SIPEARL

Control Flow Integrity

Anti-Malware active protection on Arm64 Systems

A SiPearl White Paper

Executive summary

While major IT infrastructures are confronted with repeated ransomware and other cyberattacks, the security agencies call for help from the industry to stop the propagation of malware.

The cybersecurity chain of tools is now well established to detect and eradicate malware. From perimetric security on the network to behavioral security inside each system, each component provides its contribution to the security of large IT systems.

One loophole remains, however. When malware hits a system, after bypassing all protections ahead, the processor gently executes it, implicitly allowing the malicious payload to infect the system.

At SiPearl, we have one preoccupation : How can our processors fill that loophole? How can we actively contribute to anti-malware protection?

Following a careful analysis of the mechanisms through which malware finds its way into the system, this white paper describes how the Call Flow Integrity (CFI) techniques can prevent such intrusions.

The Arm64 CPU cores at the heart of the SiPearl processors offer unique CFI capabilities on the market. Two complementary features, Pointer Authentication Code (PAC) and Branch Target Identification (BTI), build a shield against the most dangerous forms of malicious code injection, those which exploit low-level software bugs and vulnerabilities.

This white paper describes in detail PAC and BTI, how they are defined by Arm, how to leverage their enormous potential, and how the current software implementations use them.

From an operating system perspective, the usage of PAC and BTI is described in Linux, macOS and Windows. The gcc and clang compilers are also examined, including the clang fork of Apple for macOS.

Lessons are learnt from this comparison. The most advanced and complete usage of PAC and BTI is observed in macOS and the corresponding clang fork. The way it works there demonstrates the power of the control flow integrity techniques in the Arm64 cores.

Based on this comparison, what remains to be done in Linux, gcc and mainstream clang becomes clear.

For the open-source communities who would volunteer to implement these improvements, or merge existing ones, or simply for the curious reader, two thirds of this white paper is a deep dive into the technical details of PAC and BTI.

Initial release: September 2023

Contents

1. Cybersecurity threats

1.1. Defining the security target for SiPearl processors

The first generation of SiPearl processors is named Rhea. It targets the High-Performance Computing (HPC) market. It is specially designed to empower supercomputers for intensive computing.

The CPU cores in the Rhea processor are Arm Neoverse V1, which includes 256-bit vector computing units. In addition to traditional DDR memory, the Rhea processor embeds controllers for High-Bandwidth Memory (HBM), a high-end form of memory which is typically found in supercomputers.

The next generation of SiPearl processors will be twofold. First, there will be a successor to Rhea for the HPC market, with updated CPU cores and various improvements. Second, there will be a processor for the cloud and data center market.

This roadmap defines the security target for SiPearl processors : high-end HPC, cloud, data center, server systems and their environment.

This is a traditional IT-managed environment where the security is controlled by cybersecurity teams, using well-defined tools, in physically controlled environments.

Let's remark that, by contrast, embedded systems (IoT, mobile phones, etc.) are mobile, physically exposed to attacks. Their threat model is very different and includes ranges of physical attacks on the hardware components: invasive attacks (glitch, laser) or side channel attacks (power, temperature). This well-known threat model in the hardware community does not apply to HPC, cloud or data centers.

A careful definition and analysis of the threat model for the SiPearl processors and the corresponding security priorities are initial requirements to define the right security solutions.

1.2. Prioritizing the threats

What are the topmost important threats in IT systems?

How can a processor contribute to mitigate these threats?

These are the two important questions a modem processor for IT systems should answer.

It is the role of the national and international security agencies to identify the top-priority threats. These agencies operate in the field. They respond to actual threats. They fix real problems. They have learnt to identify the malicious actors, their motivations, their mode of operation.

What do they say?

Jen Easterly, head of CISA, USA [24]:

> *We cannot have the same sort of attacks on hospitals and school districts that we've been seeing for years. We have to create a sustainable approach to cyber safety.*

Guillaume Poupard, former head of ANSSI, France [25]:

> *Hospitals, institutions, corporations, and all sectors are impacted by hackers. The number of attacks has multiplied by ten in three years. And this is probably a small part of reality.*

The security agencies call for action. They request the cybersecurity community to fight criminal activities, to stop cyber-attacks. Cyber-criminal activities are real, current, frequent. They hurt our countries, the institutions, the national corporations, and society in general.

The most important vectors of cyber-attacks have been known for years: all sorts of malware propagate over networks, through direct attacks, phishing, social engineering, tricking computers or users to download or accept incoming malicious data, turning them into malware exploiting software vulnerabilities.

How can a processor help in this cybersecurity fight?

1.3. Role of the processor in the cybersecurity chain

1.3.1. Anatomy of a malware

In cybersecurity, it is common to describe malware in terms of *vector* and *payload*.

This terminology is inspired by the military, from the structure of a missile.

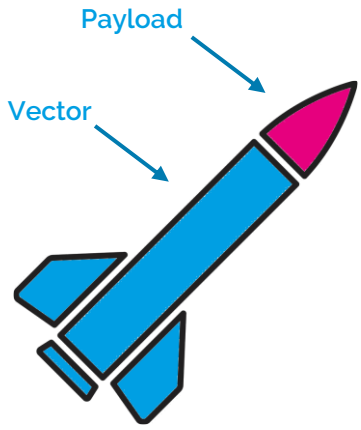

Figure 1: Vector and payload

Table 1: Vector and payload, a comparison

Vector	Military	Cybersecurity
Type	Rocket	Hacking software
Purpose	Propulsion	Exploit software vulnerabilities, penetrate systems
Skills	Astronautics	Hacking

Payload	Military	Cybersecurity
Type	Bomb	Application software
Purpose	Destroy	Steal secrets, credentials, encrypt / lock data file, etc.
Skills	Chemistry	Software development

In both cases, the military and cybersecurity, the vector is a vehicle which transports the destructive payload into the territory of the target.

In the fight against an attack, it is equally useful to counter the vector and the payload. However, different components in the system more easily address one or the other.

Let's review now where the processor can be useful.

1.3.2. Tracking malware in the cybersecurity chain for IT systems

In large IT systems, most of the cybersecurity chain is made of software. When the term *hardware* is used, it often designates a *network security appliance*, a closed system with security software running on it. There is no or little *hardware security*, as the term is used in embedded systems (mitigate physical attacks).

The way malware is tracked all along the chain is summarized in Figure 2 below.

- Perimetric security is the first line of defense. Intrusion Prevention Systems (IPS) and firewalls integrate malware detection systems and try to prevent their intrusion in the network.
- The internal network security is made of Intrusion Detection Systems (IDS). A Security Information and Event Management system (SIEM) centralizes the events and reports alerts, such as detected malware.
- Inside each server or desktop system, behavioral security includes integrity checks on the operating system and applications, applies heuristics on the running system to detect suspect activities, possibly with the help of Artificial Intelligence (AI) techniques.
- At the end of the chain, application security includes antivirus products which permanently inspect the file systems to detect malware.

Note: Modern forms of IT cybersecurity, such as Zero Trust (ZT) architecture, focus on authentication. Instead of considering the network as a trusted area, authentication is centralized, and identification is enforced virtually everywhere. However, ZT does not address malware detection. In IT infrastructures with ZT authentication, the fight against malware infection remains the responsibility of detection tools.

