Aalto University

Design across layers

achieving more by joining hardware, software, and cryptography

Lachlan J. Gunn

including work by **N Asokan, Jan-Erik Ekberg, Setareh Ghorshi, Hans Liljestrand, Thomas Nyman**

lachlan.gunn.ee

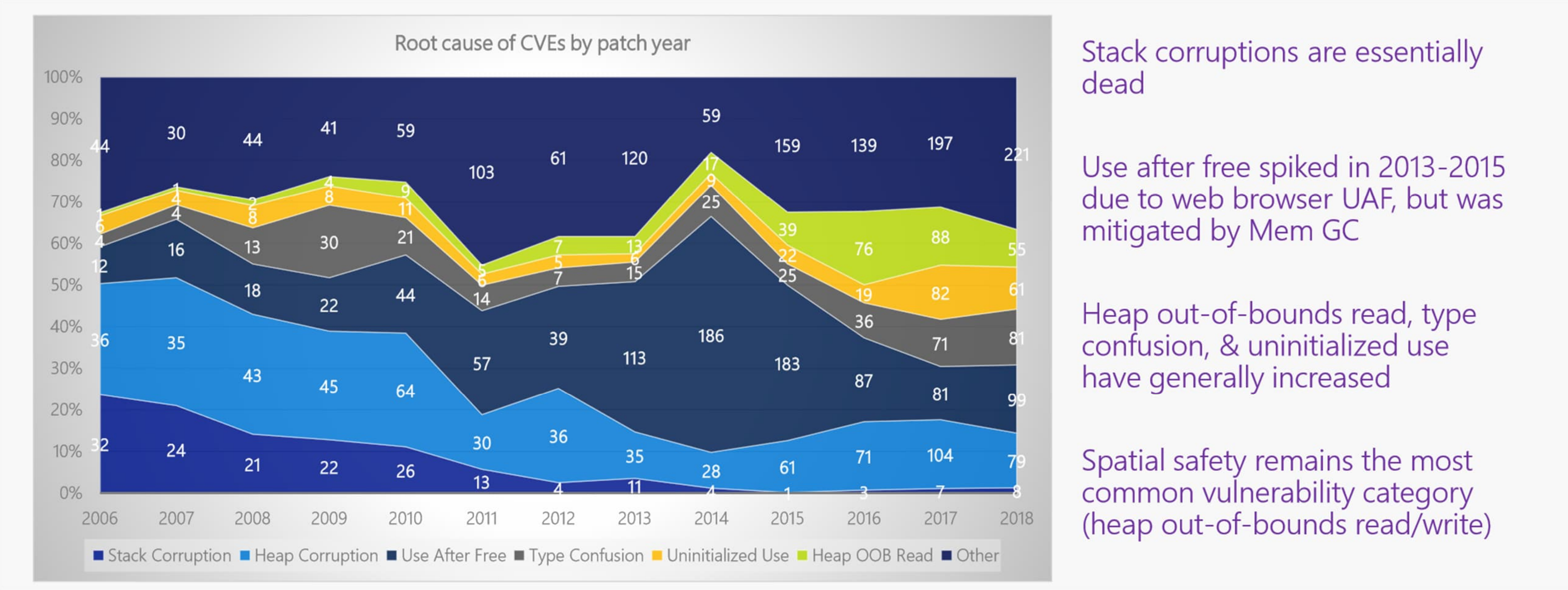
@lachlan_gunn

NANDA, London, UK, 2023-09-11

The problem

Memory corruption vulnerabilities are a persistent problem

- Microsoft: consistently 70% of CVEs



Source: Matt Miller, "Trends, Challenges, and Strategic Shifts", BlueHat IL 2019.

Software run-time protection

Memory vulnerabilities can give arbitrary read/write access to memory

Software-based defences helpful but limited

- Canaries
- Software-based control-flow integrity

Problem: Attackers can use software vulnerabilities to attack software-based defences

Solution: Implement defences in hardware, safe from vulnerable software

- Write ^ Execute
- Memory protection
- Address space layout randomisation

Cryptographic run-time protection

Problem: Hardware is inflexible

Solution: Multi-purpose hardware primitive that can be used by software in many different ways

In this talk: can cryptography protect data in memory?

