Beyond Software Defenses:
Hardware-assisted CFI on Arm-based systems

Michalis Pappas
mpappas@fastmail.fm

HKOSCon 2024

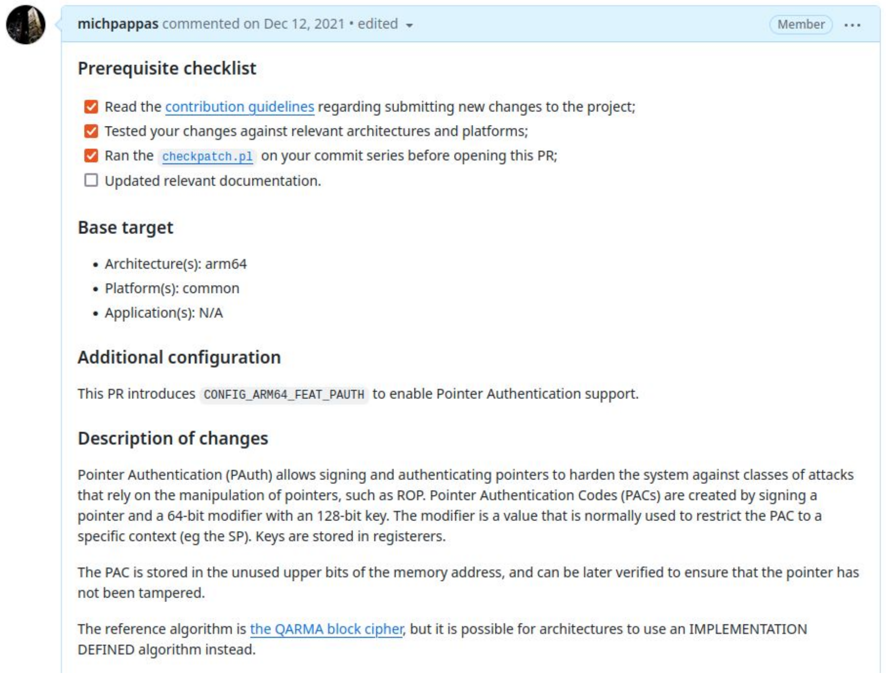
About

Michalis Pappas

Staff Engineer at Unikraft GmbH

https://github.com/michpappas

Motivation: Hardware-assisted CFI on Unikraft for arm64



michpappas commented on Dec 12, 2021 • edited · Member

Prerequisite checklist

- [x] Read the contribution guidelines regarding submitting new changes to the project;
- [x] Tested your changes against relevant architectures and platforms;
- [x] Ran the `checkpatch.pl` on your commit series before opening this PR;
- [ ] Updated relevant documentation.

Base target

- Architecture(s): arm64
- Platform(s): common
- Application(s): N/A

Additional configuration

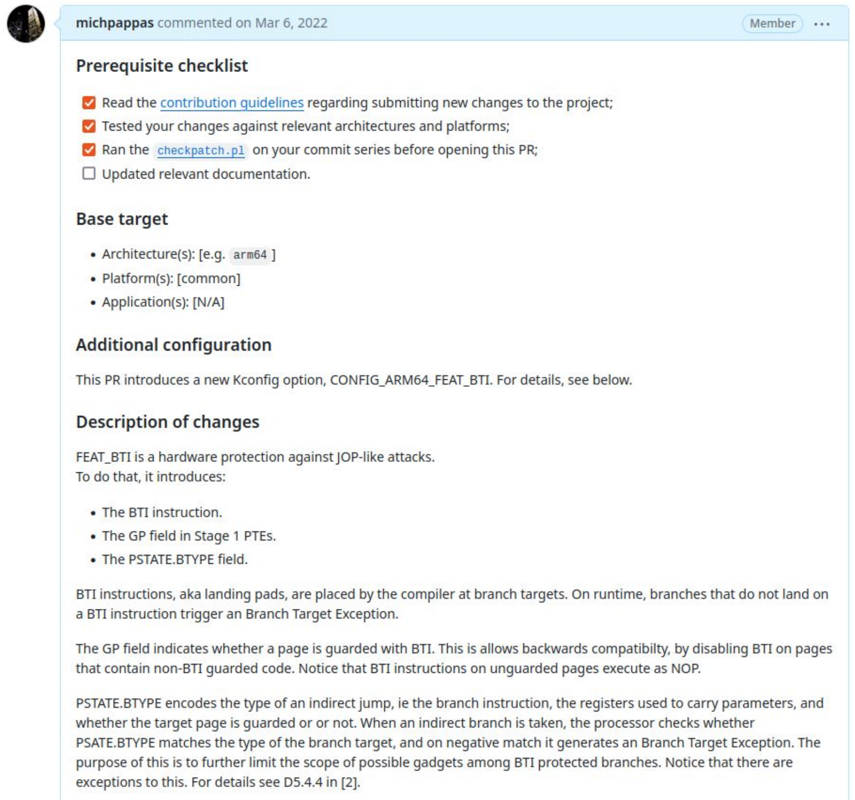
This PR introduces `CONFIG_ARM64_FEAT_PAUTH` to enable Pointer Authentication support.

Description of changes

Pointer Authentication (PAuth) allows signing and authenticating pointers to harden the system against classes of attacks that rely on the manipulation of pointers, such as ROP. Pointer Authentication Codes (PACs) are created by signing a pointer and a 64-bit modifier with an 128-bit key. The modifier is a value that is normally used to restrict the PAC to a specific context (eg the SP). Keys are stored in registerers.

The PAC is stored in the unused upper bits of the memory address, and can be later verified to ensure that the pointer has not been tampered.

The reference algorithm is the QARMA block cipher, but it is possible for architectures to use an IMPLEMENTATION DEFINED algorithm instead.



michpappas commented on Mar 6, 2022 · Member

Prerequisite checklist

- [x] Read the contribution guidelines regarding submitting new changes to the project;
- [x] Tested your changes against relevant architectures and platforms;
- [x] Ran the `checkpatch.pl` on your commit series before opening this PR;
- [ ] Updated relevant documentation.

Base target

- Architecture(s): [e.g. `arm64`]
- Platform(s): [common]
- Application(s): [N/A]

Additional configuration

This PR introduces a new Kconfig option, CONFIG_ARM64_FEAT_BTI. For details, see below.

Description of changes

FEAT_BTI is a hardware protection against JOP-like attacks.
To do that, it introduces:

- The BTI instruction.
- The GP field in Stage 1 PTEs.
- The PSTATE.BTYPE field.

BTI instructions, aka landing pads, are placed by the compiler at branch targets. On runtime, branches that do not land on a BTI instruction trigger an Branch Target Exception.

The GP field indicates whether a page is guarded with BTI. This is allows backwards compatibilty, by disabling BTI on pages that contain non-BTI guarded code. Notice that BTI instructions on unguarded pages execute as NOP.

PSTATE.BTYPE encodes the type of an indirect jump, ie the branch instruction, the registers used to carry parameters, and whether the target page is guarded or or not. When an indirect branch is taken, the processor checks whether PSTATE.BTYPE matches the type of the branch target, and on negative match it generates an Branch Target Exception. The purpose of this is to further limit the scope of possible gadgets among BTI protected branches. Notice that there are exceptions to this. For details see D5.4.4 in [2].