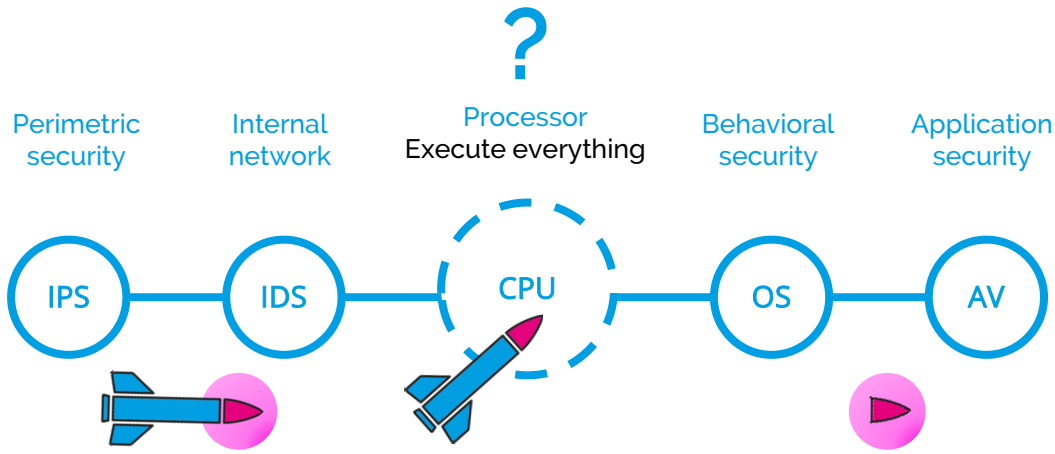

Figure 2: The current malware detection chain

Two interesting facts emerge from this description:

- Most detection tools focus on the malware payload. The viral charges are detected, either statically using sets of signatures, or dynamically based on their suspect behavior. No or few tools can detect or block the malware vectors, the penetration subset of the malware.
- Inside server systems or desktop systems, the processor has no security added value so far. A processor just executes instructions, equally, sequentially, one after the other, without checking, without wondering if it executes the operating system, an application, or some malware.

Clearly, the processor can do better.

1.3.3. How the processor should contribute to the cybersecurity chain

The processor is the component which is directly hit by malware vectors.

The purpose of the malware vector is to penetrate a system. It starts with a legitimate application or network stack with manipulates the incoming malware as simple data. Then, the malware exploits security vulnerabilities in the legitimate software to force the execution of the vector, then the payload.

What is a security vulnerability? Most of the time, this is a software bug that a hacker turns into a vulnerability, a way of penetrating the system.

We can classify these software bugs in two categories:

- High-level software bugs: design or configuration errors which open the doors of the system.
- Low-level software bugs: input validation failure or buffer overflow which are turned into code injection by hackers.

An example of design error is Log4j [27], a logging framework which, by design, allows the download and execution of arbitrary code. Examples of configuration errors are weak passwords, default passwords, opened network ports.

These high-level errors are mostly due to a lack of methodology. Security awareness is continuously increasing in the software industry, and we observe a slow but constant decrease in this type of threat.

Low-level bugs, on the other hand, are much more difficult to track and eradicate. Most of them are small programming errors, with devastating consequences. Even though more robust programming languages and tools exist, programmers are humans and low-level software bugs will likely never disappear.

This is where the processor can help.

This is the added value of the processor in the cybersecurity chain.

The processor is the only component in the system which is in direct contact with malware vectors. The processor is in a unique position to detect malware vectors in action and block them before penetration. Thus, the processor fills the hole in the security chain.

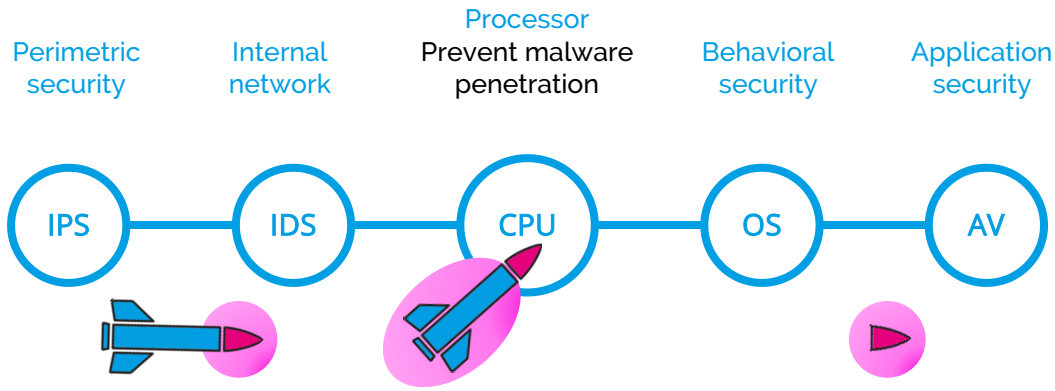

Figure 3: Active role of the processor in the malware detection chain

Detecting malware vectors is obviously not sufficient to prevent all malware injections. Typically, the processor cannot detect misconfigured systems which allow arbitrary code injection. In the security chain, no component can guarantee the security of the system alone. However, each component has a role to play. The security of the system is guaranteed by the combination of these individual roles. The purpose of this paper is to define the role of the processor in the cybersecurity chain.

Let's now review how malware takes advantage of these low-level software bugs to inject malicious code. Then, in the next chapter, we will see how the processor can block these attacks using defensive security techniques.

1.3.4. Malicious code injection techniques

Code injection is the art of turning malicious data into malware code execution.

The most common root cause is a software bug (input validation failure, insufficient memory management) causing a buffer overflow in a legitimate – but buggy – application. The buffer overflow overwrites memory containing pointers to code. When an indirect branch is taken using these code pointers, unexpected code is executed.

The art of hacking consists in subverting existing branch instructions to jump into code which is controlled by the hacker.

Buffer overflow exploitation techniques have been abundantly documented, although essentially on Intel CPU architecture. See examples in [29], [30], [31], [32].

Historical methods: direct code injection on stack

The original code injection technique is as old as the Morris worm in 1988 [26]. This typical software bug is illustrated on Figure 4 below.

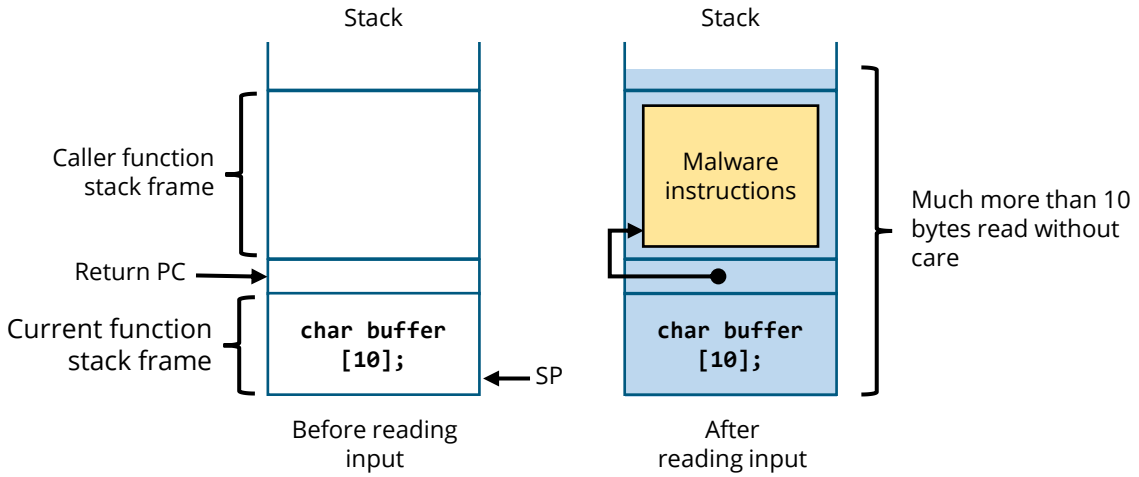

Figure 4: Older direct code injection techniques on stack overflow

Some function in the application reads a variable-length data structure from an input source which can be controlled by the attacker (a document file, a network connection). By specification of the protocol, this data structure is limited to 10 bytes for instance. The application consequently allocates a 10-byte local buffer (on stack). Because of a bug, the application fails to validate its input. If the data announces a much larger size, say 100 bytes, the application blindly reads those 100 bytes. This causes a buffer overflow, overwriting the memory beyond the input buffer.

