Aalto University

Hardware-assisted run-time Protection

On balancing security and deployability

N. Asokan
http://asokan.org/asokan/
@nasokan

How to thwart run-time attacks?

Run-time attacks are now routine

Software defenses incur security vs. cost tradeoffs

Hardware-assisted defenses are attractive

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: Run-time safety using ARM Pointer Authentication**

Aalto University

TECHNISCHE UNIVERSITÄT DARMSTADT

HardScope

Hardware-assisted Run-time Scope Enforcement

Thomas Nyman[†], Ghada Dessouky[‡], Shaza Zeitouni[‡], Aaro Lehikoinen[†], Andrew Paverd[†], N. Asokan[†], Ahmad-Reza Sadeghi[‡]

[†)] Aalto University, [‡)] Technische Universität 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

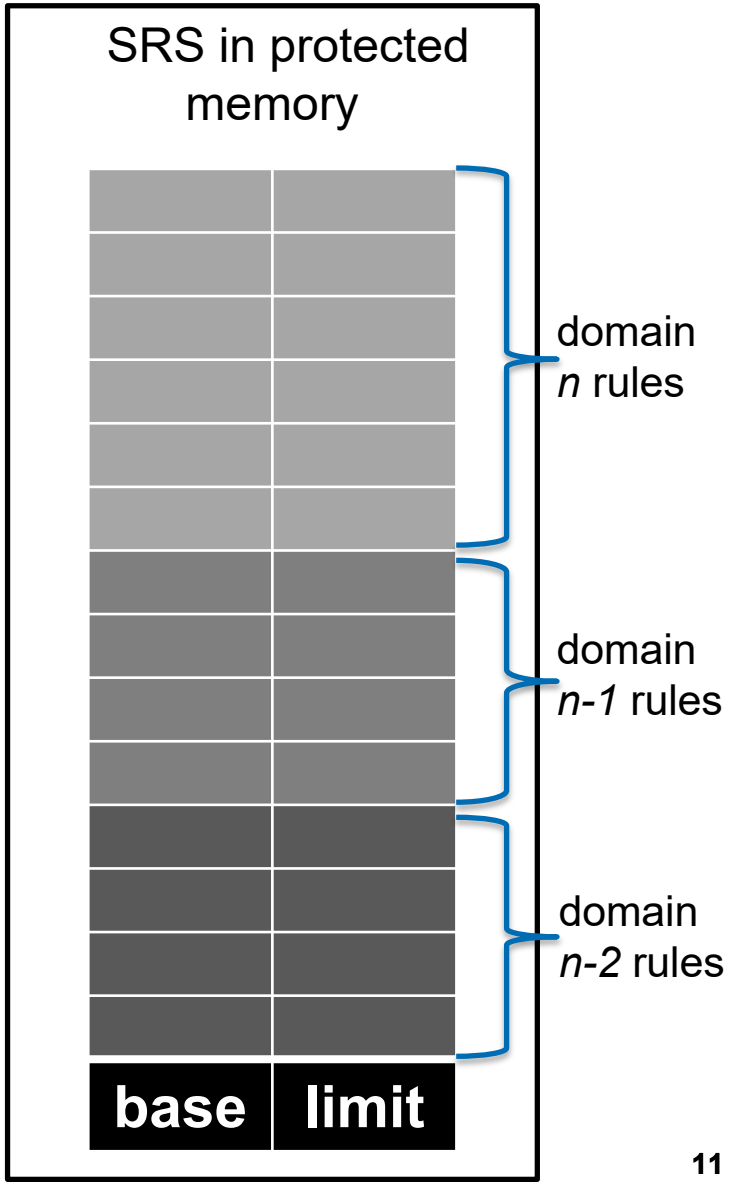
- Stores the bounds of each used *storage region* (e.g. stack variable, heap object, global variable)
- Frames created upon domain entry (`sbenter` → `push`)
- Frames discarded upon domain exit (`sbxit` → `pop`)

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

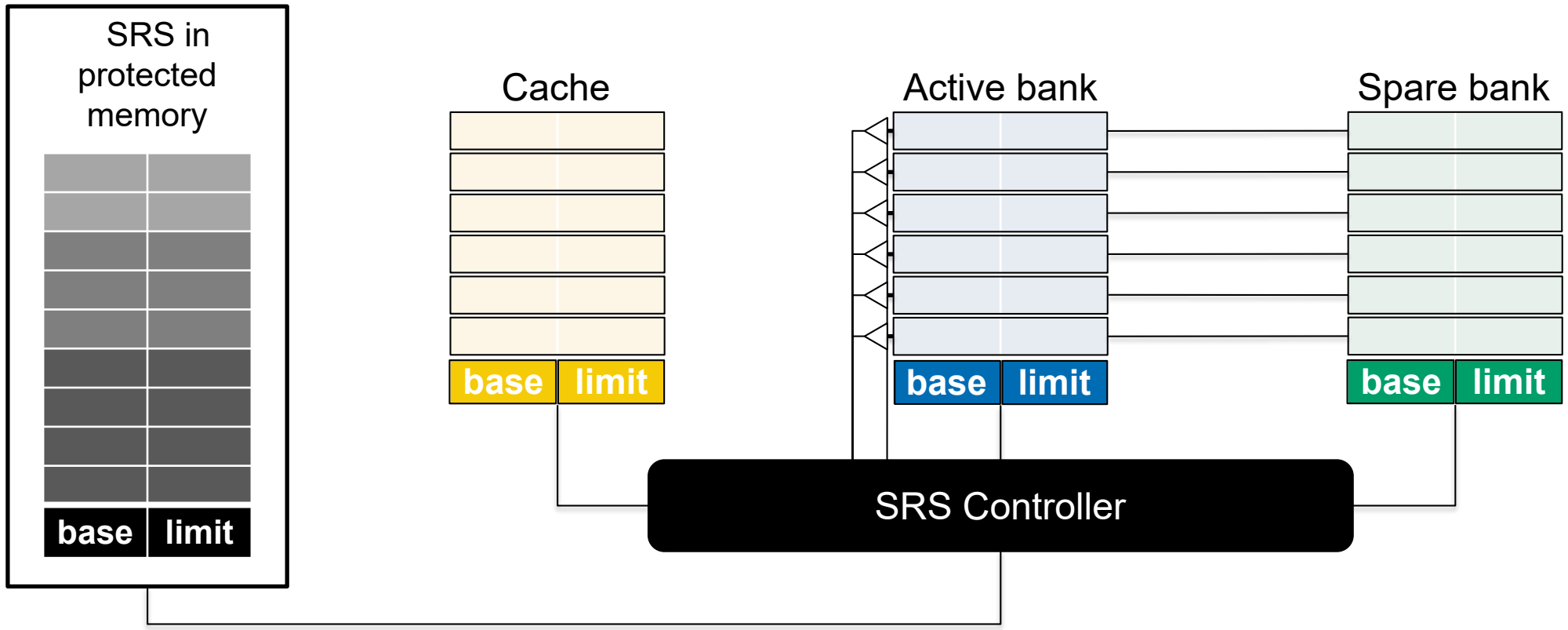
- Rules only modifiable by HardScope instructions