(Non-exhaustive) evolution† of CFI attacks & mitigations

- 👹 **1988:** The Morris worm exploits a buffer overflow in the wild
  - 👹 1996: “Smashing The Stack For Fun And Profit” Aleph One, Phrack 49

- 👮 **1997:** Solaris switches to non-executable stacks
  - 👹: 1997: "Getting around non-executable stack (and fix)" Solar Designer BUGTRAQ
  - 👮 1998: FreeBSD switches to non-executable stacks

- 👮 **1998:** “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks” Conowan et al, USENIX
- 👮 **1999:** Stack Shield: A ”stack smashing” technique protection tool for Linux
  - 👹 2000: “Bypassing StackGuard and StackShield” Bulba and Kil3r, Phrack 52
  - 👮 2003: “ProPolice: GCC Extension for Protecting Applications from Stack-Smashing Attacks” Hiroaki Etoh IBM
  - 👮 2004: GCC 3.x introduces stack protector (SSP) based on ProPolice

- 👮 **2001:** PaX introduces ASRL
  - 👹 2002: “Bypassing PaX ASLR protection” Tyler Durden, Phrack 59
  - 👮 2003: OpenBSD adds ASLR
  - 👮 2005: Linux adds ASLR based on PaX

- 👮 **2003:** AMD introduces NX bit
  - 👮 2004: Intel adds NX bit
  - 👮 2007: Arm introduces WXN

- 👹 **2008:** Return-oriented programming (RoP) / Jump-oriented programming (JoP)
  - 👹 "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86 Architecture)" Hovav Shacham
  - 👹 “Return-Oriented Programming: Exploits Without Code Injection” Stefan Savage et al

- 👮 **2013:** PaX introduces KASLR
  - 👹 Attackers have demonstrated numerous ways to bypass KASLR (https://github.com/bcoles/kasld)

- 👮 **2016:** Arm and Intel announce hardware-assisted CFI
  - 👮 Arm: PAuth and BTI
  - 👮 Intel: CET & IBT

Hardware-assisted Control Flow Integrity (CFI)

Hardware-assisted CFI in Arm:

- Pointer Authentication (PAuth): Enforces CFI by authenticating pointers.
- Branch Target Identification (BTI): Enforces CFI by restricting indirect jumps.

Threat model:

- Attacker has arbitrary READ & WRITE and is aiming for arbitrary EXEC.
- Non-executable stack in place.

Pointer Authentication (PAuth)

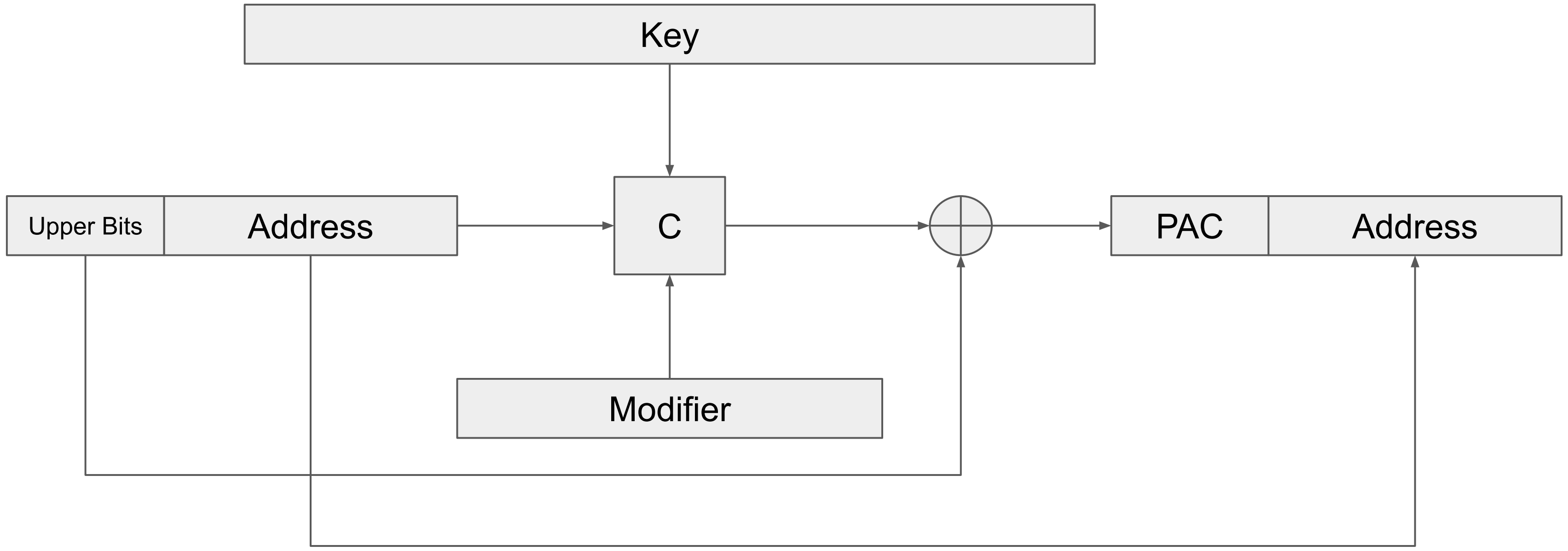
Pointer Authentication (PAuth)

Based on cryptographic signing and authentication of pointers.

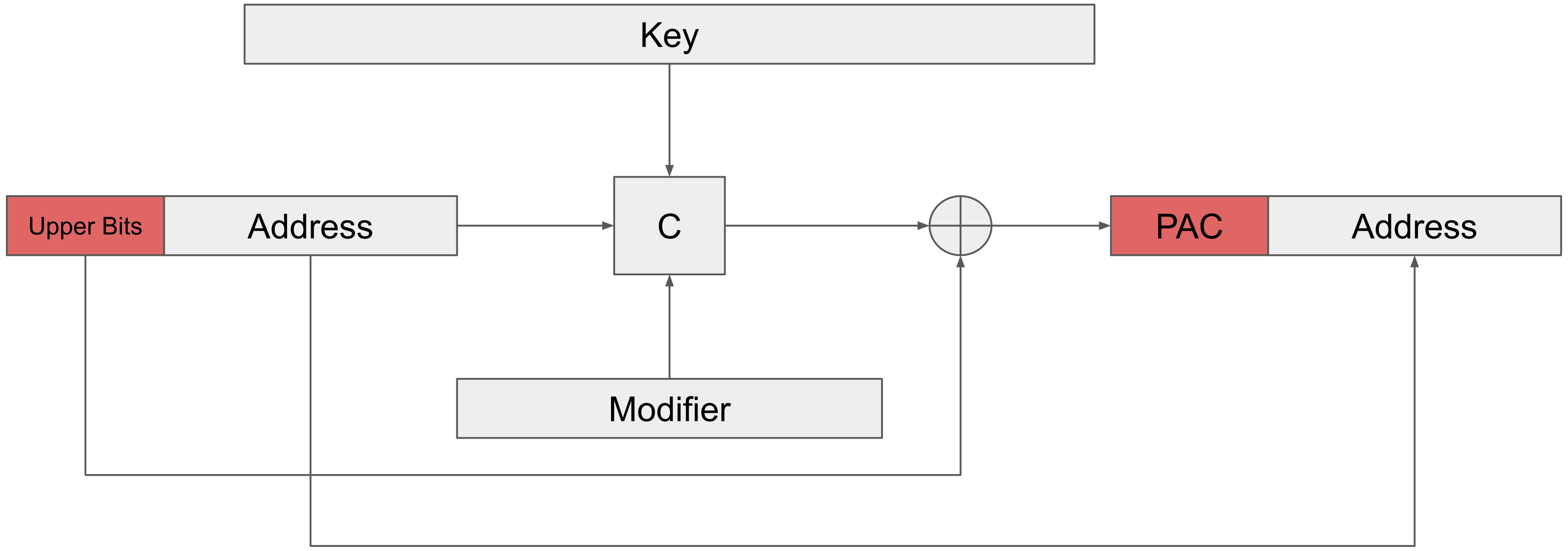
- **Armv8.3-A** introduces **FEAT_PAuth** as a **MANDATORY** extension.
- **Armv8.6-A** introduces **FEAT_PAuth2** as a **MANDATORY** extension.