On most CPU architectures, the stack expands to descending addresses. Overflowing a local buffer on stack means overwriting the return address to the caller. After the buffer overflow, when the function which performed the input operation returns, the processor pops the return address from stack and jumps to the corresponding code address. If that address was trashed with garbage, the program would likely crash.

However, in the case of a carefully crafted attack, the hacker includes the malicious code in the overflown input data and places the address of this code in the middle, where the return address is expected. Thus,

on return, the function simply jumps into the hacker's code. This is how we turn passive input data into executed code.

This was a long time ago.

Since then, most operating systems turn the stack into "non-executable mode" in the page tables of the virtual memory system. When the historical direct attack is attempted, the application crashes when the processor attempts to execute an instruction from the stack.

Modern techniques: return-oriented programming (ROP)

As hackers are quite good at mitigating countermeasures, they quickly found a workaround.

In the early 2010's, they invented the Return-Oriented Programming (ROP) techniques. Since data areas were all protected against execution, it was no longer possible to directly inject code as data. However, in the meantime, the applications became so complex that the virtual address space of any application contains megabytes of legitimate code pages, from the application, from the libraries it uses, from the other libraries which are used by the libraries, etc.

It may seem weird, but it is easy to cut a potential malware code into small blocks of instructions which can be found in memory, in the megabytes of legitimate code pages. The condition is that each small block is terminated by a "return" instruction or equivalent.

Each return-terminated piece of legitimate code is called a *gadget*. Starting from a piece of malware code, automated tools can slice it into gadgets, searching into the binaries of the application and all shared libraries which are recursively activated. Such tools are named *gadget finders*.

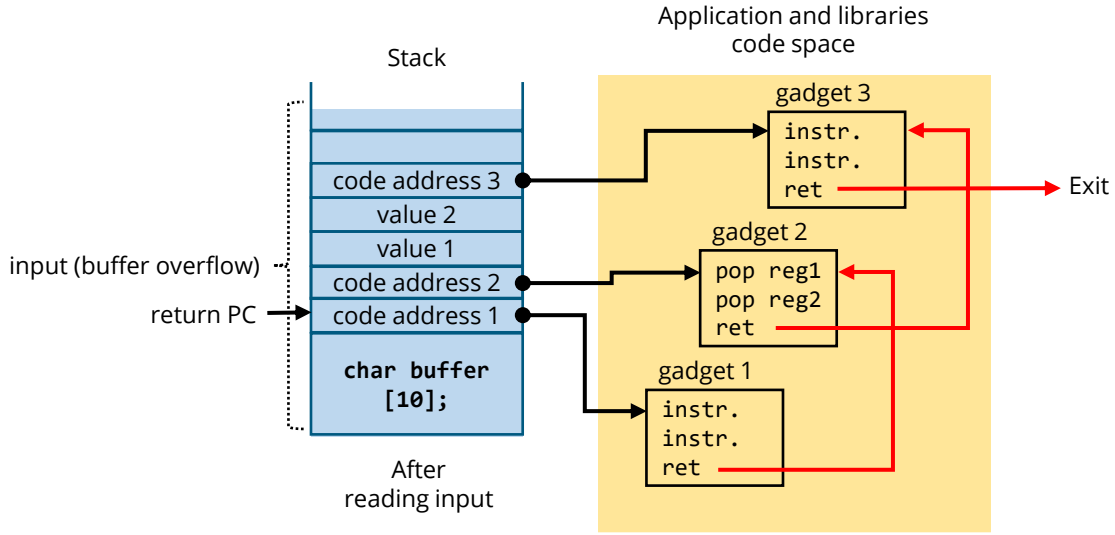

Figure 5: Simplified 3-gadget ROP chain attack

Once the list of addresses of these gadgets is built, just inject it in reverse order in the buffer overflow attack, starting at the address of the first return address. When the buggy function which causes the buffer overflow returns, the processor jumps to the first gadget in the list. Since this gadget terminates with a return instruction, the processor naturally pops the address of the second gadget and jumps to it. And so on…

Of course, depending on the CPU architecture and its ABI, the exact layout of the buffer overflow area may be slightly more complex than a simple list of addresses. But the principle remains the same.

This is illustrated in Figure 5 above.

Again, this technique may seem weird, but it works. If you are skeptical, just google for “ROP gadget finder”.

To counter these techniques, most operating systems have introduced the concept of Address Space Layout Randomization (ASLR). Each time a process is created, the main executable and all libraries are mapped in virtual memory at random addresses. Thus, it is more complicated to predict the virtual address of an identified gadget. However, there are a few constraints. Exactly all compilation units in the application and the libraries must be compiled with position-independent code. This is not always possible. Even if the amount of legitimate code at predictable virtual addresses is smaller with ASLR than without it, it does not completely prevent ROP attacks.

In addition to ROP attacks which are based on buffer overflow on stack to exploit return instruction, there is a variant named Jump-Oriented Programming (JOP). It exploits buffer overflow elsewhere, typically on the heap. The idea is to overwrite code pointers from code dispatch tables of C++ vtables and then wait for the application to call the corresponding code.

Let’s now review how the defensive security features of the Arm64 architecture can help prevent these attacks.

2. Arm64 defensive security overview

The security architecture of a system can be divided into *security by design* feature and *defensive security* features. The two sides are equally important to guarantee the security of the system.

Security by design is a theoretical top-down approach which defines the security architecture, the roles of all actors in the system, the security boundaries, and the isolation mechanisms. In the Arm architecture, security by design mechanisms include TrustZone [6] for embedded systems and Confidential Compute Architecture (CCA) [7] for server systems.

Defensive security, on the other hand, is a pragmatic bottom-up approach which addresses actual security situations, after all other security mechanisms were broken or avoided.

Preventing low-level code injection at processor level is part of defensive security. It stops the malware injection at the last moment, after all upstream malware detection tools were deceived, after bugs in legitimate application code were exploited to inject code.

2.1. Control Flow Integrity (CFI)

Malware code injection using ROP and JOP attacks can be tracked down to a unique common cause: branching at unexpected code locations.

In practice, legitimate code runs in an application or in the operating system. This legitimate code has an expected call flow, where branch instructions jump to some expected locations. However, at some point, because of some software bug, the processor suddenly branches to an unexpected location, typically in some sequence which represents the malware code.

Differentiating legitimate and unexpected branches in the code is called Call Flow Integrity (CFI).

Using CFI techniques, a processor can:

- mitigate Return-Oriented Programming (ROP) attacks, typically from buffer overflows on stack,
- mitigate Jump-Oriented Programming (JOP) attacks, typically from buffer overflows on heap.

Adding CFI capabilities to a processor usually requires improvements in the Instruction Set Architecture (ISA), the compilers and the kernel of the operating system.

Many CFI research projects exist around the globe. They define extensions to common ISA's such as Intel, MIPS, Arm, RISC-V. The most complete research project is CHERI (Capability Hardware Enhanced RISC Instructions) from the University of Cambridge and SRI International [18]. This is a multi-architecture project which proposes extensions to Intel, MIPS, Arm, or RISC-V. In the RISC-V community, many competing CFI projects exist.

In production, available today in current processors, the Arm64 architecture probably provides the most efficient set of Control Flow Integrity features.