Storage Region Stack Hardware Design

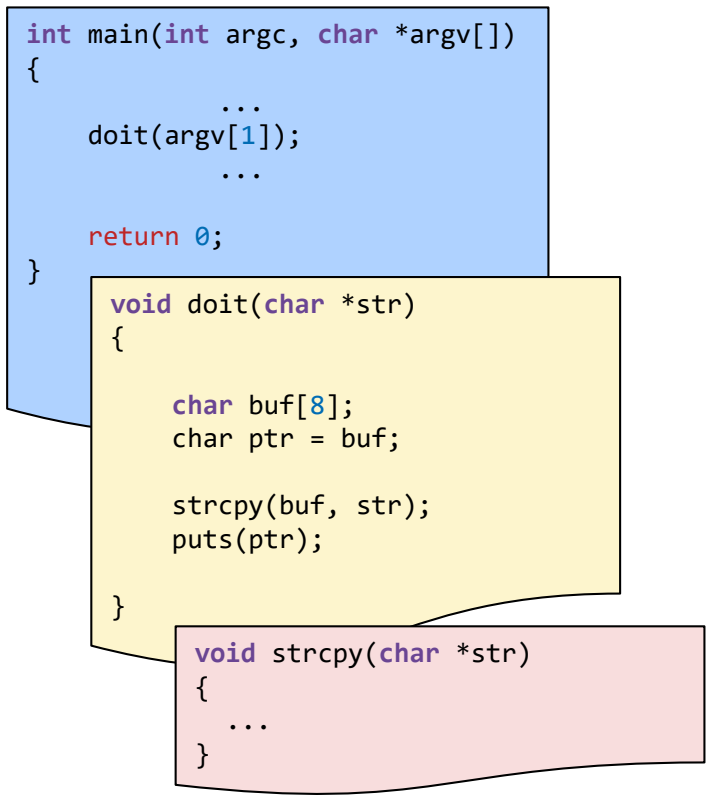
Active and delegated storage region rules stored in register banks

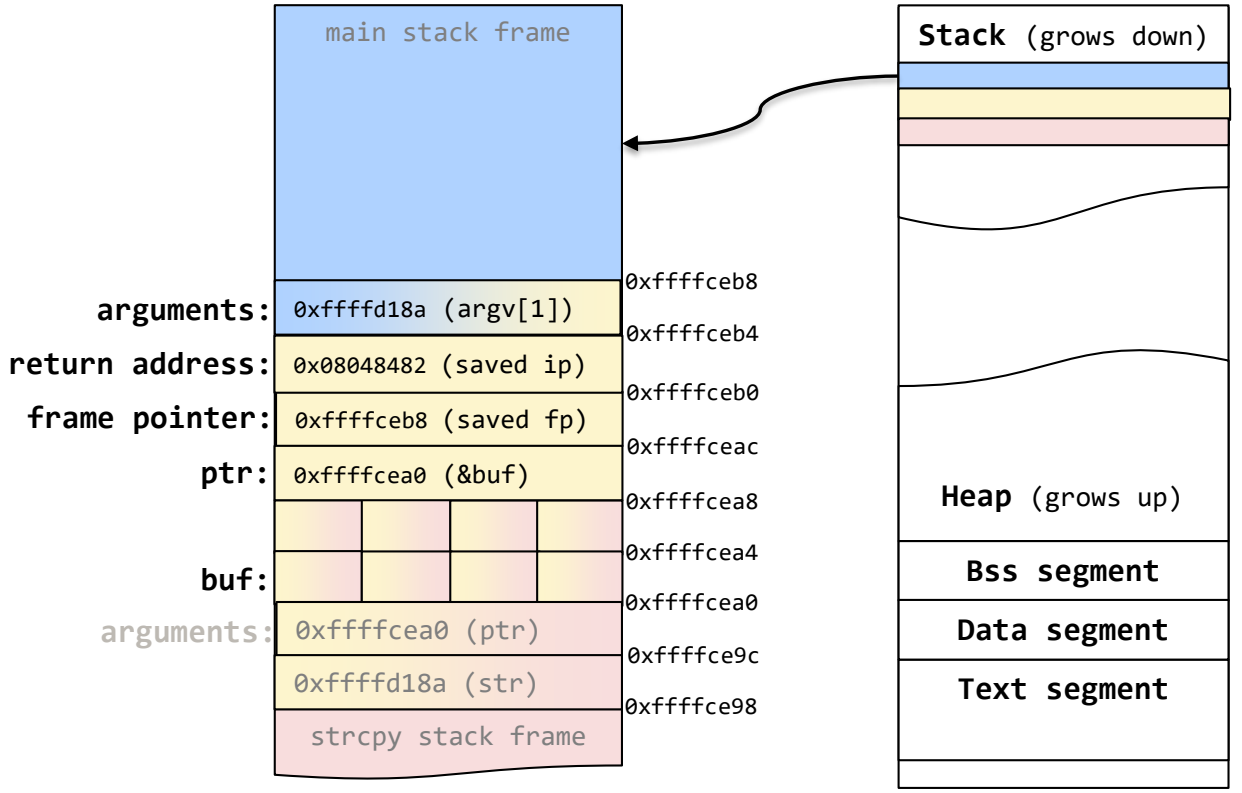
- Allows enforcement without slowing down loads / stores as active rules cached for fast access
- Cache management amortized over several instructions on execution context change



Function-granularity compartmentalization

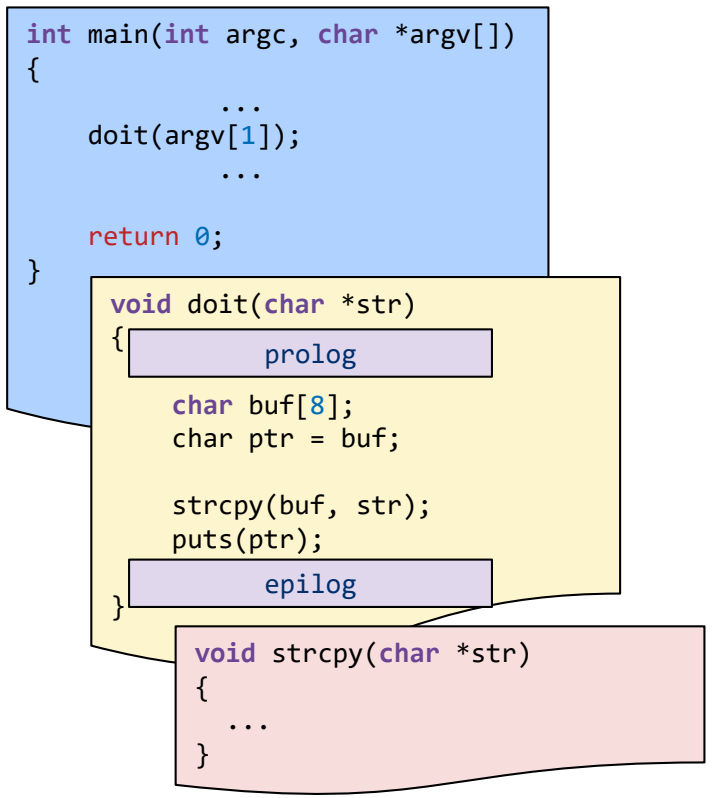
Functions separated into distinct execution contexts

