory are unsafe

Cannot guarantee memory-safety in the standard adversary model: adversary who

- has read/write access to entire process memory
- can exploit some memory vulnerability read from or write to any address
- can repeat attacks

(How) Can we use MTE to provide post-deployment run-time memory safety?

LLVM, MemTagSanitizer, online documentation

ARMv8.5-A Branch Target Identification

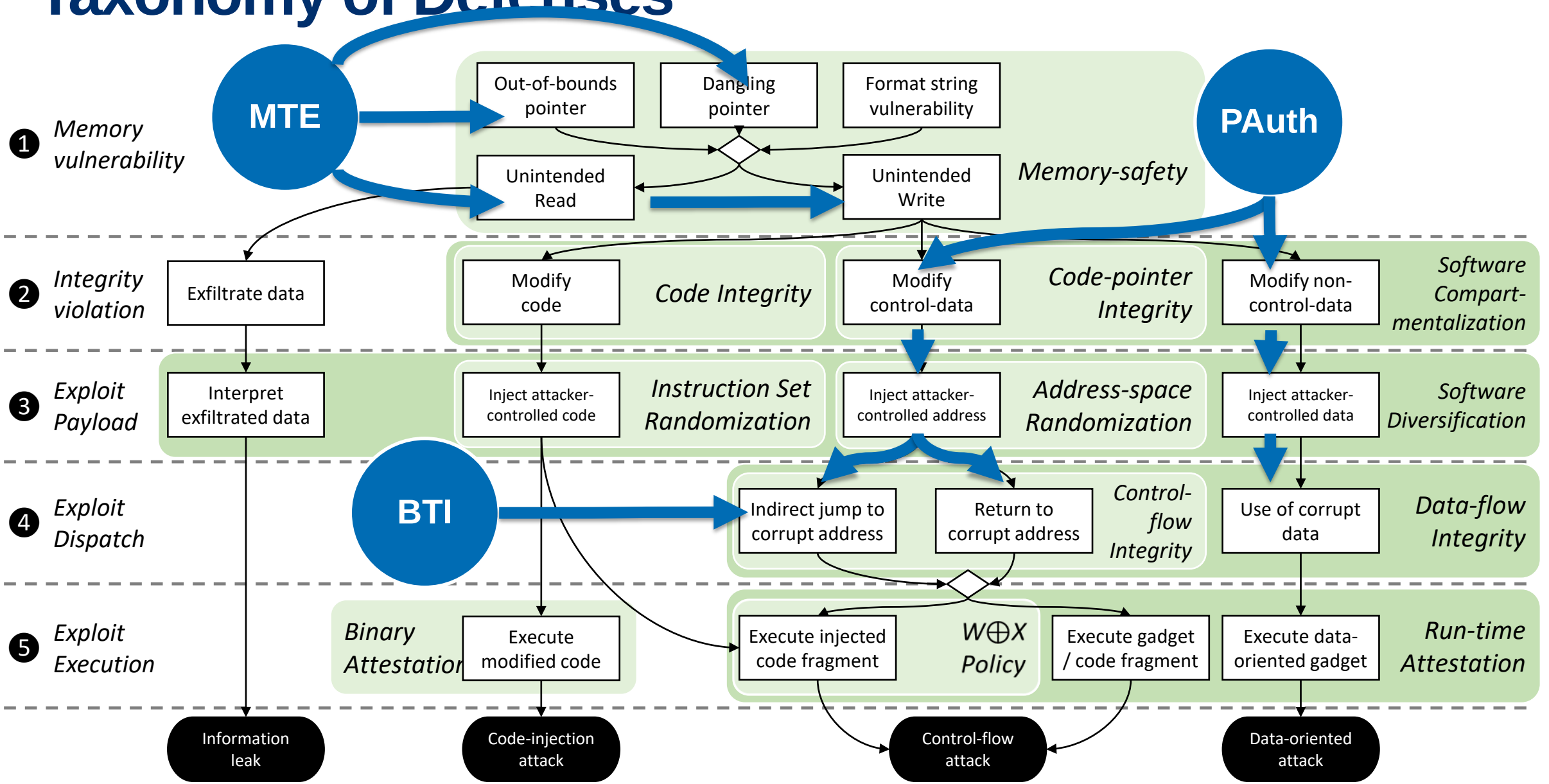
Hardware-assisted CFI similar to Intel CET Indirect Branch Tracking

Introduced in ARMv8.3-A specification (2016)

- Support for Linux proposed May 2019
- Instrumentation support in GCC 9.0 (-mbranch-protection=standard|bti)