- Modern CPUs provide acceleration:
    - Intel AES-NI
    - ARM Pointer Authentication

Goal: Protect sensitive functionality from vulnerabilities elsewhere in the program

Useful Assumptions

W^X

Executable code cannot be modified

Useful Assumptions

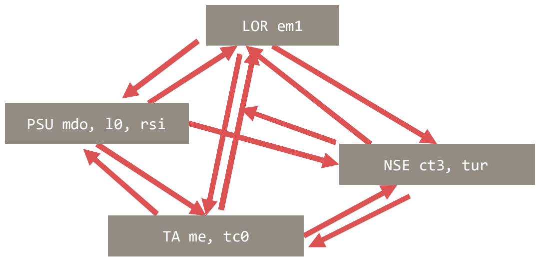
W^X

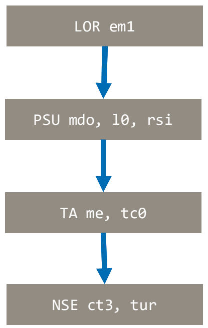
Executable code cannot be modified

Control flow integrity

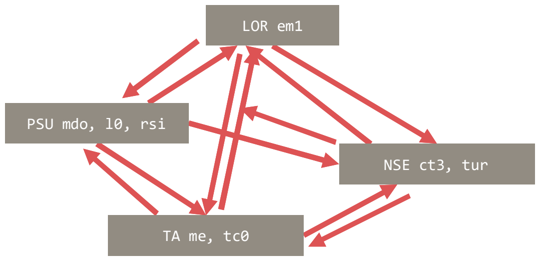
Attacker can’t make program jump to just anywhere

- Direct branches jump to designated addresses
- Calls to function pointers always jump to beginning of functions





Useful Assumptions



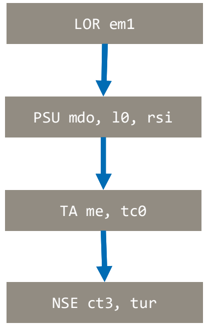
W^X

Executable code cannot be modified

Control flow integrity

Attacker can't make program jump to just anywhere

- Direct branches jump to designated addresses
- Calls to function pointers always jump to beginning of functions



Register safety

Attacker can't modify registers except by following the program

- Registers part of instruction encoding: can't change by modifying data in memory
- One register file per thread: no interference from other threads

Program model

Program is split into basic blocks

- Linear instructions followed by control flow instruction

```
func0:
  sub    r1, r3, r
  xor    r5, r2, r
  jmp    func2
```

```
func1:
  add    r1, r2, r3
  and    r1, r4, r1
  jmp    func2
```

```
func2:
  store   r5, r1
  load    r1, r8
  syscall
```