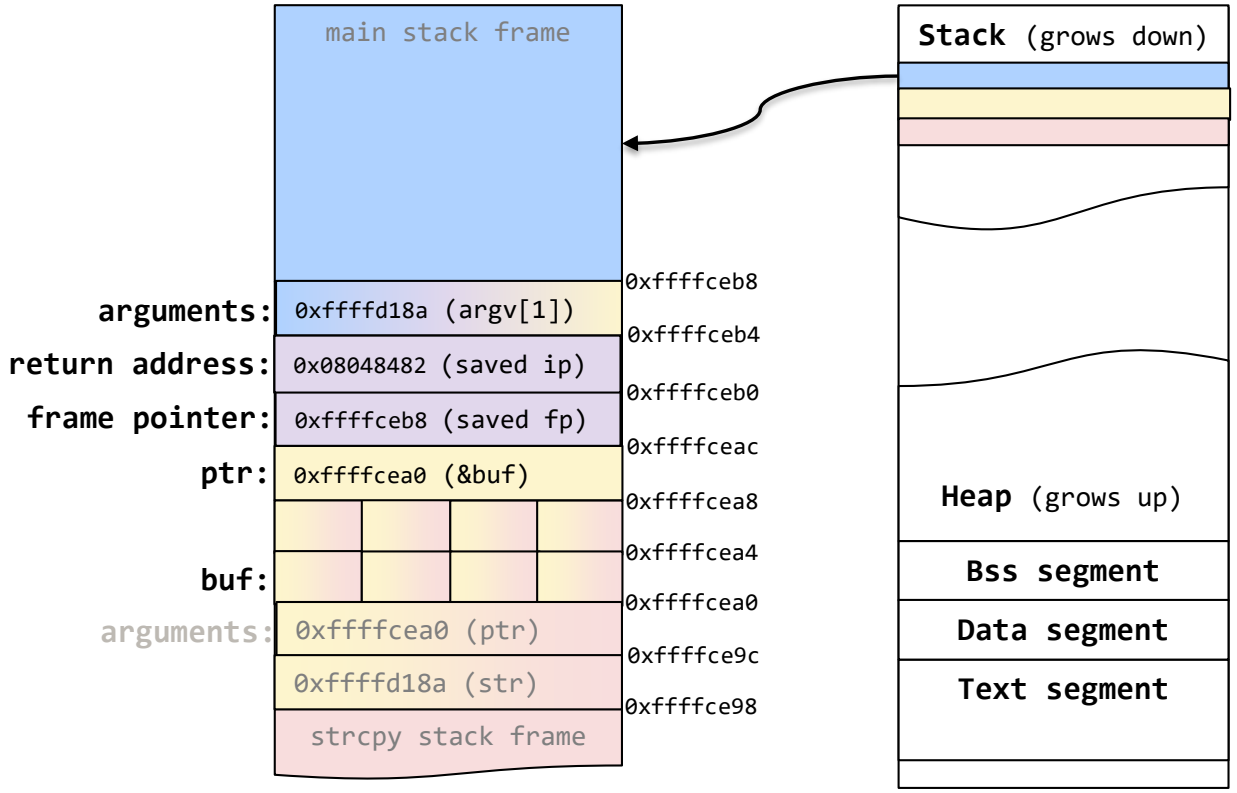




Return-state compartmentalization

Function prolog and epilog separated into own execution context



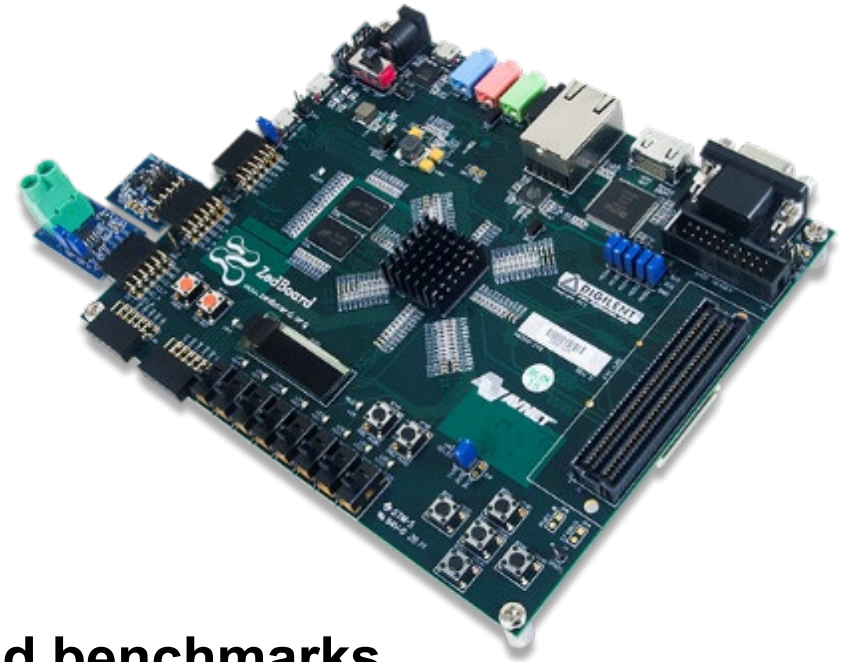


Implementation

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

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

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
- SRS frame size fixed at synthesis time
  Optimal frame size may be difficult to determine