2.2. Available mechanisms in Arm64 architecture

The CFI features in the Arm64 architecture are implemented using two complementary mechanisms:

- Pointer Authentication Code (PAC)
- Branch Target Identification (BTI)

PAC focuses on the source of branch instructions.

Using PAC, all pointers, to code and data, can be *authenticated*. A common technique of code injection consists in altering some memory location which contains a pointer to code. Any alteration of these locations results in the later execution of some unexpected code. The purpose of PAC is detecting these alterations through authentication mechanisms.

PAC will be described in detail in section 3.2.

BTI focuses on the target of branch instructions.

In most code, very few instructions are targets of legitimate branches. On the other hand, most gadget entry points in ROP and JOP attacks are not the target of any legitimate branch. Using BTI, legitimate targets are clearly identified. Branching to any other instruction is a clear symptom of code injection attack and results in an exception.

BTI will be described in detail in section 3.1.

PAC is supported on Armv8.3-A onwards, with gcc 7.1, clang 8.0, and the Linux kernel 5.0 (in user code) or 5.14 (in kernel code).

BTI is supported on Armv8.5-A onwards, with GCC 9.1, clang 8.0, and the Linux kernel 5.8.

2.3. Impact on security and performance

As usual, there is a trade-off between security and performance. The optimal solution is obtaining the highest level of security with the lowest impact on performance.

As we will see later in this document, most of the magic is in the compiler. Based on the current state of the compilers, Arm did some benchmarking which are summarized below. Considering that these results were established by Arm, they should be interpreted with some precaution. However, experience demonstrates that they are very close to reality.

The security benefit is measured by the number of remaining potential gadgets in standard applications and libraries.

The exercise was done on the GLIBC library, using PAC, BTI or a combination of the two. Without any of them, all instructions in the GLIBC code are potential gadget entry points. With BTI, only explicit BTI instructions can be used as gadgets by JOP attacks. With PAC, return instructions can no longer be used as ROP attacks[1].

The results of the Arm's benchmarks are summarized in Figure 6 below. Using PAC+BTI, the number of gadgets which are available is reduced by 97.65%.

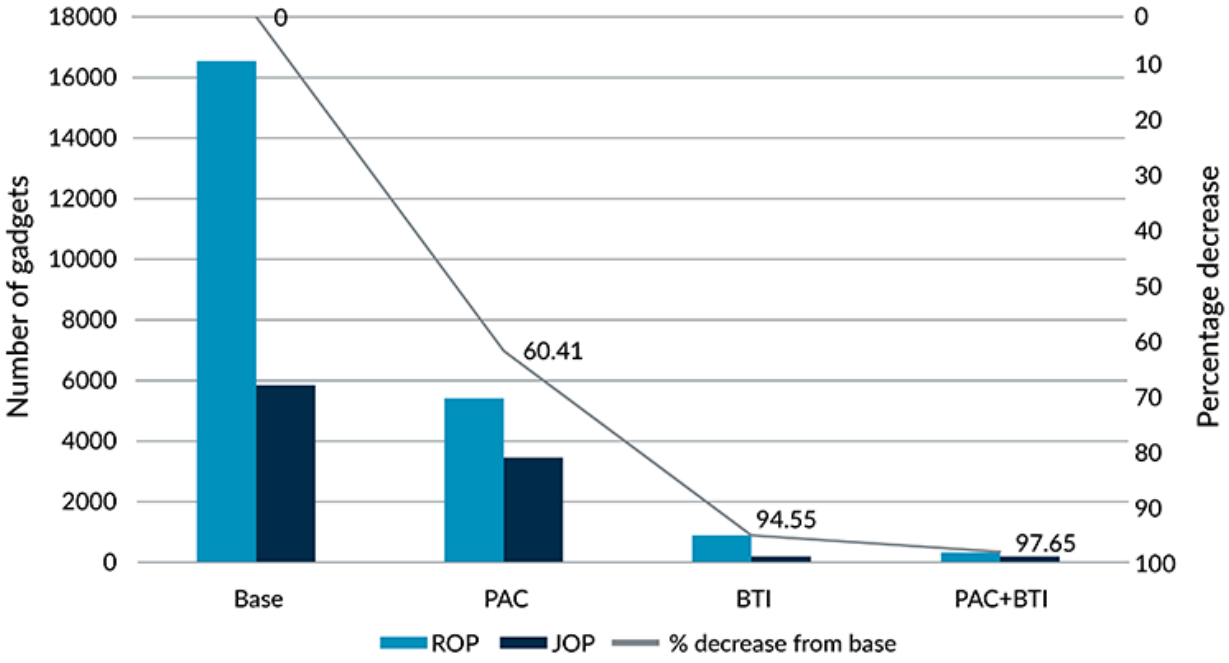

Figure 6: ROP and JOP gadgets in GLIBC © Arm [2]

There are no precise details on the test methodology to locate potential gadgets. Nonetheless, these results are credible. The only surprising point is the benefit of BTI alone, without PAC. It could be expected to have a larger impact on JOP attacks and a lower impact on ROP attacks.

The impact on code size has been evaluated by Arm on the GLIBC library as well. The results are summarized in Figure 7 below.

With PAC+BTI protection the increase of code size is 2.9% with backward-compatible code and only 1.6% when the code is compiled for Armv8.3-A or higher.

As explained in the next sections of this paper, PAC and BTI features are implemented starting with versions v8.3 and v8.5 of the Arm architecture, respectively. It is possible to generate code which is backward compatible on older versions of the architecture. In that case, the added instructions behave as NOP on older CPU cores. However, the code generation is limited to a subset of the PAC features and generates more instructions (larger code).

When the code is specifically compiled for Armv8.3-A onwards, new and more compact instructions can be used instead.

Therefore, it is recommended, when possible, to generate specialized binaries for Armv8.3-A onwards. The resulting binaries have less instructions, they are faster, and they potentially protect against a wider range of ROP and JOP attacks.

[1] PAC could be used to mitigate a larger number of attacks but, currently, Linux does not implement that kind of protection, only macOS does. See more details later in this document.

Note that BTI is fully backward compatible and fully independent of the CPU core. The right selection of the compilation target matters with PAC only.

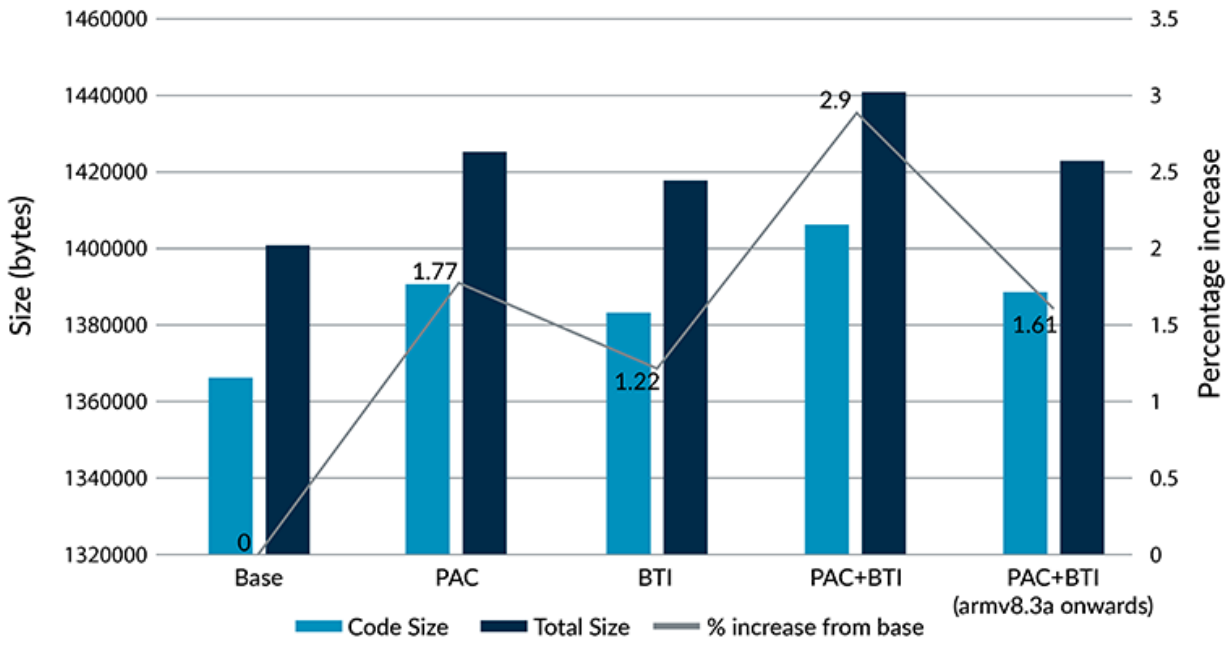

Figure 7: Change in image size from enabling ROP and JOP mitigation © Arm [2]