In this talk we will be focusing on **FEAT_PAuth2.**

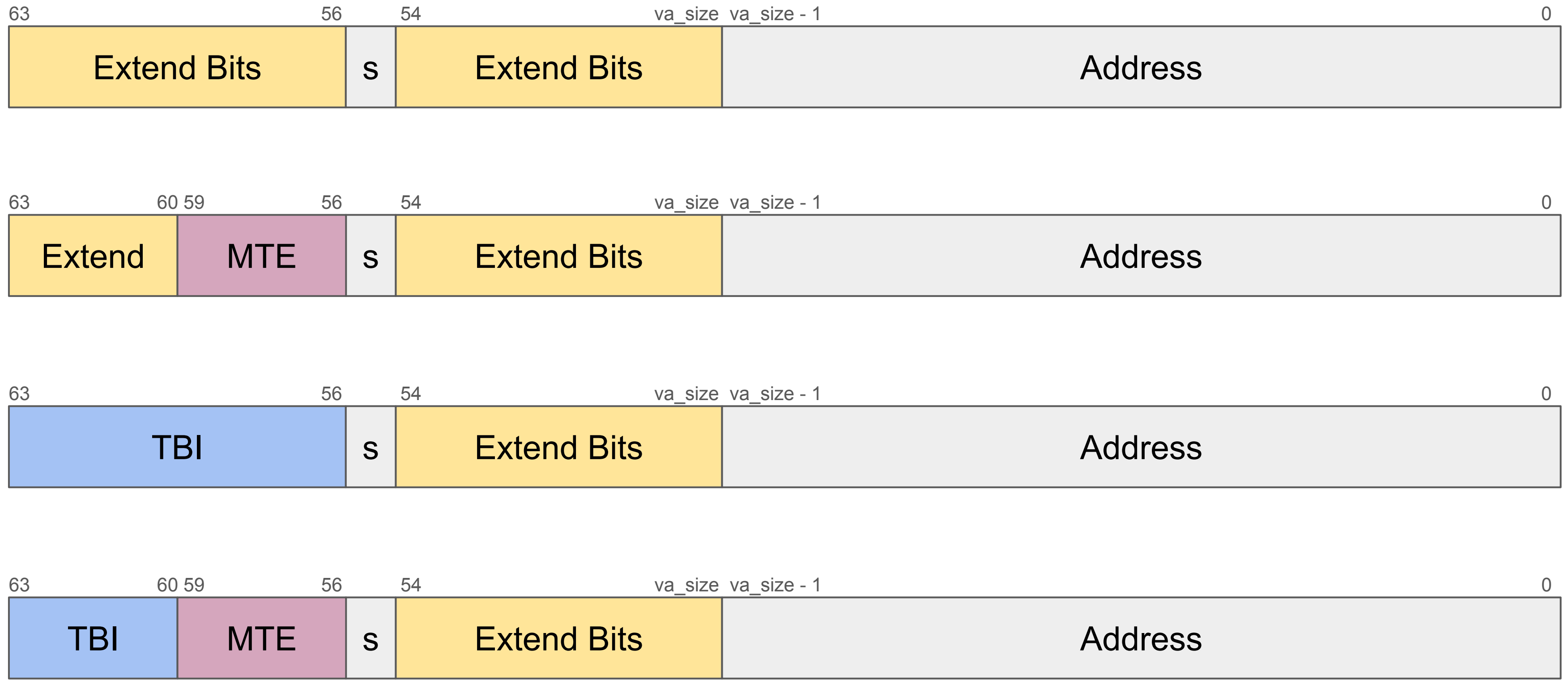
PAuth Operation: Sign



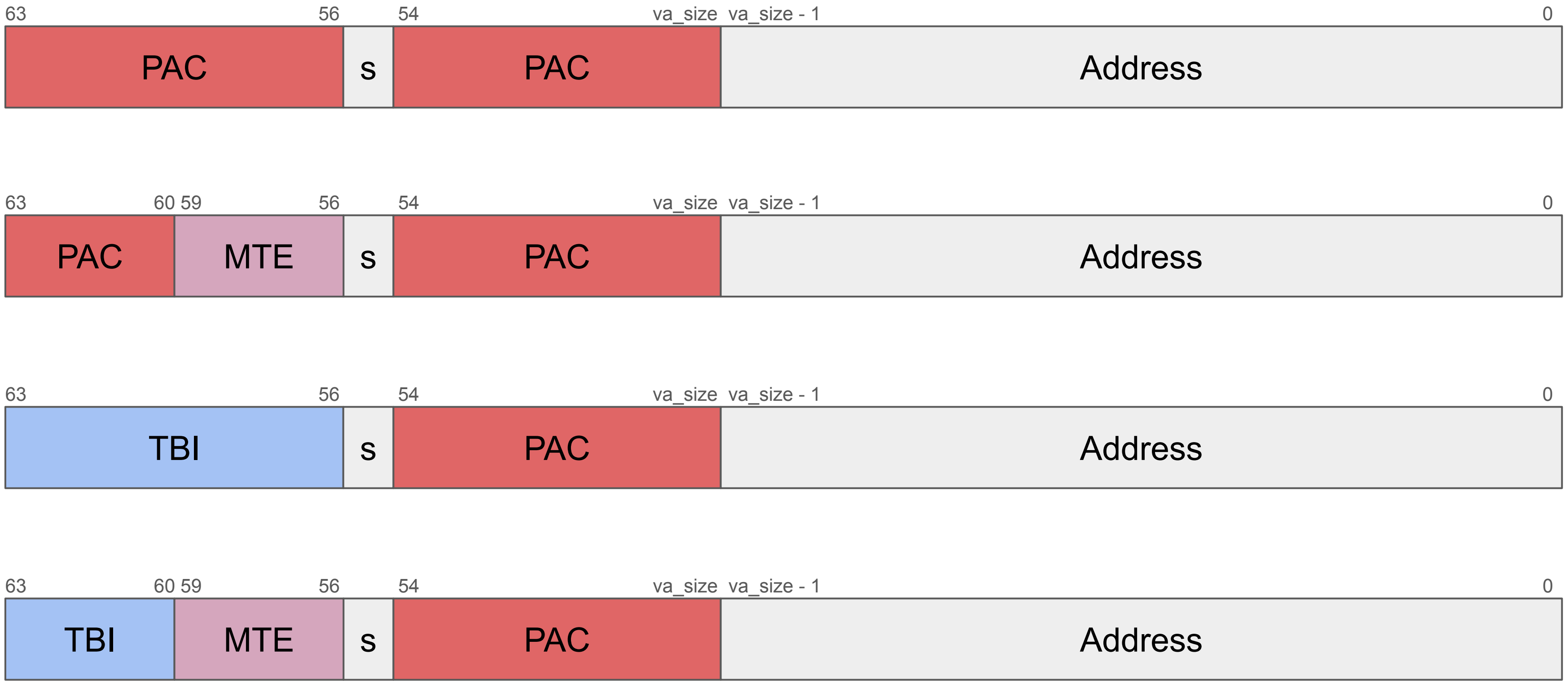
PAuth: Upper Bits / PAC



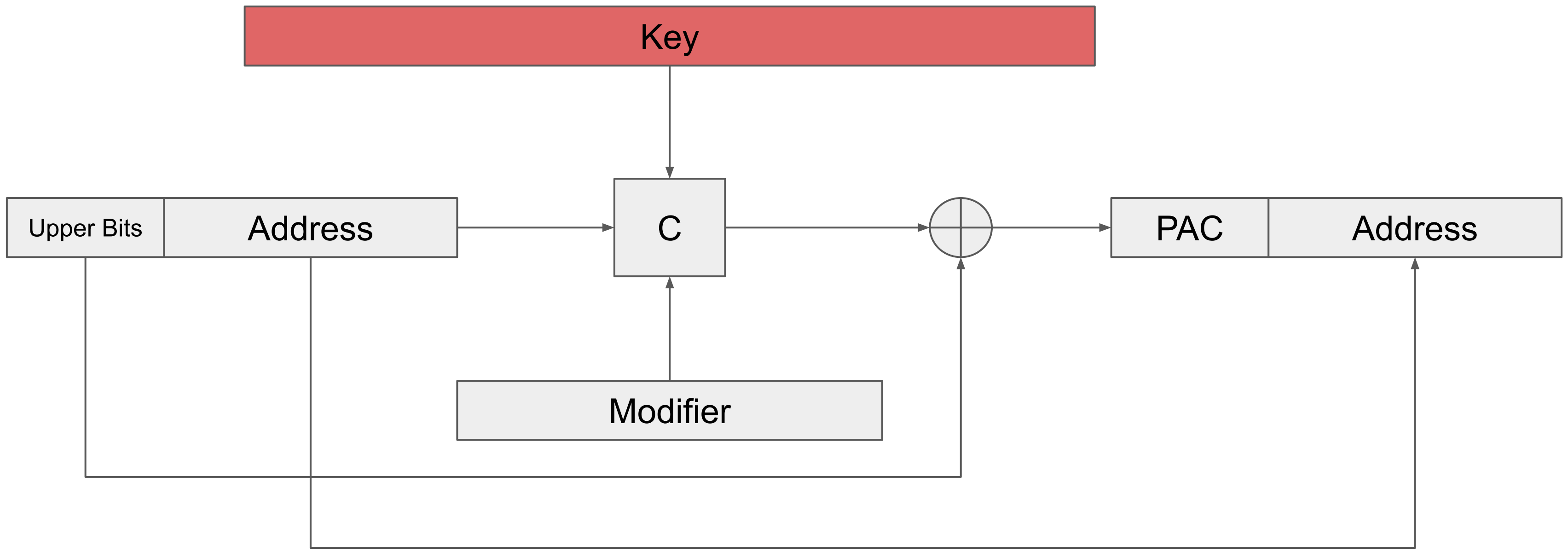
PAuth: Upper Bits / PAC



PAuth: Upper Bits / PAC



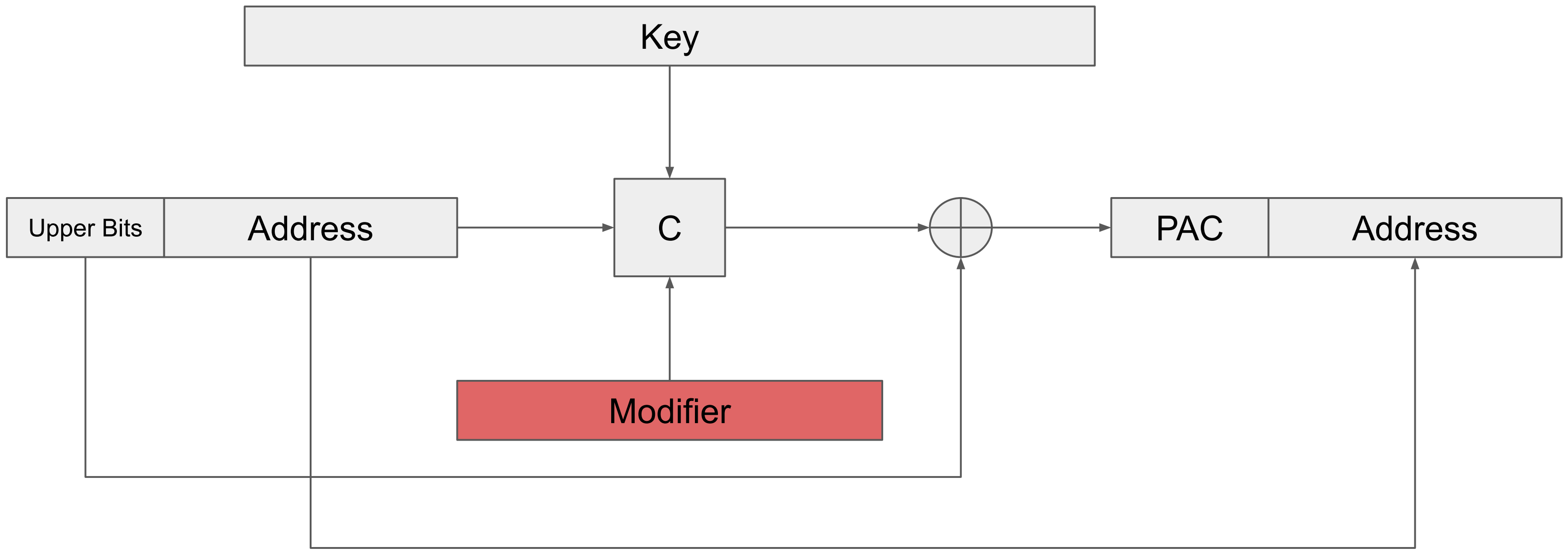
PAuth: Key



PAuth: Key Registers

Key	Purpose	Mnemonic
APIAKey_ELx	Instruction signing Key A	IA