Technical report & 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

Towards Pointer Integrity using ARM Pointer Authentication

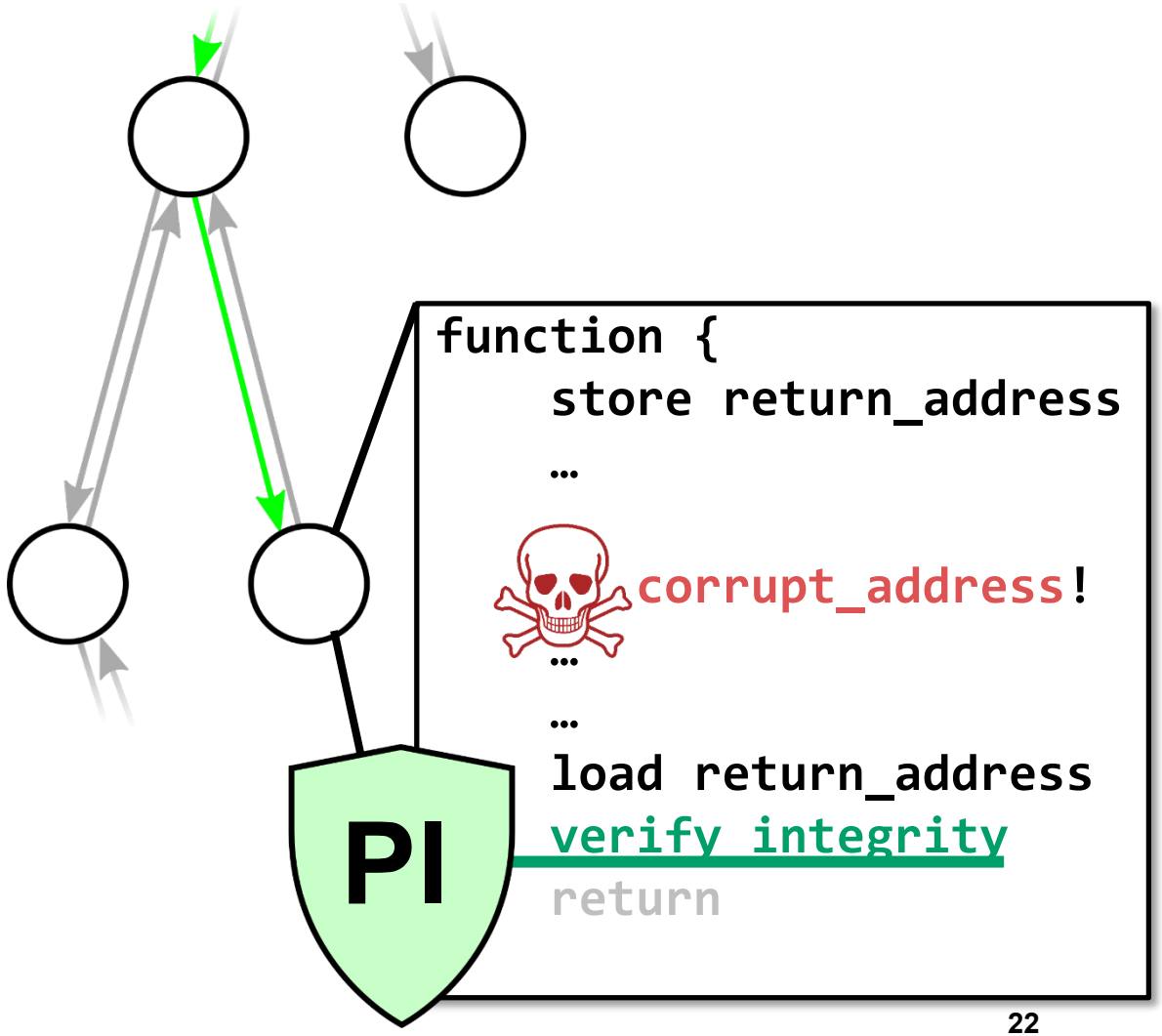
Hans Liljestrand[†], Thomas Nyman[†], Kui Wang[‡], Carlos Chinea Perez[‡], Jan-Erik Ekberg[‡], N. Asokan[†]

[†)] Aalto University, [‡)] Huawei Technologies

Pointer Integrity (PI): memory safety for pointers

Ensures that a pointer at use time is the same as at creation time

- **Code pointer integrity implies CFI**
  - CF attacks rely on pointer manipulation
- **Data pointer integrity**
  - reduces data-only attack surface
  - prevents all known Data-Oriented Programming (DOP) attacks



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

Can PI be realized in practice?

Can we use ARM v8.3-A Pointer Authentication (PA)?

But, PA is vulnerable to pointer reuse!

Our work: Design PA-assisted Run-time Safety (PARTS)

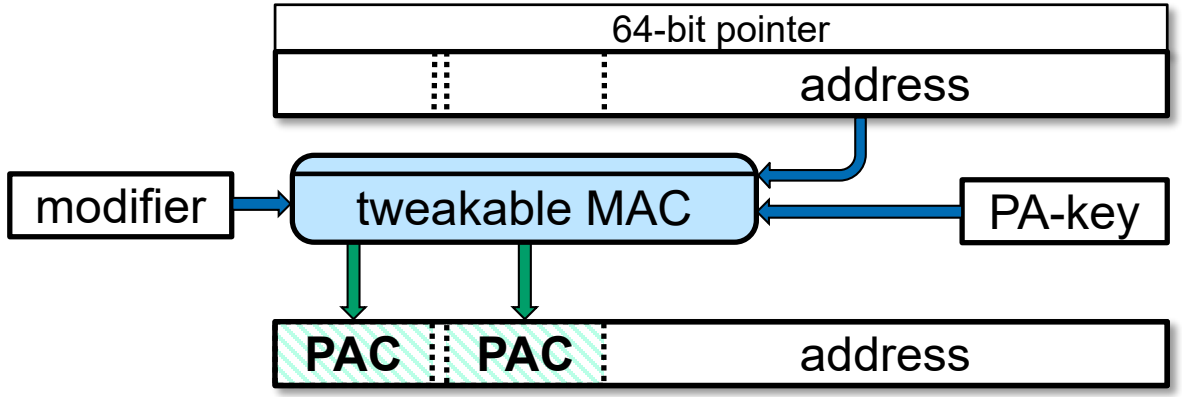
- Return address signing ≈ backward-edge CFI
- Code pointer signing ≈ forward-edge CFI
- Data pointer signing ≈ data-flow integrity for pointers
- Mitigates pointer reuse with run-time type safety

ARM "Armv8-A architecture: 2016 additions", 2016

ARM 8.3-A Pointer Authentication

Pointer Authentication Codes (PAC)