The impact on performance was not evaluated in Arm's benchmark. It is the direct consequence of the execution of the additional instructions. Therefore, the impact on performance depends on the implementation of each CPU core and cannot be evaluated in the general case.

However, the added instructions for PAC and BTI are straightforward. They do not involve any form of branch (except when they generate an exception) and probably do not break any form of speculative execution. Therefore, the performance impact is directly related to the number of added instructions.

In the *Arm Software Optimization Guides* for the Neoverse N2 [5], V1 [3], and V2 [4] CPU cores, the PACxx and AUTxx instructions have a latency of 5 cycles. This is between an integer multiplication (2 to 4 cycles) and an integer division (5 to 20 cycles).

PAC instructions are also sensitive to the speed of the underlying cipher algorithm. Several algorithms are allowed. CPU cores using the QARMA3 algorithm instead of QARMA5 may expect a small performance benefit, at the expense of security. All initial Arm-designed cores such as the Neoverse V1 implement QARMA5. This is explained in detail in section 3.2.9.

2.4. Arm optional and mandatory features

2.4.1. Arm architecture profiles, versions, and features

The Arm architecture is flexible and available in many flavors.

At a high level, there are several *profiles* for distinct classes of devices. The M-profile addresses small mobile devices. The A-profile addresses application processors. In this paper, we only consider the A-profile, as used in server-class systems.

In each profile, successive *versions* are defined. In 2023, existing 64-bit processors implement versions Armv8.0-A to Armv8.5-A. At the architecture specification level, the latest version is Armv9.2-A, although it is currently not yet implemented in any CPU.

In each profile and version, several *features* are specified. Each feature has a name in the form FEAT_xxx.

Features are specified as mandatory starting at some version and possibly mandatory starting at some other higher version.

As of Armv9.2-A, the architecture defines 244 features [21].

> Identifying the exact set of features that an Arm-based core implements is not as easy as it seems. For Arm-designed cores, the Technical Reference Manual (TRM) does not provide a precise list of all FEAT_xxx for that core. Out of 244 Arm features, 224 can be detected from software [20] but software can be run only if a chip is already available with that core.

2.4.2. Arm features for PAC and BTI

The Table 2 below lists all Arm features which implement PAC and BTI.

Table 2: Arm features for PAC and BTI

Feature	Optional	Mandatory	Description
FEAT_BTI	—	Armv8.5A	Branch target identification
FEAT_PAuth	—	Armv8.3A	Pointer authentication
FEAT_PAuth2	Armv8.3A	Armv8.6A	Enhanced pointer authentication: XOR PAC with upper bits
FEAT_CONSTPACFIELD	Armv8.3A	—	Enhanced pointer authentication: use bit 55 to compute PAC size and location
FEAT_EPAC	Armv8.3A	—	Enhanced pointer authentication: set PAC to zero on non-canonical addresses
FEAT_FPAC	Armv8.3A	—	Fault on pointer authentication failure (AUTxx)
FEAT_FPACCOMBINE	Armv8.3A	—	Fault on pointer authentication failure in combined instructions (RETAA, LDRAA, etc.)
FEAT_PACIMP	Armv8.3A	—	Pointer authentication: implementation-defined algorithm (e.g., Apple cores)
FEAT_PACQARMA3	Armv8.3A	—	Pointer authentication: QARMA3 algorithm (3-round QARMA64)
FEAT_PACQARMA5	Armv8.3A	—	Pointer authentication: QARMA5 algorithm (5-round QARMA64)

PAC is made of a set of complementary features, some of them are mandatory, some others are optional. BTI, on the other hand, is simpler in principle and made of one single feature.

These features are materialized by new mechanisms in the CPU cores.

New system registers:

- Five 128-bit registers for PAC cipher keys.

New virtual memory management features:

- BTI guarded pages.

- Virtual address manipulation (PAC and/or MTE[2] tag in upper bits).

The new instructions can be classified as follows:

- 49 new instructions[3] (48 for PAC, 1 for BTI).
- 14 new instructions are "hints", they act as a NOP on CPU cores which don't support the feature.
  - These instructions provide backward compatibility.
  - BTI is fully backward compatible.
- 35 new instructions are not hints.
  - Software may need two binary packages, one is fully backward compatible, the other is more secure, faster, more compact, but not backward compatible.

This very last point introduces a new issue: the compilation dilemma.

2.4.3. Arm features and the compilation dilemma

A target platform is defined by the features which are implemented in the CPU core. This set of features has an impact on the compilation and code generation, mostly because some features define new or specific instructions. Executing these instructions on a CPU core which does not implement the corresponding feature results in an illegal instruction exception.

To address the diversity of CPU cores, there is a compilation dilemma with two options:

- Compile each application for a specific target, using the right compilation options for the features of the target CPU core.
- Compile generic binaries which do not use specific instructions, or only the backward-compatible ones, just using the basic common Arm features.

The first option is typically used in embedded systems. The target is well defined, and the corresponding Software Development Kit (SDK) is specifically tuned for that target.

In the server, cloud, or even desktop market, the requirements are different. Users do not compile for the target. They don't even compile at all. They download binary packages for Linux Ubuntu, Red Hat, or some other distribution (letting aside Windows or macOS) and they expect the system to work at the best of its capabilities.

In this context, the current approach is the second option of the compilation dilemma: provide a common, compatible, set of binary packages.

As of mid-2023, the binary Linux kernels which are installed with recent versions of Ubuntu, Debian, Fedora, are built with the backward-compatible subset of PAC. This is better than nothing, but not much. Applications and libraries are not compiled with PAC or BTI.

This is not satisfactory. There is a huge potential for security improvements in Armv8.3-A onwards. Armv9-A will add even more. What's the use of having such a potential if we do not use it?

On the other hand, building a completely protected binary system won't work on older Arm systems.

This problem was not perceivable when Arm CPU cores were used in embedded devices only because of the "compile for the target" rule. Now that Arm is used in larger, generic systems, the open-source community should tackle that problem.

[2] MTE is the Memory Tagging Extension, a debug feature to track the usage of memory.
[3] The Armv9.2-A ISA defines more than 1600 instructions, vector (SVE) and matrix (SME) extensions included.

Binary distributions should be produced for a few predefined target profiles, for example: basic Armv8.0-A systems, CFI-protected Armv8.3-A systems (full PAC+BTI support) and maybe CCA-enhanced Armv9-A systems in the future.

> This is not new. There was a time when distinct packages were available for i386 and i686 targets. The i386 packages could work on any x86 system but ignoring the additional capabilities of i686 systems was considered not satisfactory.