APIBKey_ELx	Instruction singing Key B	IB
APDAKey_ELx	Data signing Key A	DA
APDBKey_ELx	Data signing Key B	DB
APGAKey_ELx	Generic singing key	G

PAuth: Modifier



PAuth: Modifier

Problem:

- Pointer substitution attacks.
- An attacker can reuse signed pointers (remember threat model).

Mitigation:

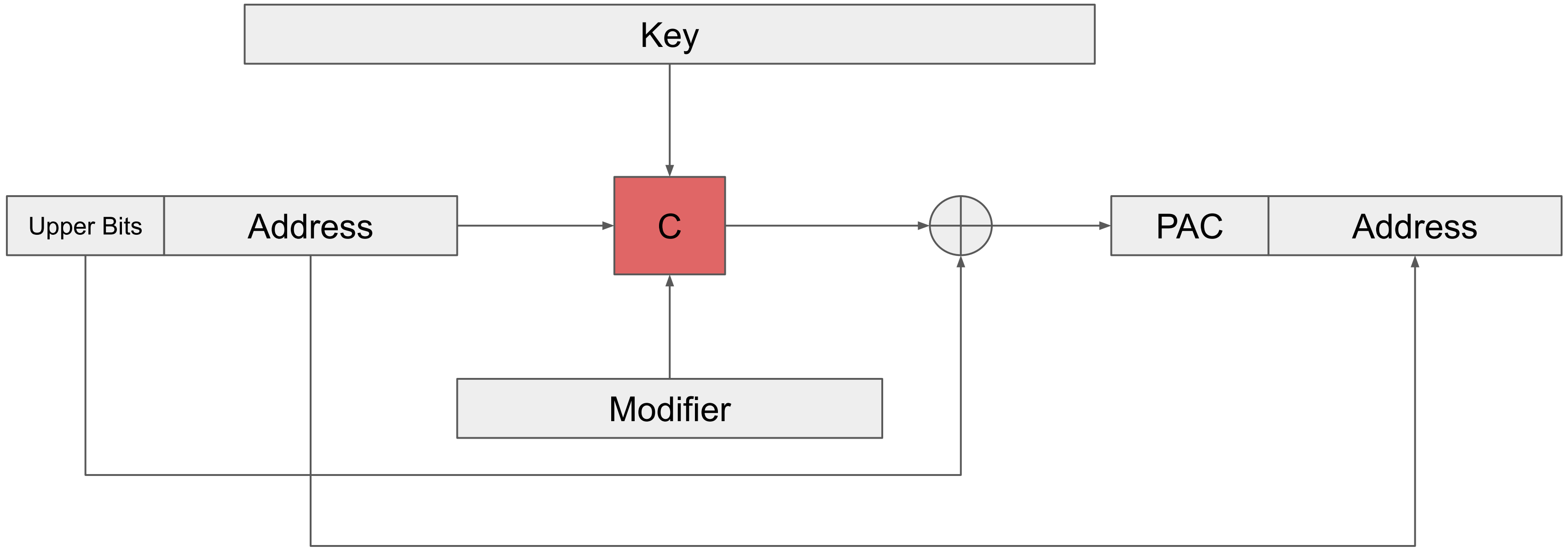
- Use modifier to **restrict** a signed pointer to a specific **context** 💡