- Tweakable MAC
- Set in unused bits of virtual address



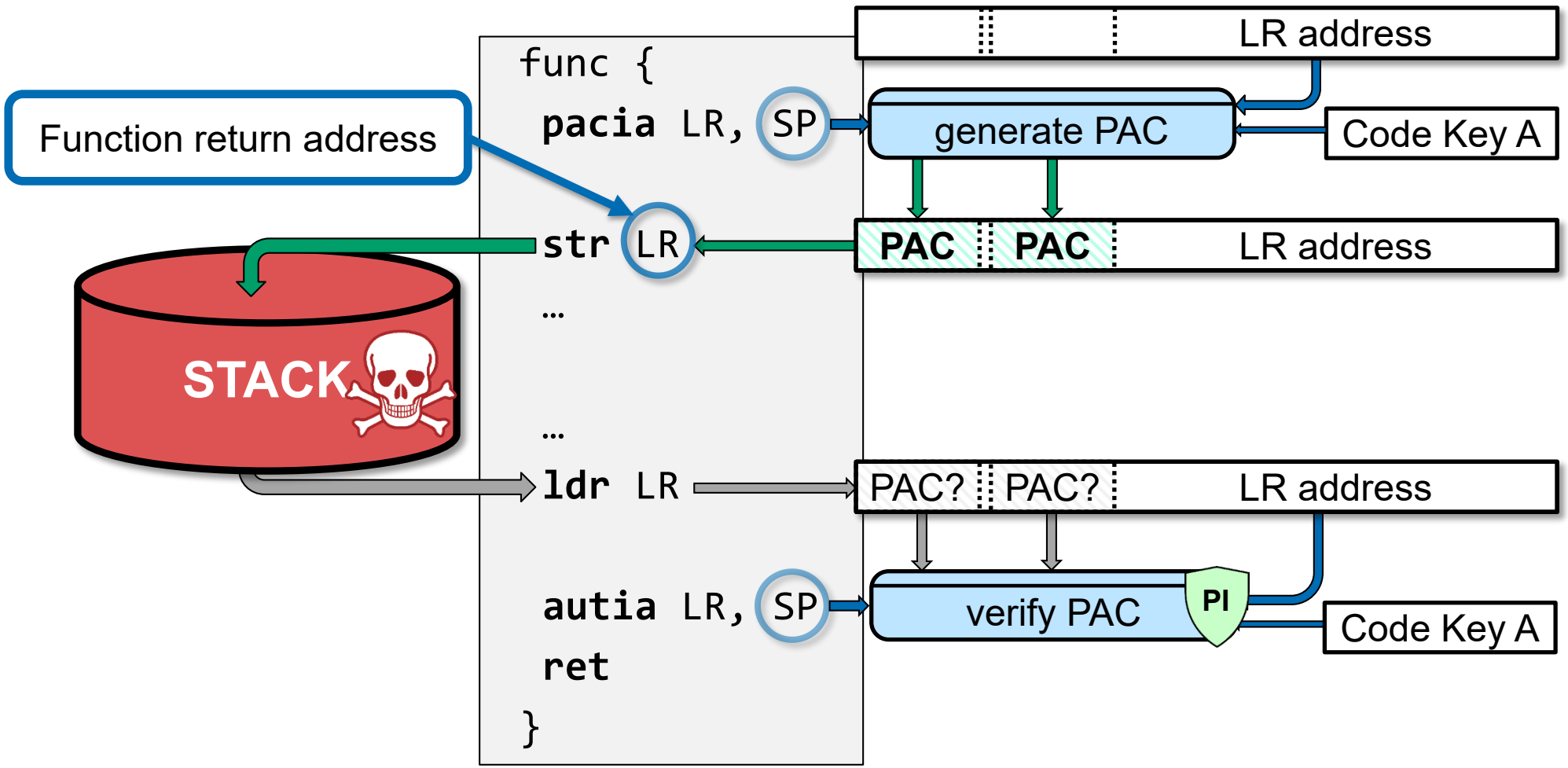
Key/configuration set at higher privilege level

Instrument with new PAC handling instructions

- Opcode determines used key
- Operands set PA modifier (tweak value)

instructions	Code-key		Data-key		Gen.-key
	A	B	A	B	
pacia	X				
pacib		X			
pacda			X		
pacdb				X	
pacga					X
autia	X				
autib		X			
autda			X		
autdb				X	

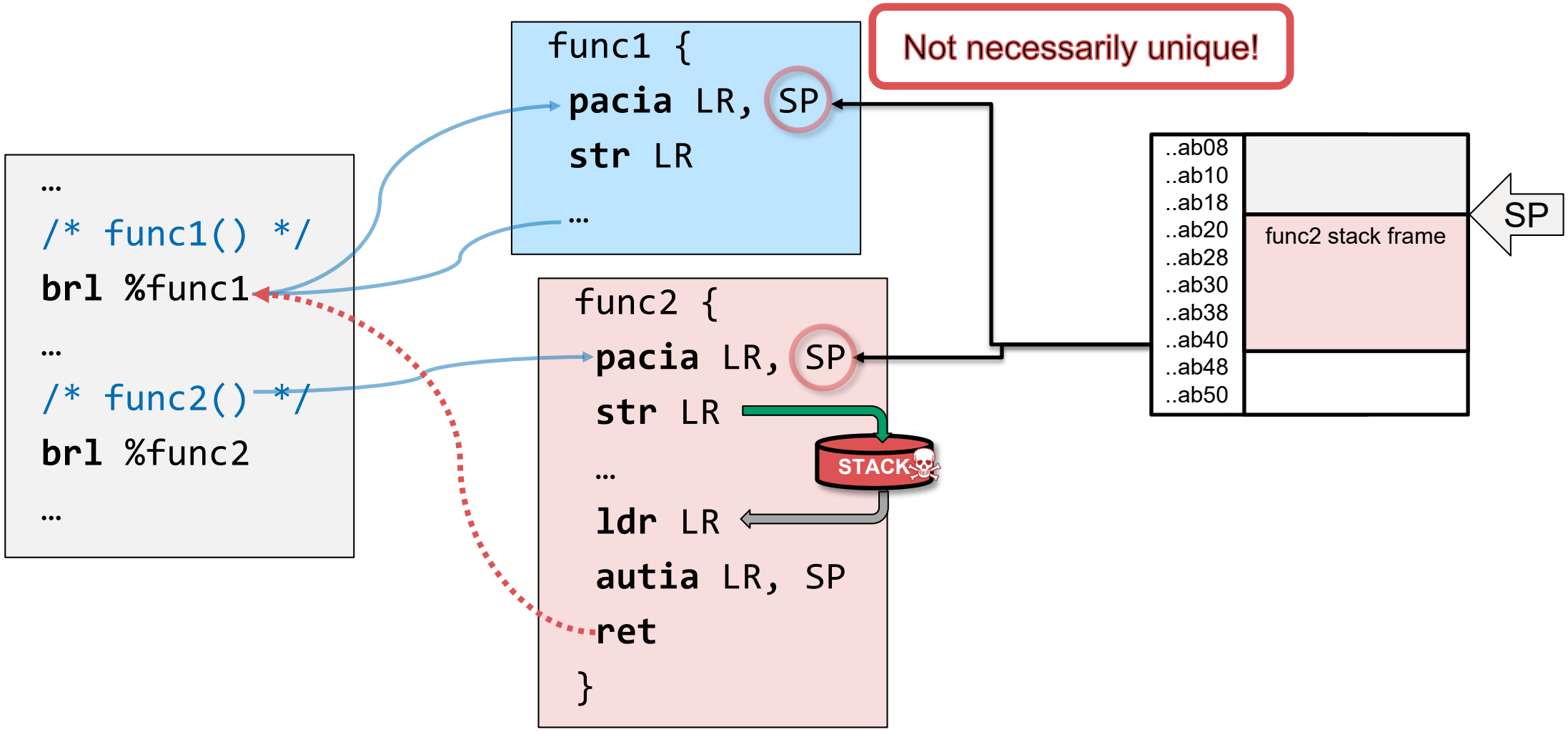
Example: PA-based return address signing



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

PA only approximates fully-precise pointer integrity

Adversary may re-use PACs



Design

Hardening return address signing

Modifier: function-id + SP value

- Function-id assigned at compile-time
- Prevent cross-function return address reuse

Future additions

- Combine with SP randomization

```
func {
 mov Xm, SP
 mov Xm, #f_id, #lsl_16
 pacia LR, Xm
 str LR
 …
 …
 ldr LR
 mov Xm, SP
 mov Xm, #f_id, #lsl_16
 autia LR, Xm
 ret
}
```