2.5. Arm64 control flow integrity features in SiPearl processors

As mentioned in section 1.1, Rhea, the first generation of SiPearl processors, addresses the HPC market. It is based on Arm Neoverse V1 cores. This core implements the Armv8.4-A architecture. As such, it includes PAC features but not BTI.

The exact list of CFI features in Neoverse V1 is the following: FEAT_PAuth, FEAT_PAuth2, FEAT_EPAC, FEAT_PACQARMA5.

The next generation of SiPearl processors will target the cloud, data center and server market. These processors use CPU cores at Armv9.0-A and Armv9.2-A levels. They implement all PAC and BTI features and are particularly suited to enhance the cybersecurity chain of the data centers by mitigating malware injection attacks.

From a security perspective, these Armv9 cores also implement the Arm Confidential Computer Architecture (CCA). It defines a powerful model of hardware segmentation between so-called “realms” (think “virtual machines”). This segmentation is most wanted in cloud environments where complete strangers share the same physical infrastructure.

This exciting security topic can be described as “security by design” or “proactive security” and will be addressed in a separate white paper.

3. Control flow integrity techniques in details

The last part of this white paper dives into the gory details of PAC and BTI on Arm64 processors. If you just want an overview of control flow integrity techniques, it is not necessary to read it. On the other hand, if you want to improve the compilers or the Linux kernel to take full advantage of PAC (there is a lot of room for improvement), this is a recommended reading.

Although it is common to mention PAC+BTI in this order, let’s start with BTI for one simple reason: it is simpler than PAC.

3.1. Branch Target Identification (BTI)

3.1.1. BTI principles

BTI focuses on the target of illegitimate jump operations.

How can an existing branch instruction perform an illegitimate jump? First, it is worth noting that there are two types of branch instructions: direct and indirect.

In an ISA with fixed-length instruction such as Arm64, all direct branch instructions are *relative*. It reads, for instance, “*branch 136 bytes ahead of current program counter*”. Since code pages are read-only, they cannot

be altered and the offset value “136” in the example is hardcoded in the instruction. Thus, direct branches are always legitimate and not subject to code injection.

Indirect branches, on the other hand, are vulnerable. When an instruction is jump or call at address from register X16, for instance, we do not know where the content of this register comes from. It has probably been loaded or computed from a memory location. This location may have been altered by a buffer overflow.

This is why the BTI mechanism only focuses on indirect jumps.

In real code, very few instructions are legitimate targets for indirect jumps. There are the function entry points, the various branches of switch / case structures with a dispatch table, and a few other corner cases. That’s all. In practice, this represents approximately 1% of all instructions (1.22% in GLIBC, according to Figure 7, page 13).

On the other hand, an illegitimate jump into a gadget is a branch at some arbitrary location, where no legitimate indirect jump is supposed to land.

The idea behind BTI is to clearly identify which instructions are valid targets for indirect jumps (1% of the code) and which are not (99% of the code).

In practice, a new instruction named BTI is introduced. This is just a placeholder. Functionally, this is a NOP. However, it identifies a legitimate target for an indirect branch. The compiler adds a BTI instruction wherever it generates code for such a legitimate target.

During execution, the processor carefully checks that all indirect jump instructions land at a BTI instruction. Otherwise, trying to branch to any other instruction is a clear symptom of ROP or JOP attack and a fault is generated.

This is illustrated in Figure 8 below.

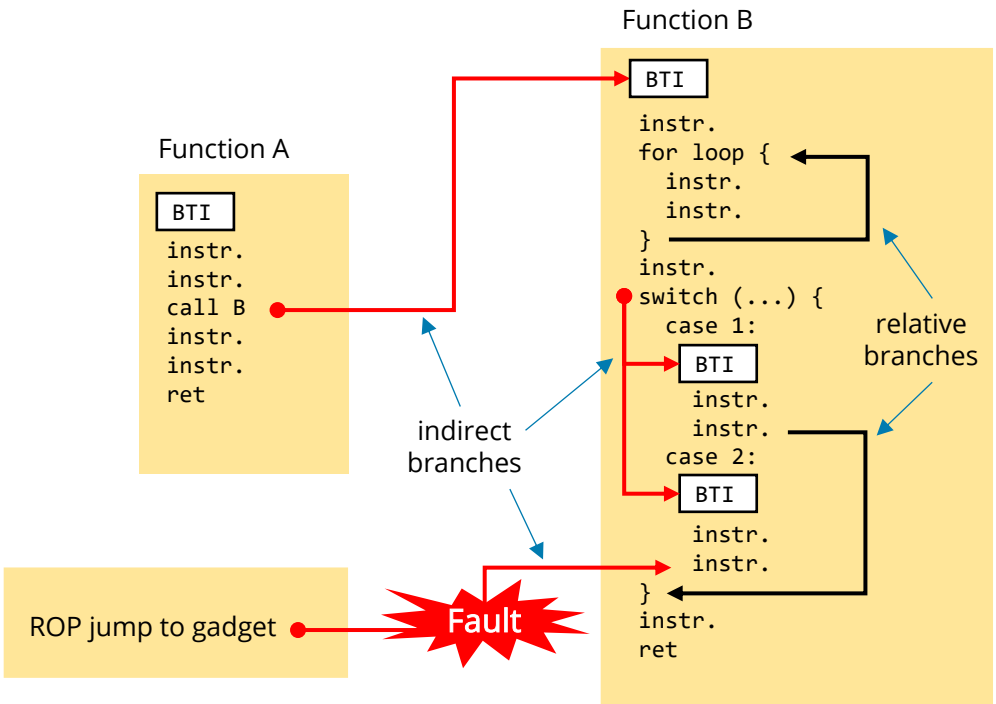

Figure 8: Branch target identification principles

We can see that relative branches do not need a BTI instruction since they are hard-coded and cannot be circumvented.

A ROP jump into a gadget, on the other hand, is an indirect jump to an arbitrary instruction which is not a BTI 99% of the time and will generate a fault.

What about the remaining 1%? There is indeed a 1% risk of having a gadget starting at a BTI instruction. However, gadgets are small pieces of code. Executing a complete malware bootloader means chaining N gadgets. And the probability of having N gadgets in a row starting with BTI is 0.01^N, which tends to zero when N grows.

3.1.2. Backward compatibility

Hardware backward compatibility

To address hardware backward compatibility, the BTI instruction has been allocated in the HINT range. This means that CPU cores which do not implement BTI (typically before Armv8.5-A) simply ignore the BTI instruction.

This is why it is recommended to compile all new code with BTI instructions. It will run on all Arm64 processors.

Software backward compatibility

Not all existing code is compiled with BTI instructions. Legacy applications were built with no BTI, and we still need to run them on recent CPU cores with BTI support.

To support this use case, the Armv8.5-A architecture introduces the concept of *guarded page*. The guarded page indicator is a bit in the page tables of the virtual memory system (just like the non-execute bit). Filtering indirect jumps on BTI is active only if the target instruction is in a guarded page. Otherwise, indirect jumps are allowed to land anywhere in a non-guarded page.

In practice, when the compiler generates BTI instructions, it marks the code as such. When the linker grabs all modules (which may have been separately compiled with or without BTI), it groups code with BTI in guarded pages and code without BTI in non-guarded pages. Legacy application binaries have no guarded pages.

Thus, the complete backward compatibility and interoperability between old and new code is guaranteed.

3.1.3. The BTI instruction

Let's now study the BTI instruction in detail.

BTI is not simply a generic landing place for indirect jumps. It is possible to filter which types of indirect jump are allowed here. For this finer level of filtering, the BTI has a 2-bit operand with four possible values, as illustrated in Table 3 below.