C(P1, K1, M1) = PAC1

C(P1,K1, M2) = PAC2

- Modifier not secret
- Possible values: stack pointer, register, zero

PAuth: Cipher



PAuth: The QARMA Cipher

Reference cipher developed at Qualcomm’s Product Security office in Munich and was named after **Q**ualcomm **ARM** **A**uthenticator.

Cipher:

- Lightweight Tweakable Block Cipher (TBC)
- Purpose-designed with low latency, power consumption, silicon real-estate in mind
- 64-bit and 128-bit block-size / 128-bit and 256-bit key-size

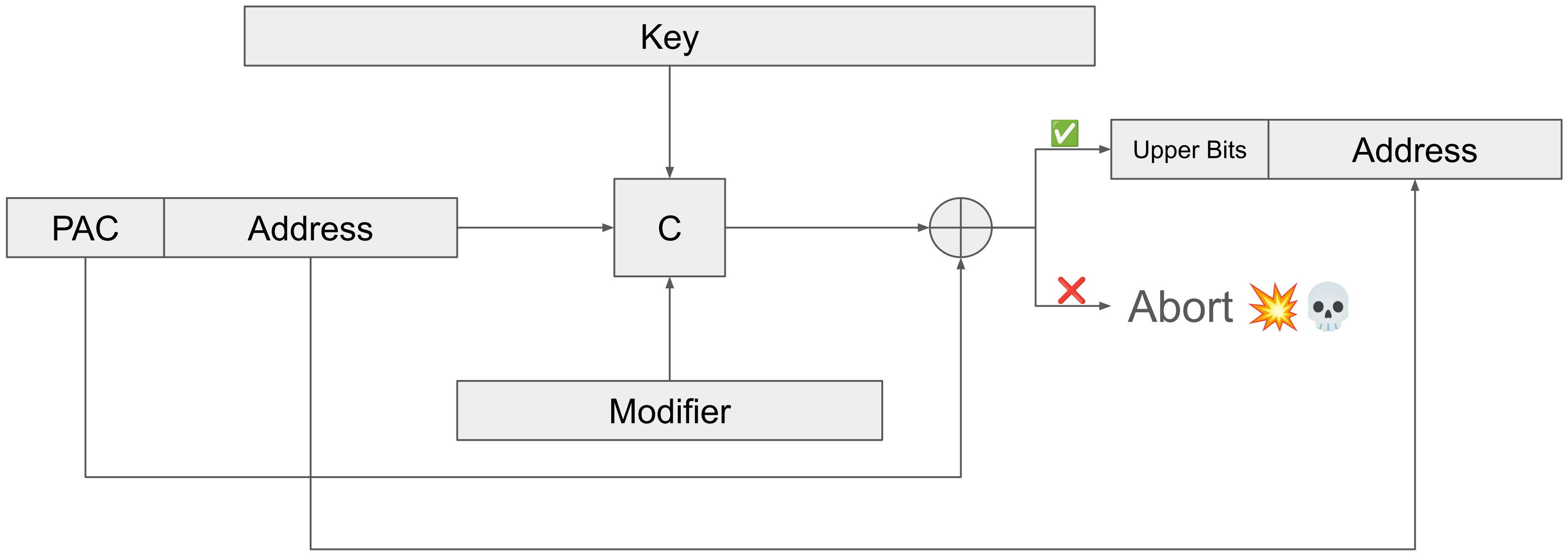
In PAuth:

- 64-bit blocks and 128-bit keys
- Available variants are QARMA-5 and QARMA-3

For more info see:

- **Paper:** https://eprint.iacr.org/2016/444.pdf
- **Submission to NIST:** https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/qameleon-spec.pdf

PAuth Operation: Authenticate



PAuth: Implementation

New Instructions: Basic Set (32 instr.)

Instruction Class	Purpose
PAC[key][modifier]	Sign pointer with key and modifier
AUT[key][modifier]	Authenticate pointer with key and modifier
XPAC[I,D,LRI]	Strip PAC from pointer into register

⚠️ Implemented as **HINT** instructions, execute as **NOP** on pre-Armv8.3 CPUs.

PAuth: Implementation

New Instructions: Combined set (14 instr)

Instruction Class	**Purpose**	**Key Type**
RETA[A,B]	Authenticated return	Instruction Key A, B
BRA[A,B][Z]	Authenticated branch	Instruction Key A, B
BLRA[A,B]Z	Authenticated branch-link	Instruction Key A, B
ERETA[A,B]	Authenticated exception return	Instruction Key A, B
LDRA[A,B]	Authenticated load	Data Key A, B

⚠️ Implemented as **FUSED** instructions, cause **FAULT** on pre-Armv8.3 CPUs.

PAuth: Implementation

```
0000000000400564 <main>:
  400564:     d503233f      paciasp
  400568:     a9bf7bfd      stp    x29, x30, [sp, #-16]!
  40056c:     910003fd      mov    x29, sp
  400570:     90000000      adrp   x0, 400000 <_init-0x3e8>
  400574:     9118c000      add    x0, x0, #0x630
  400578:     97ffffb6      bl     400450 <puts@plt>
  40057c:     52800000      mov    w0, #0x0
  400580:     a8c17bfd      ldp    x29, x30, [sp], #16
  400584:     d50323bf      autiasp
  400588:     d65f03c0      ret
```

```
0000000000400564 <main>:
  400564:     d503233f      paciasp
  400568:     a9bf7bfd      stp    x29, x30, [sp, #-16]!
  40056c:     910003fd      mov    x29, sp
  400570:     90000000      adrp   x0, 400000 <_init-0x3e8>
  400574:     9118a000      add    x0, x0, #0x628
  400578:     97ffffb6      bl     400450 <puts@plt>
  40057c:     52800000      mov    w0, #0x0
  400580:     a8c17bfd      ldp    x29, x30, [sp], #16
  400584:     d65f0bff      retaa
```

Branch Target Identification (BTI)

Branch Target Identification (BTI)

Armv8.5-A introduces **FEAT_BTI** as a **MANDATORY** extension.

Mitigates attacks on *indirect jumps* (BLR, BR, RET).