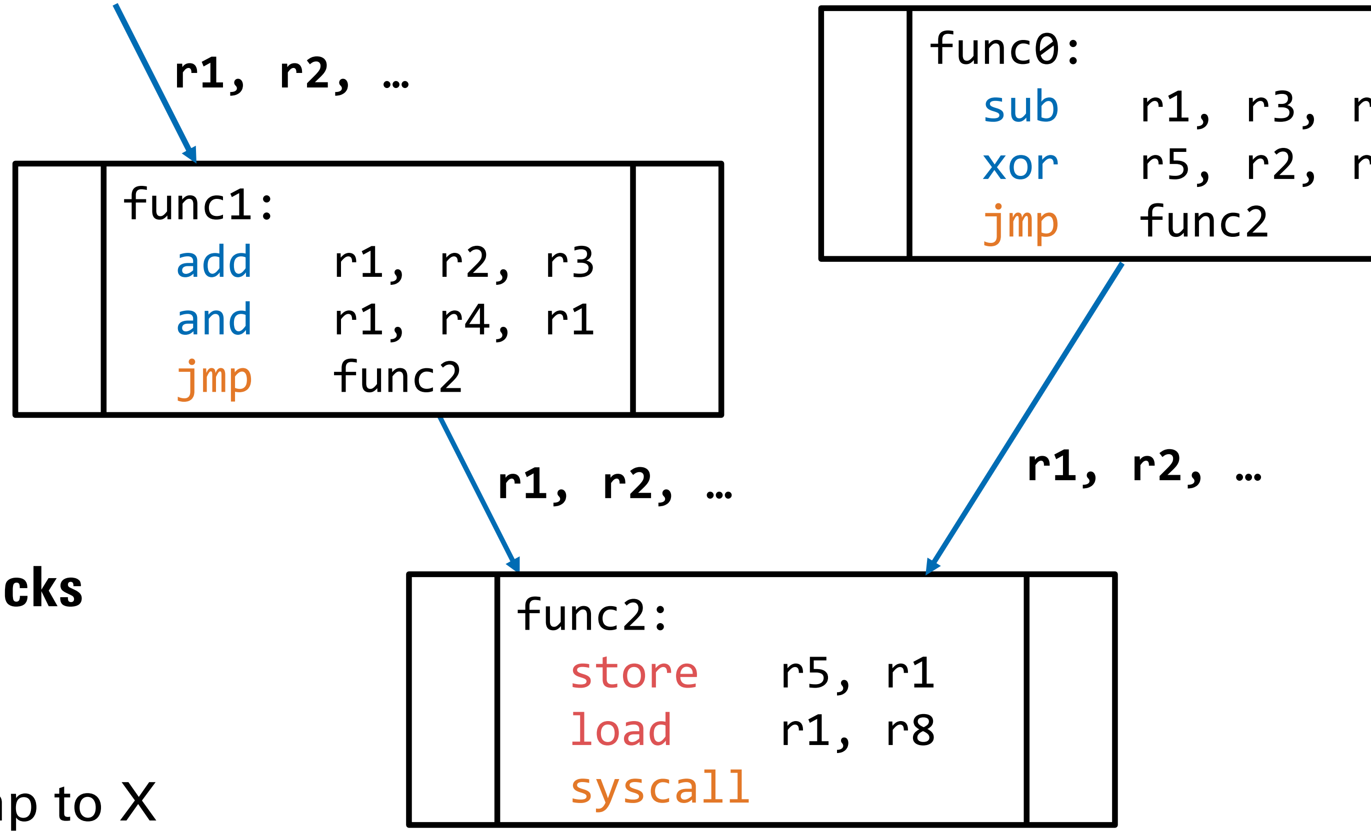
Program model

Program is split into basic blocks

- Linear instructions followed by control flow instruction

Registers provide secure channel between blocks

- Limited communication volume
- Initial state before block X = final state after a block that can jump to X



Program model

Program is split into basic blocks

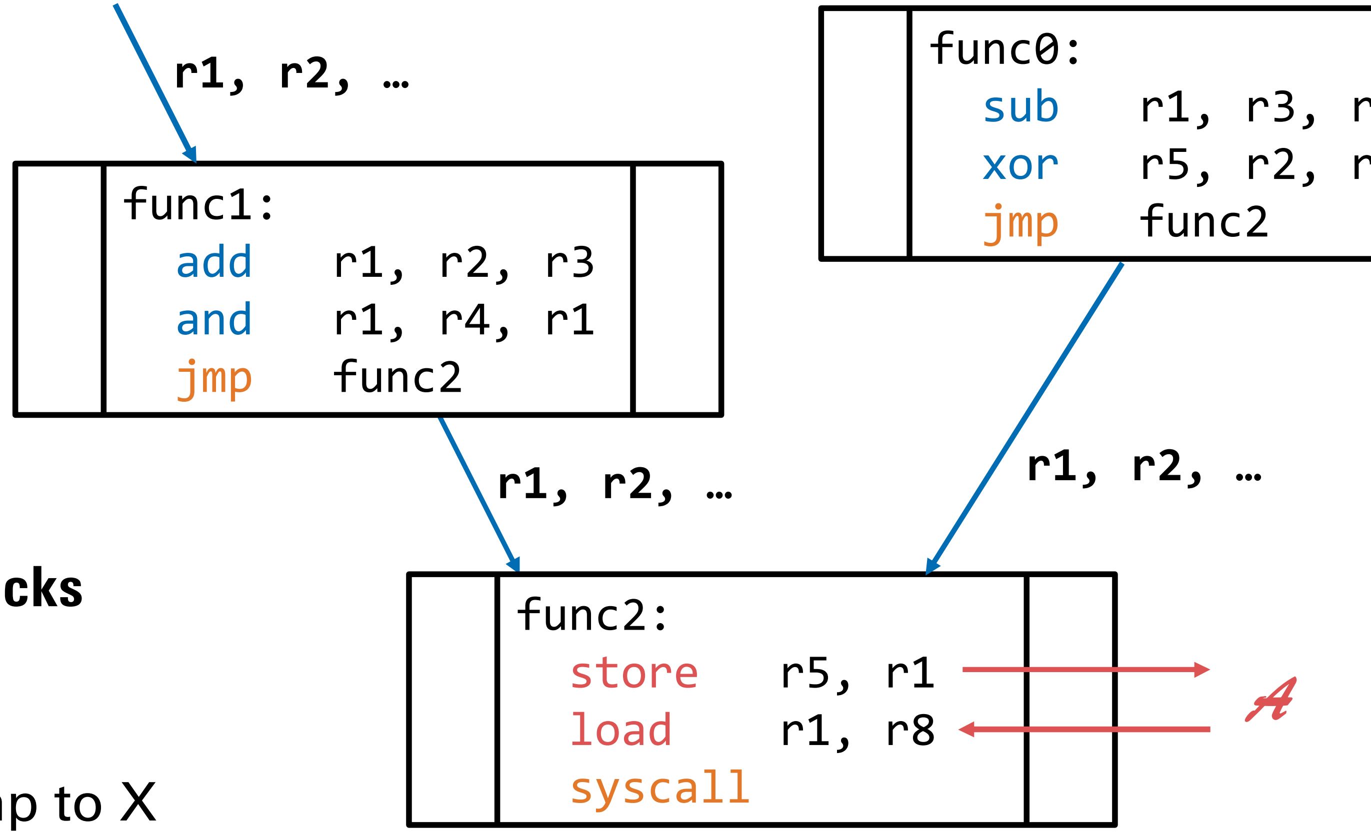
- Linear instructions followed by control flow instruction