Table 3: BTI operand

Instruction	HINT	Operand	Description
BTI	HINT 32	00	No branch is allowed to land here (not used in practice)
BTI c	HINT 34	01	Only calls ("c") are allowed to land here (BLR-type instructions)
BTI j	HINT 36	10	Only jumps ("j") are allowed to land here (BR-type instructions)
BTI jc	HINT 38	11	Calls and jumps are allowed to land here

Thus, the entry point of a function is a “BTI c” instruction while the first instruction of a switch / case branch is a “BTI j” for instance.

3.1.4. The PSTATE.BTYPE new processor state

Starting with Armv8.5-A, a new 2-bit processor state named PSTATE.BTYPE has been introduced.

The processor state is set by each instruction and describes the “branch type” of the current instruction. The possible values of PSTATE.BTYPE are presented in Table 4 below.

Table 4: How PSTATE.BTYPE is set by instructions

Executed instruction	From region	Register used	PSTATE.BTYPE
BLR, BLRAA, BLRAAZ, BLRAB, BLRABZ	Any	Any	10
BR, BRAA, BRAAZ, BRAB, BRABZ	Guarded	Any, except X16, X17	11
BR, BRAA, BRAAZ, BRAB, BRABZ	Guarded	X16, X17	01
BR, BRAA, BRAAZ, BRAB, BRABZ	Non guarded	Any	01
RET, RETAA, RETAB	Any	Any	00
Other instructions (non-indirect branch)			00

We can interpret PSTATE.BTYPE as follows:

- 00 : not an indirect branch (except RET), no branch hijacking is possible.
- 10 : indirect branch to subroutine.
- 11 : indirect branch from controlled code, i.e., from a guarded page, not using X16/X17 (IP0/IP1).
- 10 : indirect branch from uncontrolled code, i.e., from non-guarded page or using X16/X17.

As a general rule, all indirect branch instructions set PSTATE.BTYPE to a non-zero value (with the notable exception of return instructions).

Table 5: BTI operand / PSTATE.BTYPE compatibility

BTI operand	BTYPE = 00 Not hackable	BTYPE = 01 Jump from uncontrolled	BTYPE = 10 Call function	BTYPE = 11 Jump from controlled
00 (none)	Not compatible	Not compatible	Not compatible	Not compatible
01 (c)	Compatible	Compatible	Compatible	Not compatible
10 (j)	Compatible	Compatible	Not compatible	Compatible
11 (jc)	Compatible	Compatible	Compatible	Compatible

When the next instruction to be executed is inside a guarded page, PSTATE.BTYPE is checked. When this state value is non-zero (i.e., the processor executes some form of indirect branch), a *branch target exception* is generated, unless the instruction is one of:

- BTI instruction with a compatible PSTATE.BTYPE (see Table 5 above)
- BRK instruction (breakpoint exception)
- HLT instruction (debug event)
- PACIASP, PACIBSP with
  - PSTATE.BTYPE = 01 or 10
  - PSTATE.BTYPE = 11 and SCTLR_EL1.BT0/1 = 0 (plus additional control by hypervisor and monitor through SCTLR_EL2 and SCTLR_EL3)

The reason for the very last rule on PACIxSP instructions is unclear. Maybe it was specified to avoid a BTI instruction when a function starts with a PACIxSP instruction.

Note: RET instructions are not under BTI control. Control flow integrity on return is based on PAC, authentication of the pointer in X30. A probable reason for this is that controlling return targets would require a BTI instruction after each call instruction (*branch with link register*). This would inflate the code and would create too many valid gadgets. Controlling RET through BTI would consequently have a negative impact on code size, performance, and security.

3.1.5. Compiler support

The gcc and clang compilers support a common option named `-mbranch-protection`.

This option manages the various types of branch protection. To build a module with branch target identification, use `-mbranch-protection=bti`.

It is also possible to use `-mbranch-protection=standard`. This option includes all "standard" branch protection features and BTI is one of them.

Function entry points

The sample code below illustrates the prolog and epilog of a function, without and with BTI.

Source code

```
int func1()
{
    ...
    return 0;
}
```

gcc -c test.c

```
stp  x29, x30, [sp, #-16]!
mov  x29, sp
...
mov  w0, #0x0
ldp  x29, x30, [sp], #16
ret
```

gcc -c test.c -mbranch-protection=bti

```
bti  c
stp  x29, x30, [sp, #-16]!
mov  x29, sp
...
mov  w0, #0x0
ldp  x29, x30, [sp], #16
ret
```

The "bti c" instruction is the only difference between the two code generations. It marks the only valid landing instruction for indirect jumps.

To make sure that the guarded page information is propagated all the way down to the linker, the compiler sets the BTI information in the special section named `.note.gnu.property`. This is demonstrated below, using the object file which was compiled with `-mbranch-protection=bti`.

```
$ readelf test.o -n -x .note.gnu.property

Hex dump of section '.note.gnu.property':
  0x00000000 04000000 10000000 05000000 474e5500 ............GNU.
  0x00000010 000000c0 04000000 01000000 00000000 ................

Displaying notes found in: .note.gnu.property
  Owner                Data size        Description
  GNU                  0x00000010       NT_GNU_PROPERTY_TYPE_0
      Properties: AArch64 feature: BTI
```

Switch / case structures

The next example illustrates the protected code for a switch / case structure.

```
int func1(int x)
{
    ...
    switch (x) {
        case 0: return 0;
        case 1: return 1;
        case 2: return 2;
        ...
        case 99: return 99;
        default: return -1;
    }
}
```

The generated code illustrates the following differences:

- Targets of indirect calls (“bti c”) vs. indirect jumps (“bti j”).
- Targets of indirect branches (with a BTI instruction) vs. direct branches (without).

gcc -c test.c -mbranch-protection=bti

```
target of any type of call  →    0:  bti  c                    ; function prolog
      (possibly indirect)        4:  stp  x29, x30, [sp, #-32]!
                                 8:  mov  x29, sp
                                     ...
                                18:  cmp  w0, #0x63            ; test against 99
             direct jump        →  1c:  b.hi 4e8 <func1+0x4e8>  ; jump to “default:”
                                20:  adrp x1, 0 <func1>        ; load dispatch table
                                24:  add  x1, x1, #0x0         ; (here, a computed address)
                                28:  ldr  w0, [x1, w0, uxtw #2]
                                2c:  adr  x1, 38 <func1+0x38>
                                30:  add  x0, x1, w0, sxtw #2
          indirect jump @x0     →  34:  br   x0                ; jump to computed target
  target of indirect jump       →  38:  bti  j                 ; “case 0:”
                                3c:  mov  w0, #0x0
             direct jump        →  40:  b    4ec <func1+0x4ec> ; jump to epilog
  target of indirect jump       →  44:  bti  j                 ; “case 1:”
                                48:  mov  w0, #0x1
             direct jump        →  4c:  b    4ec <func1+0x4ec> ; jump to epilog
  target of indirect jump       →  50:  bti  j                 ; “case 2:”
                                54:  mov  w0, #0x2
             direct jump        →  58:  b    4ec <func1+0x4ec> ; jump to epilog
                                     ...
  target of indirect jump       → 4dc:  bti  j                 ; “case 99:”
                                4e0:  mov  w0, #0x63
             direct jump        → 4e4:  b    4ec <func1+0x4ec> ; jump to epilog
    target of direct jump       → 4e8:  mov  w0, #0xffffffff   ; “default:”
    target of direct jump       → 4ec:  ldp  x29, x30, [sp], #32 ; function epilog
                                4f0:  ret
```

3.2. Pointer Authentication Code (PAC)

3.2.6. PAC principles

PAC focuses on the source of illegitimate jump operations.