Branch Target Identification (BTI)

```
4010d1e4:   eb00007f        cmp     x3, x0
4010d1e8:   54ffffc1        b.ne    4010d1e0 <strerror_r+0xd0>  // b.any
4010d1ec:   17ffffdf        b       4010d168 <strerror_r+0x58>
4010d1f0:   d503249f        bti     j
4010d1f4:   f0000081        adrp    x1, 40120000 <days_per_mon+0x15c8>
4010d1f8:   912e8021        add     x1, x1, #0xba0
4010d1fc:   17ffffeb        b       4010d1a8 <strerror_r+0x98>
4010d200:   d503249f        bti     j
4010d204:   f0000081        adrp    x1, 40120000 <days_per_mon+0x15c8>
4010d208:   912de021        add     x1, x1, #0xb78
4010d20c:   17ffffe7        b       4010d1a8 <strerror_r+0x98>
4010d210:   d503249f        bti     j
4010d214:   f0000081        adrp    x1, 40120000 <days_per_mon+0x15c8>
4010d218:   912d8021        add     x1, x1, #0xb60
4010d21c:   17ffffe3        b       4010d1a8 <strerror_r+0x98>
4010d220:   d503249f        bti     j
4010d224:   f0000081        adrp    x1, 40120000 <days_per_mon+0x15c8>
4010d228:   912d4021        add     x1, x1, #0xb50
4010d22c:   17ffffdf        b       4010d1a8 <strerror_r+0x98>
4010d230:   d503249f        bti     j
...
```

BTI: Implementation

New Instruction: `BTI {type}`

The compiler generates BTI instructions with {target} set to the expected branch type.

- **c:** indirect calls (BLR Xn)
- **j:** indirect jumps (BR Xn)
- **jc:** indirect jumps or indirect calls

⚠️ The BTI instruction is implemented on the **HINT** space, will execute as **NOP** on pre-Armv8.5-A CPUs.

BTI: Implementation

Page Tables: New Guarded Page (GP) attribute

- Pages marked as *guarded* are protected with BTI
- Pages marked as *unguarded* don’t require a BTI instruction on branch targets

Allows backwards compatibility with code that has not been compiled with BTI.

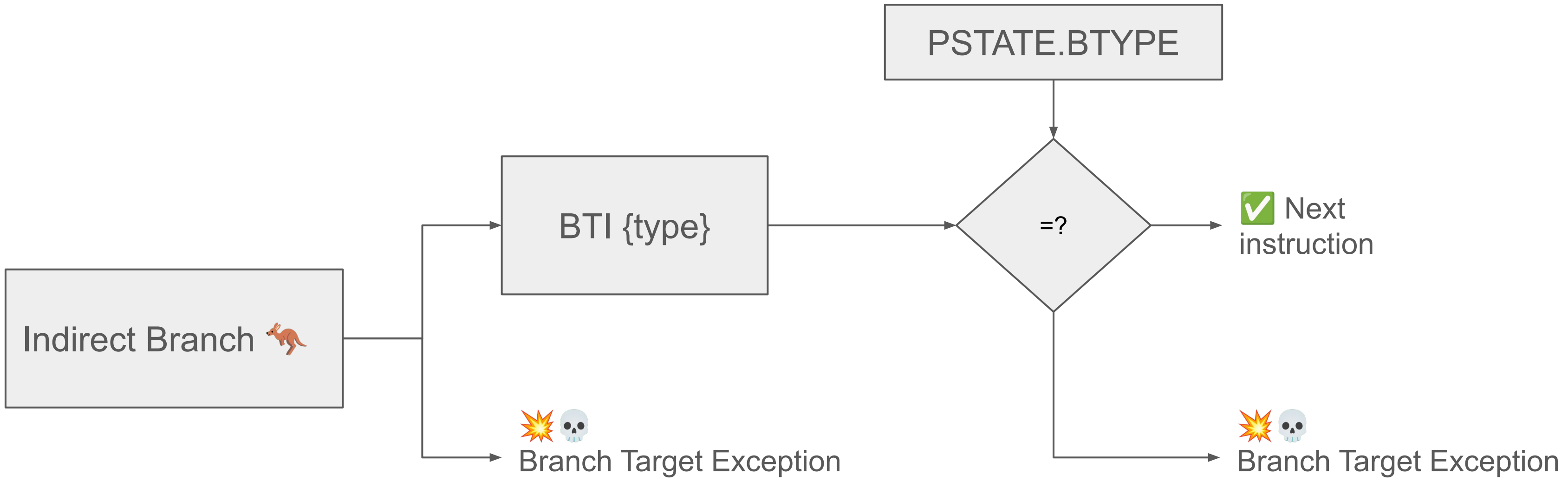
BTI: Implementation

New Process State (PSTATE) Bits: BTYPE

Encodes branch type. Updated on every instruction.

BTYPE	**Instructions**	**Memory Type**	**Register**	**Notes**
0b00	Non-branch	Any	Any	-
0b01	Any	Non-Guarded	Any	-
0b01	BR, BRAA, BRAAZ, BRAB, BRABZ	Guarded	X16, x17	IP0, IP1 (see APCS) Non-function calls. Veneers, Jump Tables.
0b10	BLR, BLRAA, BLRAAZ, BLRAB, BLRABZ	Any	Any	Function calls.
0b11	BR, BRAA, BRAAZ, BRAB, BRABZ	Guarded	Any except x16, x17	Non-function calls. Case statements.

BTI: Implementation



Ecosystem Support

Compiler Support (gcc / clang)

Common parameter for gcc & clang:

```
    -mbranch-protection=<type>
```

<type> values:

- **none:** Disables all branch protections
- **pac-ret:** Enables PAuth for function returns on non-leaf functions
    - **+leaf** addendum to sign return address of leaf functions
    - **+bkey** addendum to use Key B
- **bti:** Enables BTI
- **standard:** Enables all protections

Compiler Support (gcc / clang)

How good is it?

- PAuth only protects the return address (RoP)
- No C++ vtables protection
- No authenticated branching
- No authenticated loads
- Relies on BTI to protect against JoP

Apple’s fork of LLVM is more interesting: https://github.com/apple/llvm-project

Linux / Android Support

Userland: CONFIG_ARM64_PTR_AUTH / CONFIG_ARM64_BTI

- Per-process keys (IA, IB ,DA, DB, G). Kernel updates on context-switch.
- Process keys shared between threads.
- Inherit keys upon fork(), re-init keys upon execve().

Kernel: CONFIG_ARM64_PTR_AUTH_KERNEL / CONFIG_ARM64_BTI_KERNEL

- Only IA
- Key switched upon kernel entry / exit.

Again, Apple more interesting.

Hardware Support

SoC	PAuth	BTI
Amazon Graviton 3	Y	N
Amazon Graviton 4	Y	Y
Apple M1	Y	N
Apple M2	Y	Y
Apple M3	Y	Y
Google Tensor G3	Y	Y
Mediatek Dimensity 9000	Y	Y
Qualcomm Snapdragon 8 Gen 1 (SM8450)	Y	Y

SoC	PAuth	BTI
Qualcomm Snapdragon 8 Gen 2 (SM8550)	Y	Y
Qualcomm Snapdragon 8 Gen 3 (SM8650)	Y	Y
Qualcomm Snapdragon 8cx Gen3	Y	N
Samsung Exynos 2200	Y	Y
Samsung Exynos 2400	Y	Y

† https://gpages.juszkiewicz.com.pl/arm-socs-table/arm-socs.html

Evaluation

Evaluation: PAuth Security

Probability to guess PAC:

$$P(☠️) = 1 / 2^{PAC\_bits}$$

Virtual Address Size	**TBI disabled** **MTE disabled**	**TBI disabled** **MTE enabled**	**TBI enabled** **MTE don't care**
36-bits	PAC bits: 27 P(☠️) = 1 / 134,217,728	PAC bits: 23 P(☠️) = 1 / 8,388,608	PAC bits: 19 P(☠️) = 1 / 524,288
39-bits	PAC bits: 24 P(☠️) = 1 / 16,777,216	PAC bits: 20 P(☠️) = 1 / 1,048,576	PAC bits: 16 P(☠️) = 1 / 65,536
42-bits	PAC bits: 21 P(☠️) = 1 / 2,097,152	PAC bits: 17 P(☠️) = 1 / 131,072	PAC bits: 13 P(☠️) = 1 / 8,192
48-bits	PAC bits: 15 P(☠️) = 1 / 32,768	PAC bits: 11 P(☠️)= 1 / 2,048	PAC bits: 7 P(☠️)= 1 / 128
52-bits	PAC bits: 11 P(☠️) = 1 / 2,048	PAC bits: 7 P(☠️) = 1 / 128	PAC bits: 3 P(☠️) = 1 / 8

Evaluation: BTI Security

Given a object with **M** BTI instructions, the probability to find a chain of **N** gadgets is:

$$\mathbf{P(☠️) = M^N}$$

For **M** ≃ 1%, P(☠️) approaches zero as **N** grows.