Registers provide secure channel between blocks

- Limited communication volume
- Initial state before block X = final state after a block that can jump to X

Memory controlled by the attacker

- Loads and stores become interactions with $\mathcal{A}$

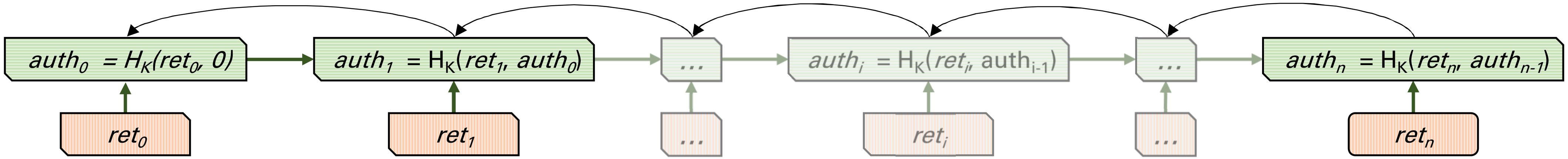


S Ghorshi, L J Gunn, H Liljestrand, N Asokan, "Towards cryptographically authenticated in-memory data structures", IEEE SecDev'22

Functionality #1: Secure Stack

Goal: Store return address stack **in memory**

Approach: store **MAC chain** of return address authentication tokens



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

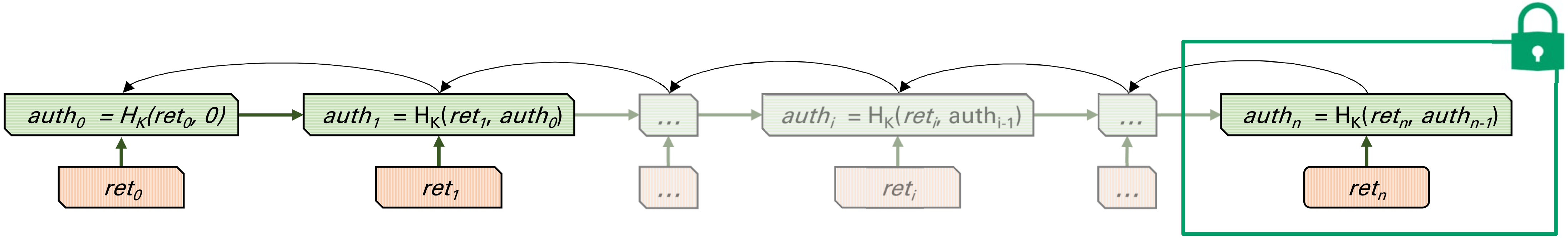
H Liljestrand, T Nyman, L J Gunn, J-E Ekberg, N Asokan, "PACStack: An authenticated call stack", USENIX Security 2021.

Functionality #1: Secure Stack

Goal: Store return address stack in memory

Approach: store MAC chain of return address authentication tokens

- Single authentication token kept in register authenticates entire return address stack



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

H Liljestrand, T Nyman, L J Gunn, J-E Ekberg, N Asokan, "PACStack: An authenticated call stack", USENIX Security 2021.

Cryptographic analysis

We reduced the stack's security to MAC collision probability

Challenge: MAC collisions occur on average after $1.253 * 2^{b/2}$ return addresses

- For $b = 16$, $n = 321$ addresses

H Liljestrand, T Nyman, L J Gunn, J-E Ekberg, N Asokan, "PACStack: An authenticated call stack", USENIX Security 2021.