Mashtizadeh et al. “Cryptographically Enforced Control Flow Integrity”, ACM CCS 2015

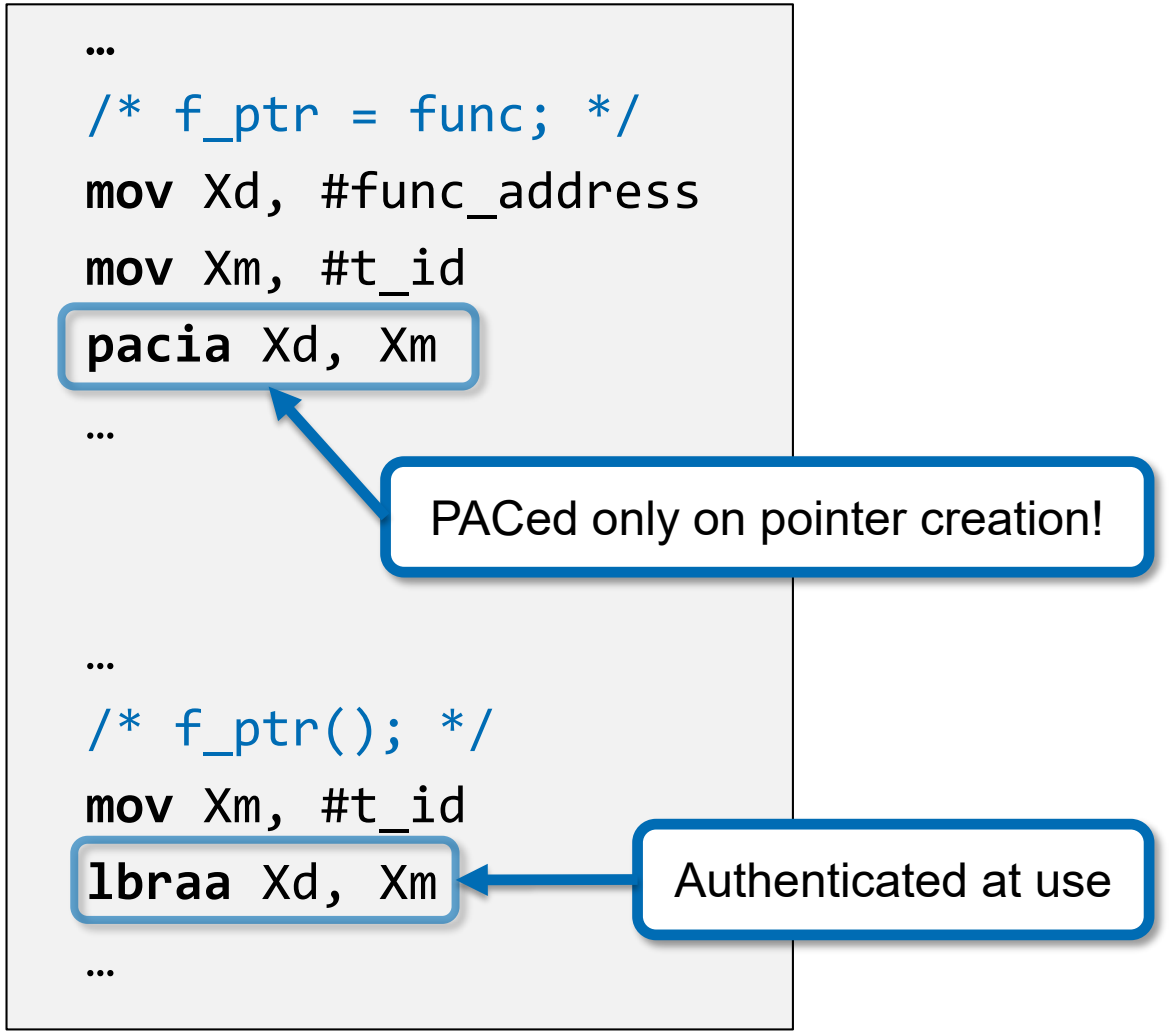
Code pointer signing

Modifier: type-id

- Assigned at compile-time
- Based on LLVM ElementType
  - ≈ function signature

Uses on-use authentication

- With combined auth+branch instr.