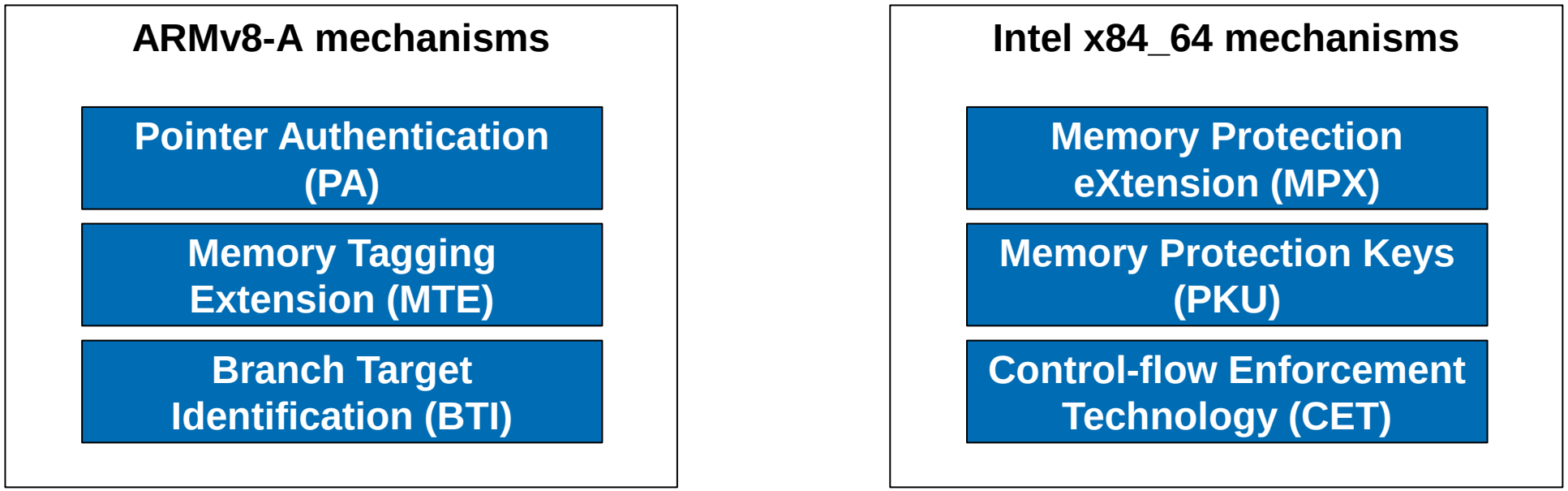
ARM. Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile. Version E.a. July 2019

Taxonomy of Defenses



Adapted from Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013)

Hardware assisted defenses in CotS processors



Intel CET ShadowStack vs. ARMv8.3-A PA

	Intel CET Shadow Stack	ARMv8.3-A PA
Return address protection	✓	✓
Indirect branch protection	✗	capable*
Data pointer protection	✗	capable*
Enforcement model	Deterministic	Probabilistic**
Immune to pointer reuse	✓	✗
Memory Overhead	Low to Moderate	N/A
Run-time Overhead	? (likely low)	Low

*) Liljestrand et al. PAC It Up: Towards Pointer Integrity using ARM Pointer Authentication. USENIX Security (2019)
**) Liljestrand et al. PACStack: an Authenticated Call Stack. Usenix Security (2021)

Custom solution vs. generic building block

PA can be used for

• Return address protection	*original Qualcomm design*, ***PACStack** (2)*
• **Indirect branch protection**	***PARTS (1)***
• **Data pointer protection**	***PARTS (1)***
• **Strengthening Stack Canaries**	***PCan (2)***
• **Authenticated data structures**	***work-in-progress (4)***
• **…**	

1) Liljestrand et al., PAC It Up: Towards Pointer Integrity using ARM Pointer Authentication. USENIX Security (2019)
2) Liljestrand et al., PACStack: an Authenticated Call Stack. Usenix Security (2021)
3) Liljestrand et al., Protecting the stack with PACed Canadies, ACM SysTex (2019)
4) Ghorshi et al., *work-in-progress* (2022)

Takeaways

New hardware-assisted defenses are emerging and are (going to be) widely available

How to deal with downsides?

e.g., optimally minimize scope for PA reuse attacks?

- **For return addresses: PACStack (Usenix SEC ‘21)**

e.g., using MTE for post-deployment run-time safety? (wip)

Are there other novel uses? (USENIX SEC ’19, SysTex ‘19, wip)

How do different hardware primitives compare?

How can we formalize run-time attacks and defenses?

https://ssg.aalto.fi/research/projects/harp/

Acknowledgments

Icons on slides 3, 4, 5, 15, 16, 17, 19, 21, 22, 23 and 24 made by Good Ware from www.flaticon.com licensed by CC 3.0 BY

The PHP logo on slide 25 made by Colin Viebrock licensed by CC BY-SA 4.0

The BSD daemon on slide 25 is copyright of Marshall Kirk McKusick

All product and company names and logos are trademarks™ or registered ® trademarks of their respective holders. Use of them does not imply any affiliation with or endorsement by them.

Slide 6 (Return-oriented programming (high-level idea) is by Luca Davi.