Cryptographic analysis

We reduced the stack's security to MAC collision probability

Challenge: MAC 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 XOR-red with *auth*
- Wrong guesses result in segfault

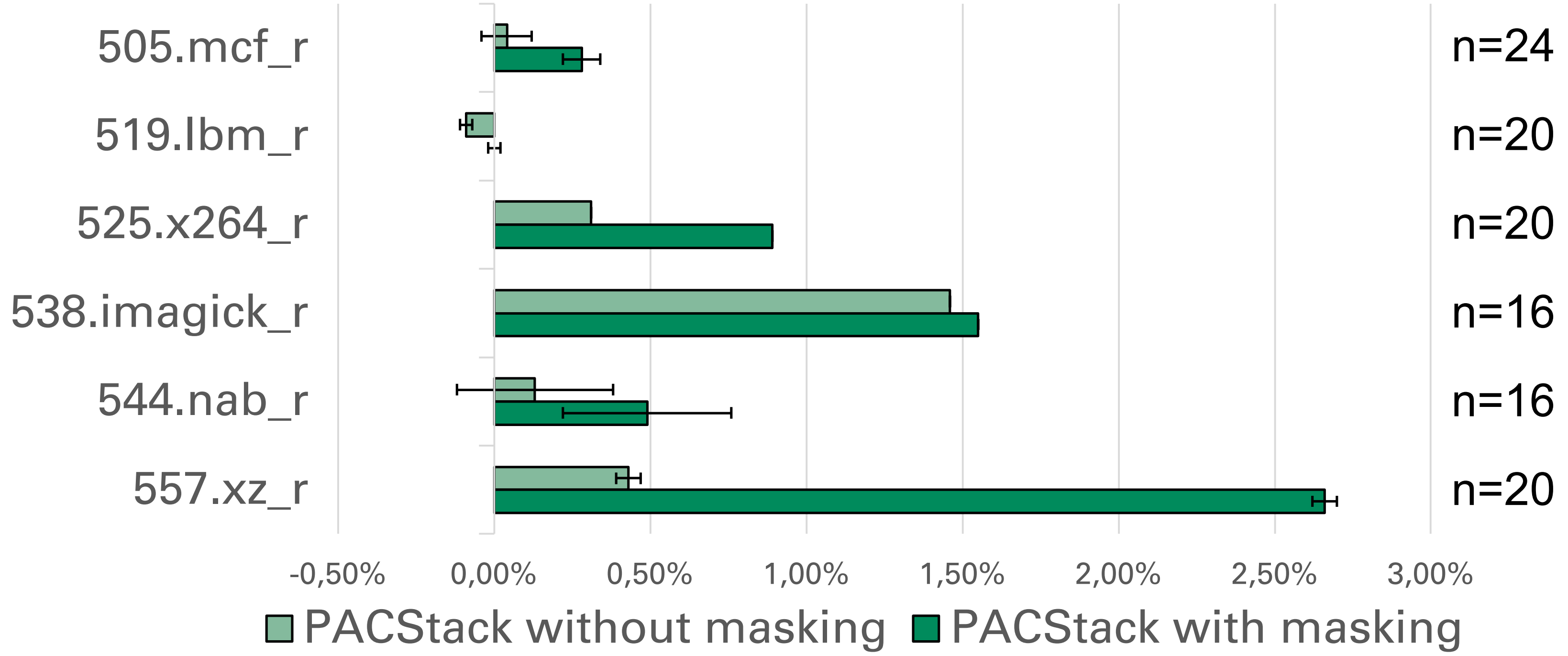
Attack	Success w/o Masking	Success w/ Masking
Reuse previous auth collision	1	2^{-b}
Guess auth to existing call-site	2^{-b}	2^{-b}
Guess auth to arbitrary address	2^{-2b}	2^{-2b}

H Liljestrand, T Nyman, L J Gunn, J-E Ekberg, N Asokan, "PACStack: An authenticated call stack", USENIX Security 2021.

Evaluation: SPEC CPU 2017 C-language benchmarks

Estimated performance overhead based on 4-cycles per PA instruction

- without masking < 0.5% (geo.mean)
- with masking < 1% (geo.mean)



H Liljestrand, T Nyman, L J Gunn, J-E Ekberg, N Asokan, "PACStack: An authenticated call stack", USENIX Security 2021.

Protecting other program data

PACStack only protected return address stack

- Specialised mechanism for a specialised data structure

Can we protect general program data structures?

Challenges:

1. Wide variety of data structures with different performance expectations
2. Limited number of protected registers for arbitrarily many data structures
3. How to stop bad data being stored in the first place?

S Ghorshi, L J Gunn, H Liljestrand, N Asokan, "Towards cryptographically authenticated in-memory data structures", IEEE SecDev'22

Authenticated data structures

Different cryptographic methods have different performance characteristics

Hash chain: O(1) access at one end, O(*size*) random access

- Useful for stacks

Merkle tree: O(log *size*) random access

- Useful for trees, vectors, etc.