† https://sipearl.com/en/sipearl-white-paper-anti-malware-active-protection-on-arm64-systems

Evaluation: PAuth Effectiveness

Arm’s white-paper on PAuth / BTI [2]:

- *ROP gadgets reduced by ~77x when PAuth is used without BTI.*

Based on glibc (unknown version) & ROPGadget (unknown gadget depth)

Arm’s presentation on Chromium [3]:

- *ROP gadgets reduced by ~59x.*
- *Report combined results for ROP/JOP with Pauth+BTI.*

Based on glibc (Ubuntu 14.04) and ROPGadget (gadget depth ≤ 10)

Evaluation: PAuth Effectiveness

My numbers (Nginx 1.15 / Unikraft 0.17.0 / GCC-13)

None:

⌨️
```
ROPgadget.py --binary --nojop build/nginx_qemu-arm64
```

➡️ 41,386 unique gadgets

PAuth:

💡RET → ROP gadget ⇒ RETAA → ! ROP gadget

⌨️
```
ROPgadget.py --binary --nojop build/nginx_qemu-arm64 | grep -v retaa
```

➡️ 3,285 unique gadgets

Result: 92% improvement

Evaluation: PAuth Effectiveness

How about the remaining 8%?

```
00000000401fb4d0 <timespec_get>:
    401fb4d0:   7100043f     cmp  w1, #0x1
    401fb4d4:   54000060     b.eq 401fb4e0 <timespec_get+0x10>  // b.none
    401fb4d8:   52800000     mov  w0, #0x0                      // #0
    401fb4dc:   d65f03c0     ret
    401fb4e0:   d503233f     paciasp
    401fb4e4:   a9bf7bfd     stp  x29, x30, [sp, #-16]!
    401fb4e8:   aa0003e1     mov  x1, x0
    401fb4ec:   910003fd     mov  x29, sp
    401fb4f0:   52800000     mov  w0, #0x0                      // #0
    401fb4f4:   97fff7f3     bl   401f94c0 <__clock_gettime>
    401fb4f8:   2a2003e0     mvn  w0, w0
    401fb4fc:   a8c17bfd     ldp  x29, x30, [sp], #16
    401fb500:   531f7c00     lsr  w0, w0, #31
    401fb504:   d65f0bff     retaa
```

Evaluation: BTI Effectiveness

My numbers (Nginx 1.15 / Unikraft 0.17.0 / GCC-13 / ROPGadget)

None:

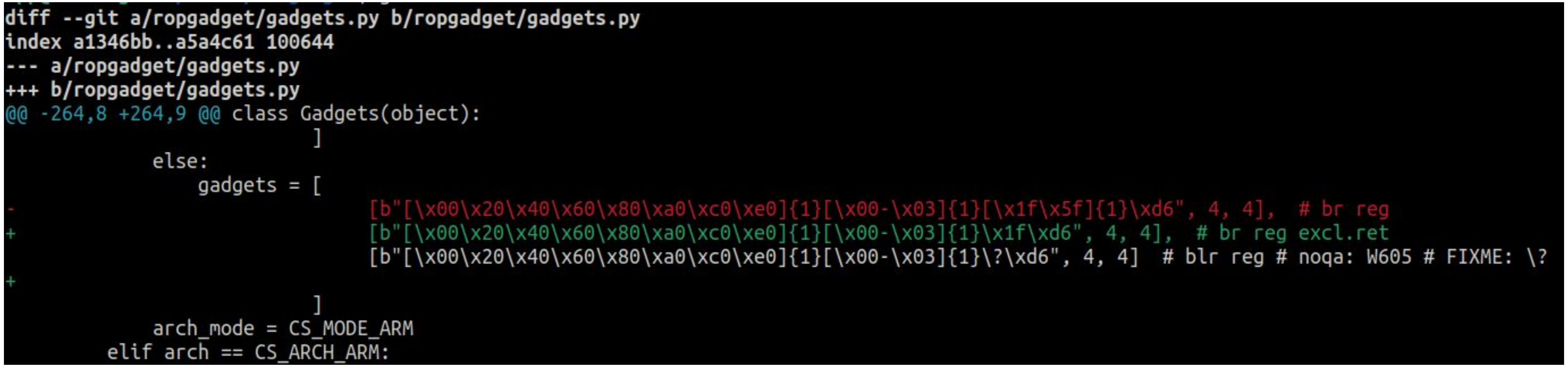
💡 JOP Gadget = #depth instructions to next indirect branch

⚠️ ROPGadget uses BR, BLR, RET for JOP gadgets ⇒ patch to exclude RET

⌨️ ROPgadget.py --binary --nojop build/nginx_qemu-arm64

➡️ 7,944 unique gadgets

Evaluation: BTI Effectiveness



```
diff --git a/ropgadget/gadgets.py b/ropgadget/gadgets.py
index a1346bb..a5a4c61 100644
--- a/ropgadget/gadgets.py
+++ b/ropgadget/gadgets.py
@@ -264,8 +264,9 @@ class Gadgets(object):
                            ]
             else:
                 gadgets = [
-                               [b"[\x00\x20\x40\x60\x80\xa0\xc0\xe0]{1}[\x00-\x03]{1}[\x1f\x5f]{1}\xd6", 4, 4],  # br reg
+                               [b"[\x00\x20\x40\x60\x80\xa0\xc0\xe0]{1}[\x00-\x03]{1}\x1f\xd6", 4, 4],  # br reg excl.ret
                                [b"[\x00\x20\x40\x60\x80\xa0\xc0\xe0]{1}[\x00-\x03]{1}\?\xd6", 4, 4]  # blr reg # noqa: W605 # FIXME: \?
+
                            ]
             arch_mode = CS_MODE_ARM
         elif arch == CS_ARCH_ARM:
```

Evaluation: BTI Effectiveness

BTI:

- 💡 JOP Gadget = BTI + #depth instructions to next indirect branch
- ⚠️ ROPGadget doesn't know about BTI ⇒ Filter out gadgets with BTI
- ⌨️
```
ROPgadget.py --norop --binary build/nginx_qemu-arm64 | \
sed 's/.* : //' | grep '^bti' | sort | uniq | wc -l
```
- ➡️ 48† unique gadgets

Result: 99.4% improvement

† In practice even less as don't take into account the branch type

Evaluation: PAuth Performance

Instruction Overhead:

+2 instructions per function (PACIASP/RETAA)

Instruction Latency: (Neoverse N2)

- **PACIASP:** 5 cycles
- **RETAA:** 5 cycles

➡️ +10 cycles per function

Evaluation: BTI Performance

Instruction Overheard:

+1 instruction per indirect branch (BTI)