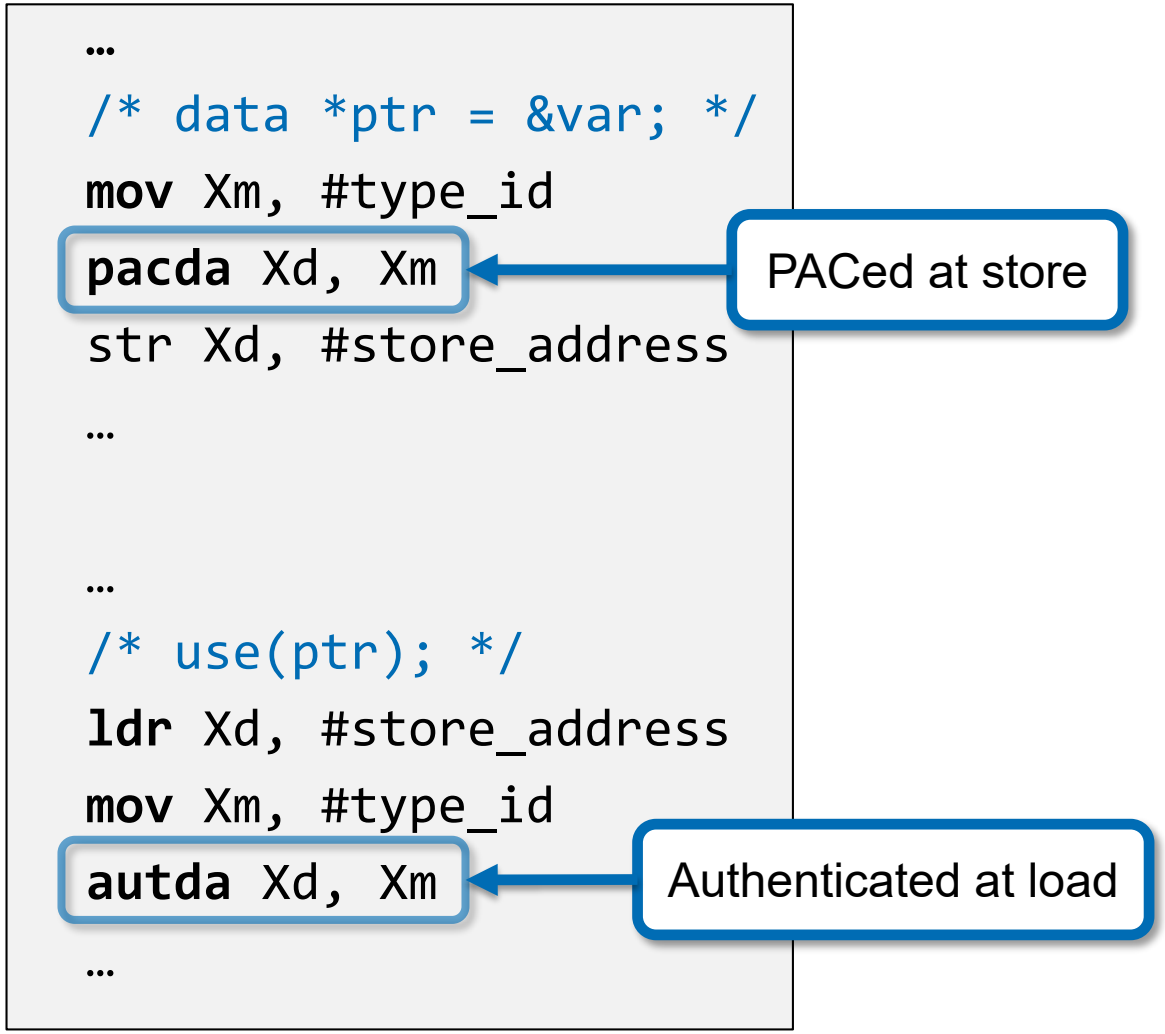
Data pointer signing

Modifier: type-id

- Assigned at compile-time
- Based on IR ElementType
  - ≈ data type

Uses on-load authentication

- always auth on load



Brute-forcing PACs

Wrong PAC causes process crash

- Recall: PAC keys reset on process start
- Probability *p* of guessing *b*-bit PAC correctly after *n* tries:

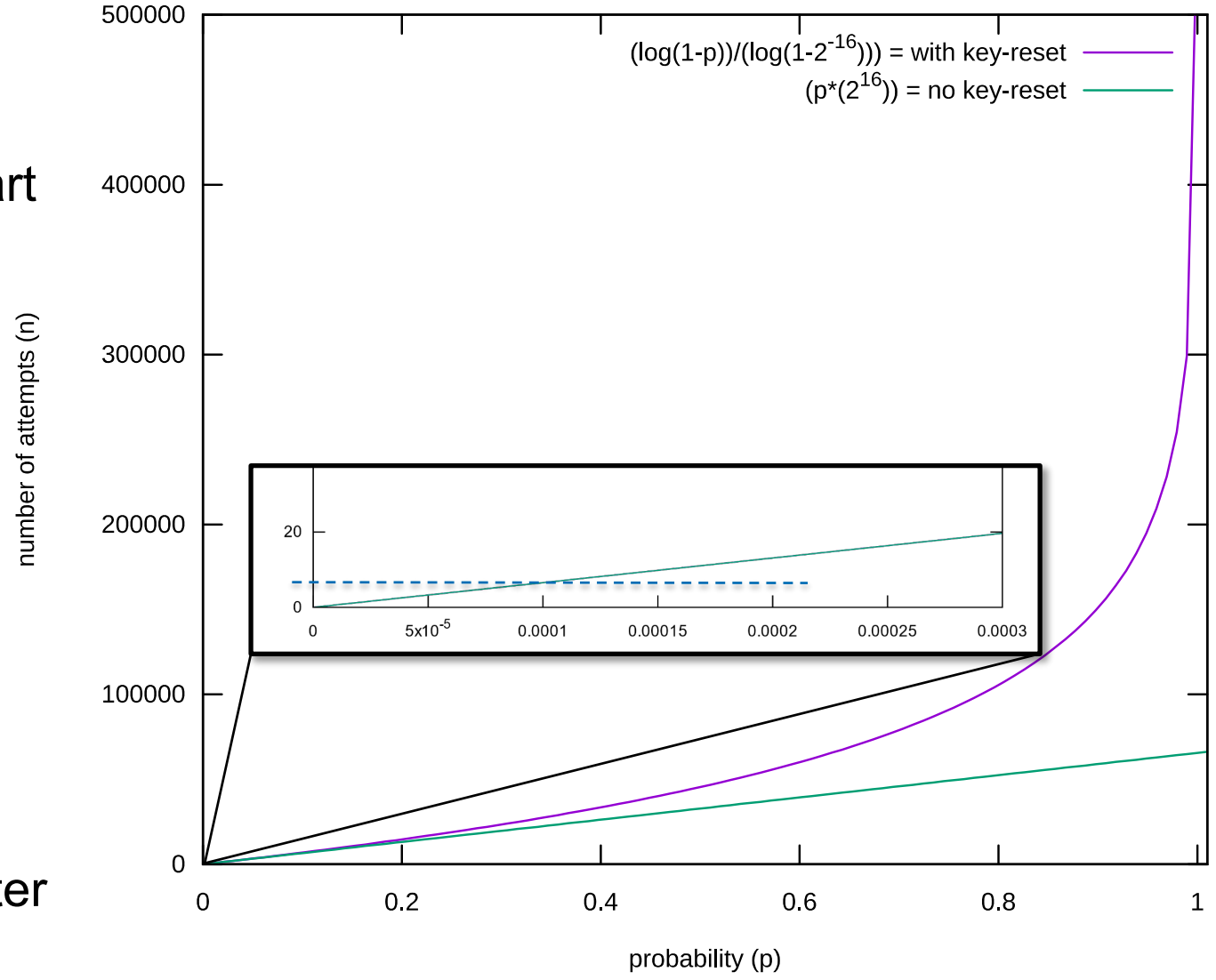
$$1 - p = (1 - 2^{-b})^n$$

- For *b*=16, *p*=0.5, *n* = 45425

Threading/pre-forking is a concern

- Commonly used: e.g., Android Zygote
- Key reset on live process infeasible
- Restarting siblings+parent disruptive
- **Approach**: Restart siblings+parent after threshold number of crashes

Key reset does not offer significant gains at low p values

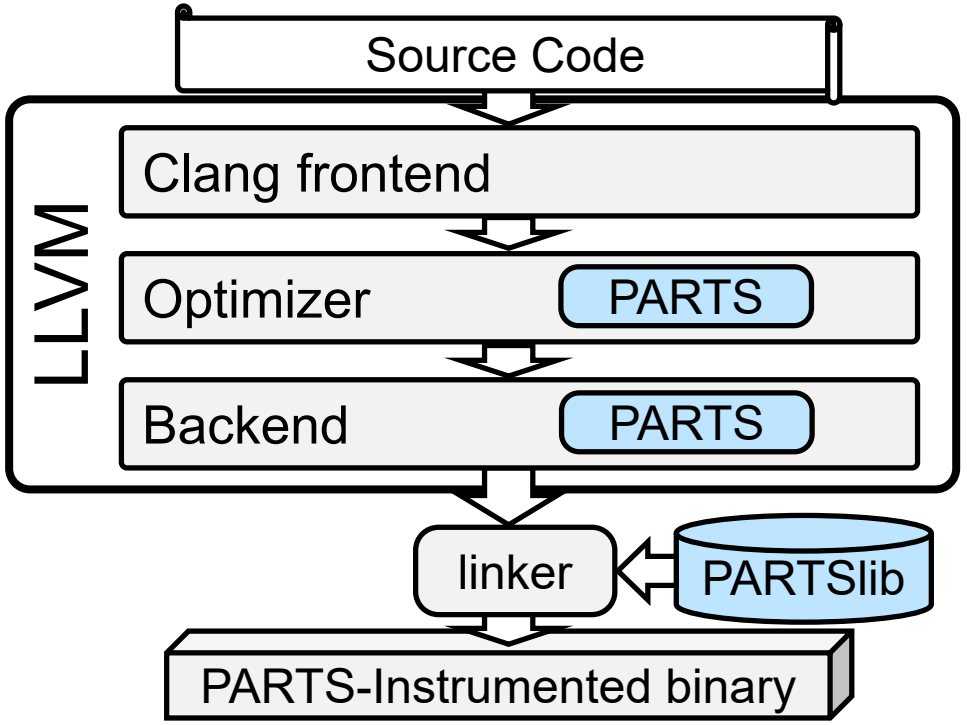


Implementation

PARTS implementation architecture

- **LLVM based compile-time instrumentation**
  - Optimizer passes
  - AArch64 backend specific changes