Each data structure implementation reduces its contents to a single "top MAC"

- Merkle tree reduces all top MACs to a thread-global MAC kept in register

S Ghorshi, L J Gunn, H Liljestrand, N Asokan, "Towards cryptographically authenticated in-memory data structures", IEEE SecDev'22

Functionality #2: Secure Queue

First-In-First-Out (FIFO) order

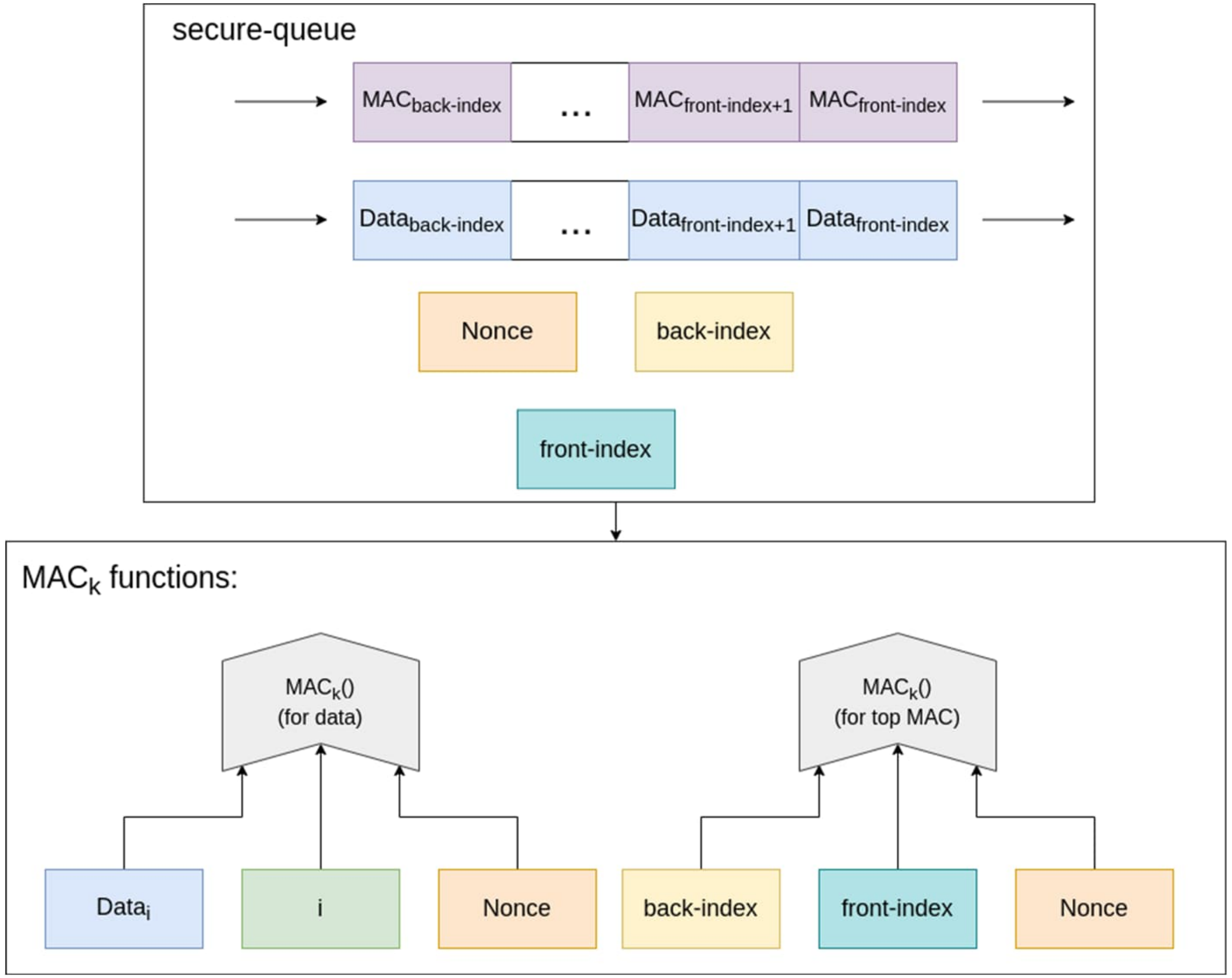
- O(1) Read/write from front/back

Hash chains no good here

- Need to be modified at both ends
- Chains need O(*size*) to update

Queue-specific approach

- Data MACs tie data to insertion order
- Top MAC authenticates head/tail indices
- Achieves normal O(1) performance