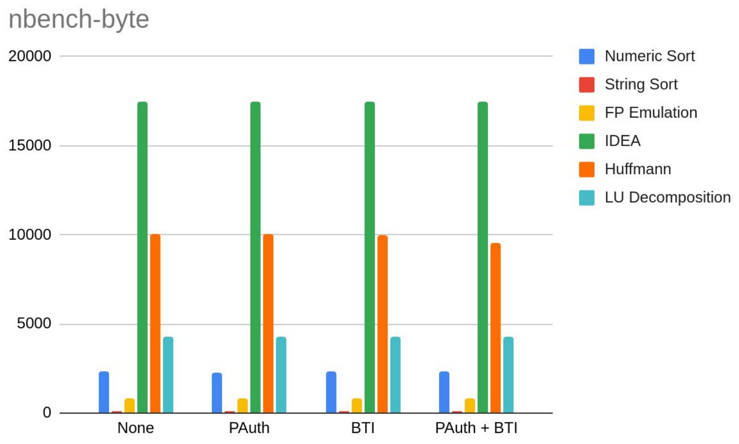
Instruction Latency: (Neoverse N2)

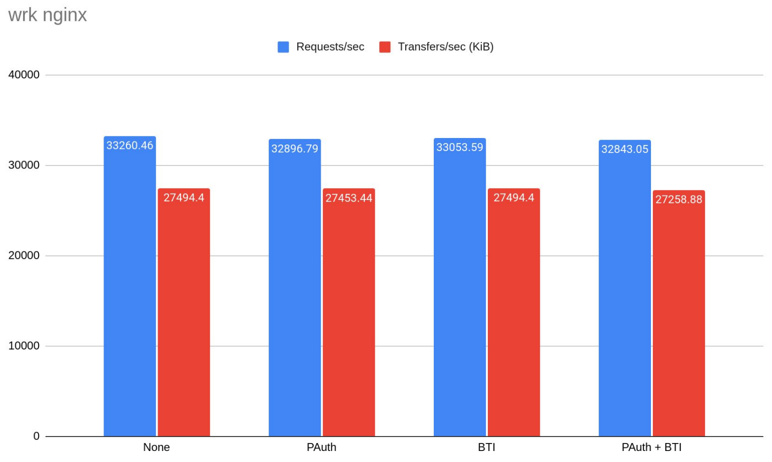
- **BTI:** undocumented (should be 1 cycle)

➡️ +1 cycle per indirect branch

Evaluation: Performance

My numbers:



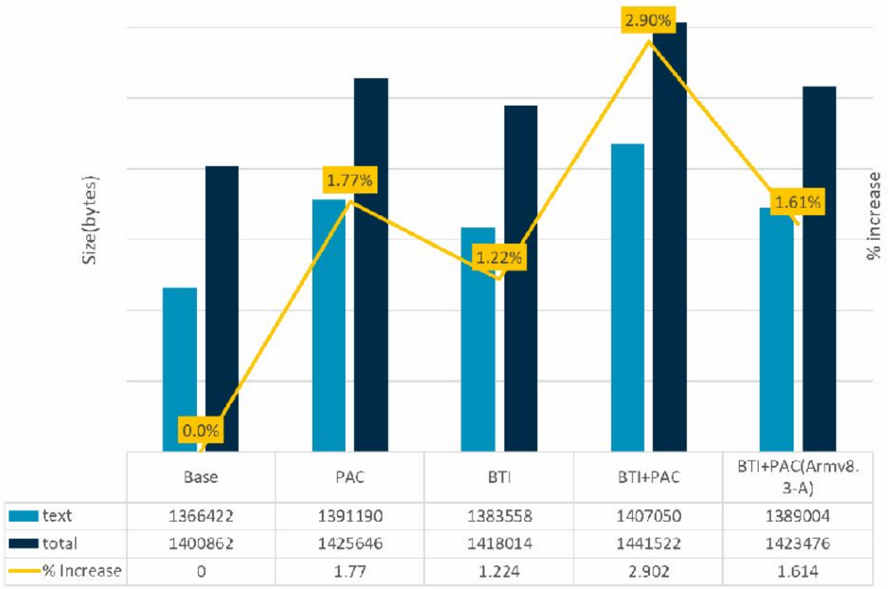


Unikraft 0.17.0 / GCC-13 / QEMU 9.x with HVF acceleration on Apple M2 (Mac Mini)

Evaluation: Code Size

Arm's measurements [2]: glibc (unknown version)

- **PAuth (basic set):** +1.77%
- **BTI (basic set):** +1.22%
- **PAuth + BTI (basic set):** +2.90%
- **Pauth + BTI (comb. set):** +1.61%



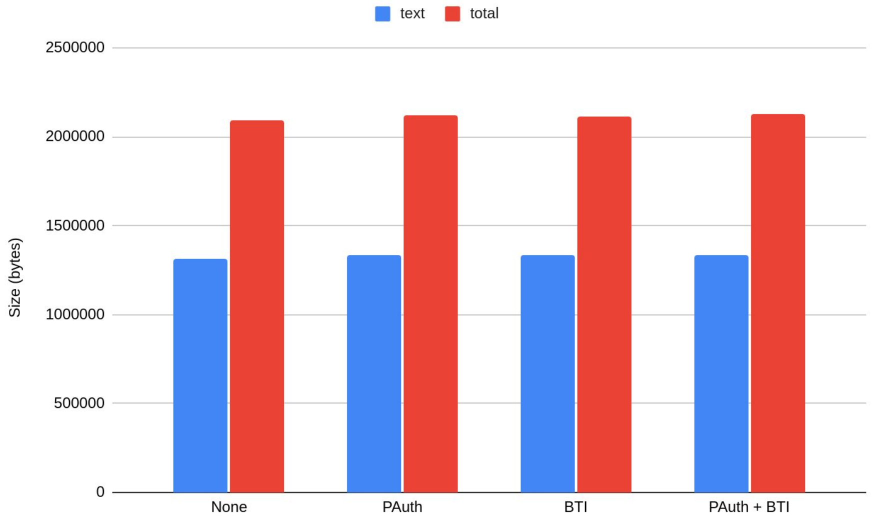
	Base	PAC	BTI	BTI+PAC	BTI+PAC(Armv8.3-A)
text	1366422	1391190	1383558	1407050	1389004
total	1400862	1425646	1418014	1441522	1423476
% Increase	0	1.77	1.224	2.902	1.614

Evaluation: Code Size

My numbers:

nginx 1.25 / Unikraft 0.17.0 / GCC-13

- **PAuth (comb. set):** +1.52%
- **BTI (comb. set):** +1.26%
- **Pauth + BTI (comb. set):** +1.77%



	None	**PAuth**	**BTI**	**PAuth + BTI**
text	1311264	1333316	1331652	1337188
total	2094704	2123376	2115184	2127472
increase %	0%	1.52%	1.26%	1.77%

In Practice

Surely CFI is enforced by now! 👮🏻 🏆

- 👹 Examining Pointer Authentication on the iPhone XS
- 👹 Splitting atoms in XNU
- 👹 HackPac: Hacking Pointer Authentication in iOS User Space
- 👹 iOS Kernel PAC, One Year Later
- 👹 Fugu 14 jailbreak
- 👹 Everything has Changed in iOS 14, but Jailbreak is Eternal
- 👹 Fugu 15 jailbreak
- 👹 PACMAN: Attacking ARM Pointer Authentication with Speculative Execution
- 👹 Apple PAC, Four Years Later
- 👹 Demystifying Pointer Authentication on Apple M1

Final Thoughts

- PAuth & BTI are performant and effective.
- Great value for money: Implementation effort vs improvement.
- Pragmatic: No code rewrite in applications, minimal implementation in kernel.
- A leap forward compared to software mitigations.
- Eventually expected to be available in all Arm-based SoC.
- Although not impossible to bypass, still make CFI attacks significantly harder.
- More hardware-assisted mitigations on the way:
  - Memory Tagging (MTE) against memory safety violations.
  - Keep an eye on CHERI.

QUESTIONS