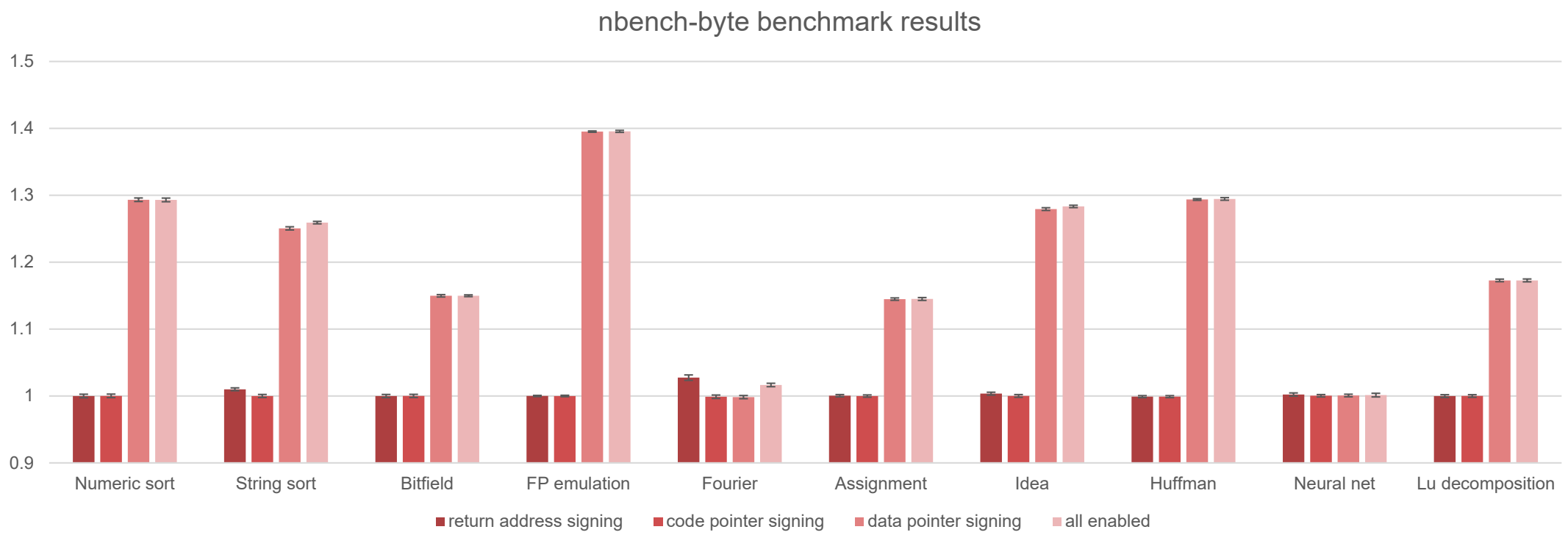
- **Uses PA to protect:**
  - Return addresses on the stack
  - Local, global, and static pointers
  - Pointers in C structures



Evaluation: nbench performance

Reasonable overhead (geom.mean)

- **Combined return address and code pointer signing < 0.5%**
- **Data pointer signing ~19.5%**



Technical report & source code

PAC it up: Towards Pointer Integrity using ARM Pointer Authentication
accepted to USENIX Security 2019
Research report version available at arxiv.org/abs/1811.09189

Compiler and code samples *(will appear at)*:

github.com/pointer-authentication

arxiv.org/abs/1811.09189

github.com/pointer-authentication

PARTS: next steps

- **Threat surface:** estimate prevalence of pointer reuse scenarios in real-world programs
- **Performance**: evaluate on real hardware
- **Generality**: extend to C++
  - How to handle C++ class hierarchies?
  - Can we protect C++ exception handling?
  - Other C++ specific features?
- **Extensions:** (how) can we use PA towards achieving full memory safety?

HardScope vs. Pointer Authentication

Share the same high-level objective but take entirely different approaches

Enforce data-flows between fine-grained subjects by scope enforcement

vs

Secure data-flows by ensuring integrity of pointers

HardScope: Challenges to acceptance

Hardware changes (even minimal ones) pose a major hurdle

Backward compatibility vs security

- Dynamic (non-continuous) data structures, global pointers, …

Scalability vs. extent of problem

- For embedded domain: low #rules/domain, but are data-oriented attacks a real concern?

RISC-V vs. real deployment

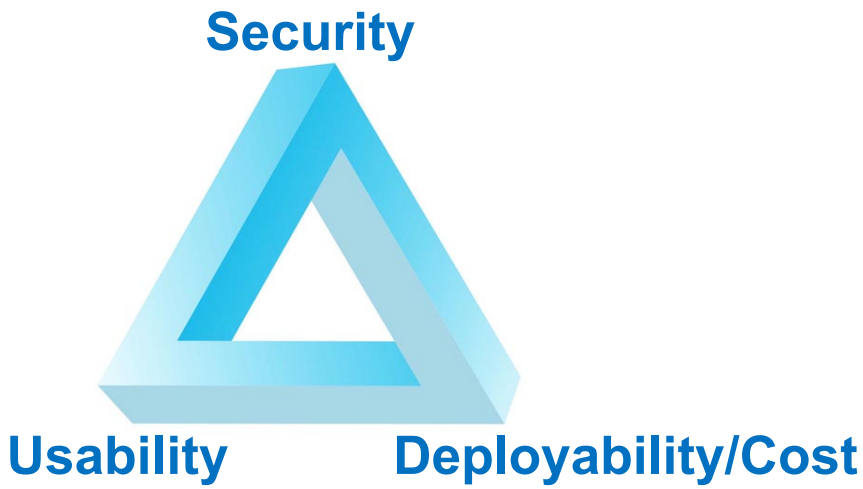
Hardware-assisted run-time protection: the promise

Pointer Authentication is powerful

- **What are other creative uses of PA?**

Other hardware primitives in the pipeline

- **Memory Tagging**
- **Branch Target Indication**



Research Institute for Collaborative Autonomous and Resilient Systems

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

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