S Ghorshi, L J Gunn, H Liljestrand, N Asokan, "Towards cryptographically authenticated in-memory data structures", IEEE SecDev'22

Performance

Microbenchmarks:

Data Structure	Number of operations	Secure	Unmodified	Overhead
Stack<int>	1000	16 853.65 µs	11.21 µs	1503 x
Queue<int>	1000	16 793.65 µs	11.13 µs	1508 x
Red-Black Tree<string, string>	10	553 959.23 µs	150.63 µs	3676 x

OpenCV performance tests:

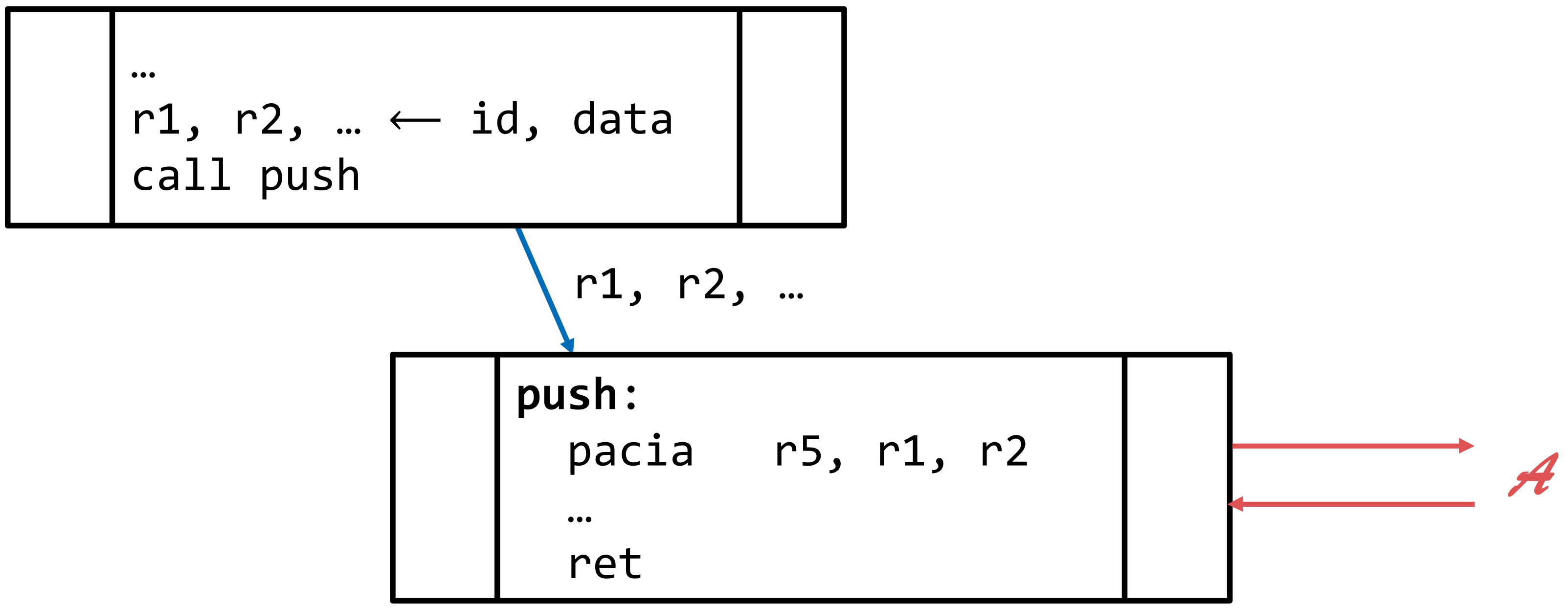
- 3.42% overhead
- 6.42% with secure random access

S Ghorshi, L J Gunn, H Liljestrand, N Asokan, "Towards cryptographically authenticated in-memory data structures", IEEE SecDev'22

Design challenges

Challenge: How to stop bad data being stored in the data structure?

Best solution: Pass data to protocol implementation via registers

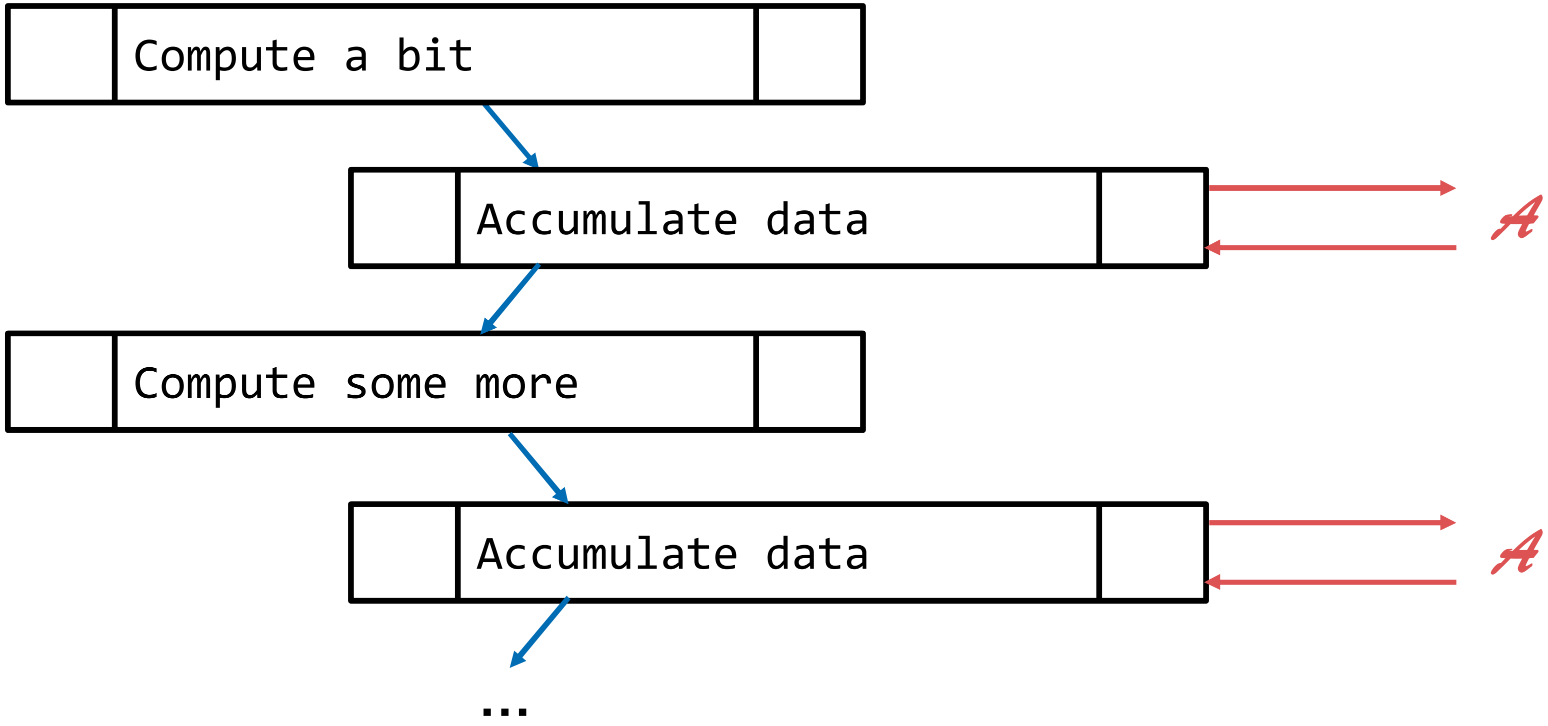


Challenge: Not all types fit into registers

Design challenges

Challenge: **Not all types fit into registers**

Workaround: **Use coroutine-like "streaming" implementations**



Easier solution: **weaken the adversary model**

Different adversaries

Fast **$\mathcal{A}$ can write to memory at any time**

Strongest attacker in a multithreaded setting.

Slow **$\mathcal{A}$ can write to memory, but too imprecisely to do so between ops in a single basic block**

Models an attacker in a multithreaded program who can't easily synchronise between threads.

Single **$\mathcal{A}$ can write to memory, but only when the program counter is at a vulnerable address**

Models an attacker exploiting vulnerabilities in a single-threaded program.

S Ghorshi, L J Gunn, H Liljestrand, N Asokan, "Towards cryptographically authenticated in-memory data structures", IEEE SecDev'22

Remaining challenges

Adversary models

- $\mathscr{A}$ – Slow definition is a bit arbitrary; is there a better alternative?

General computation

- How can we produce generic code that is safe under -Slow and -Fast models?
- Many computations can't fit data into registers
- Compiler must emit code that cryptographically protects working storage

Multithreading

- How can we share authenticated data between threads?

No time to present this now, but ask me later about

Blinded Memory

Joint work with N. Asokan, Hossam ElAtali, Hans Liljestrand

Takeaways

Cryptography with hardware primitives can
secure critical functionality in vulnerable software

Crypto accelerators in current CPUs make this viable

- ~1% overhead for PACStack

gunn.ee