ROP and JOP attacks are made of jumps to unexpected locations, forming a gadget chain which usually acts as a bootstrap for some malware payload. The branch instructions are part of some valid code. However, the value of the pointers they use were somehow altered. Typically, these pointers were stored in memory (a return pointer on stack, a function address in a dispatch table) and were altered in memory after some buffer overflow attack.

The Pointer Authentication Code feature provides a way to verify that a pointer to code or data was not altered before dereferencing it.

The principle is simple. On 64-bit systems, code or data pointers never use 64 bits in virtual address values. Using all bits would mean that a process addresses 16 exabytes (16 billion of billions of bytes). As of today, this is unlikely.

Consequently, PAC uses the upper bits of a virtual address to store a signature of that address.

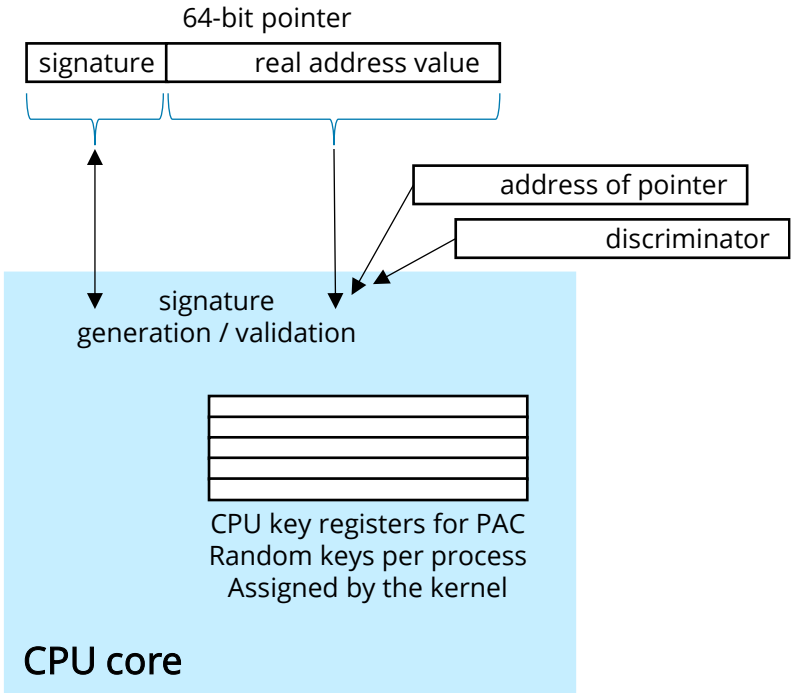

Figure 9: Pointer authentication code principles

Cryptographic aspects

In cryptographic terms, the PAC is a CMAC [28], a cipher-based message authentication code. It uses a symmetric cryptographic algorithm, in the CPU core hardware, to compute a CMAC-like signature.

There are five distinct special registers (2 code, 2 data, 1 generic) in the CPU which contain five possible secret keys for the PAC symmetric cipher algorithm.

These key registers are accessible at EL1 only (in kernel mode). User applications cannot read them and predict the signature of an address. This is done on purpose to prevent malicious software from forging the signature of a hacked address.

For enhanced protection, it is common to add a seed, based on the placement and semantics of the signed pointer. More on this will be found in section 3.2.13.

This is illustrated in Figure 9 above. The seed is typically a combination of the address where the pointer is stored and some semantic discriminator value.

Protect memory, not registers

ROP and JOP attacks overwrite data in memory. Usually, the content of registers is considered as safe. A register may load an altered value from memory but a ROP or JOP attack cannot modify the content of a register. Therefore, the signature is added to a pointer just before writing it in memory. Conversely, the signature is verified right after loading a pointer value from memory.

Explicit instructions

Adding and verifying a signature is done explicitly, using dedicated instructions which are generated by the compiler. These instructions are PACxx to add a signature and AUTxx to verify it. More complex combined instructions also exist.

The validation operation removes the signature when it is valid. Then, the register contains the plain 64-bit address. It should be noted that this is required to use the pointer. A pointer which still contains a signature is probably not a valid address and dereferencing it would generate an exception.

If the validation of the signature fails, either the pointer is left to a non-canonical address (an invalid virtual address which generates a fault) or a fault is immediately generated. The exact behavior depends on the FEAT_FPAC feature.

3.2.7. Common PAC use case: function call

All PAC operations are done in the generated code of the called function. Thus, there is no interoperability issue between the caller and the callee. The caller can be compiled with or without PAC, the callee takes the responsibility of its own protection alone.

Step 1: protection.

- The function is called. The BLR instruction sets the return address in register X30, for instance:

  `X30 = 00000012-3456789A`

- In the function prolog, a PAC instruction (e.g., PACIASP) adds an authentication code in upper bits:

  `X30 = D90E8012-3456789A`

- The signed return address is pushed on stack, in the call frame.

Step 2 : authentication.

- The function epilog pops the return pointer from stack into X30. The signature is still present:

  `X30 = D90E8012-3456789A`

- An AUT instruction (e.g., AUTIASP) checks the PAC and removes it if still correct:

  `X30 = 00000012-3456789A`

- The final RET instruction returns to the caller through X30.
- Variant: if the target CPU core is Armv8.3-A or higher, one single RETAA instruction can be used instead of AUTIASP + RET.

Step 2 : in case of hack, e.g., a ROP attack exploiting a stack overflow bug.

- The altered return pointer is popped from stack into X30. If even a single bit is altered:

  `X30 = D90E8012-3456789B`

- The AUT instruction detects the invalid PAC and leaves a non-canonical address in register X30:

  `X30 = 20000012-3456789B`

- The final RET instruction tries to return to the caller through X30. Since the register contains a non-canonical address, an exception is generated and the hacked application crashes.
- Conclusion: the ROP attack failed.
- Variant: if the CPU core implements FEAT_FPAC ("Fault on PAC"), the AUTIASP or RETAA instruction directly generates a fault when an incorrect PAC is detected, without waiting for the non-canonical address to be dereferenced.

More complex use case: tables of pointers

Consider a table of pointers which may reference data or code. It can be a switch dispatch table or a C++ vtable (the table containing the addresses of all virtual methods for a class).

- Each pointer is signed with a process-specific key. In the case of a static table, the signature is added in the initialization phase of the executable, when virtual addresses are resolved.
- In the case of C++ objects, the address of the vtable is separately signed in all object instances.
- All load, store or jump instructions using a pointer validate the signature.
- In case of JOP buffer overflow attack, the altered pointer has an invalid signature and dereferencing generates a fault.

A fully commented example of this scenario is provided in section 3.2.15.

This use case requires more care in terms of interoperability between modules which are compiled with different options. The technical tools to solve the interoperability issue are available in the Arm architecture. However, this requires the cooperation of the operating system.

Currently, this use case is implemented on macOS only. Linux leaves all tables of pointers unprotected. There is a lot of room for improvement here for the Linux kernel, gcc and clang.

3.2.8. PAC instructions

The three main families of PAC instructions are listed below.

- PACxx instructions compute a pointer signature and add it in the upper bits of that pointer.
- AUTxx instructions validate the signature of a pointer and remove it if it is correct.
- XPACxx instructions remove a signature without validating it. They are rarely used.

Other PAC instructions are *combined* instructions. This means that, in one instruction, they validate the signature of a pointer and, if it is correct, dereference it to branch to code (BRAx, BLRAx), return to the caller (RETAx, ERETAx) or load data (LDRAx).

The Table 6 below lists the 48 new instructions for PAC.

The 13 instructions with a blue bar on the left are in the HINT range and behave as NOP on older CPU cores which do not support PAC. Any code using only this subset of the PAC instructions is backward-compatible.

Table 6: New instructions for pointer authentication code (PAC)

AUTDA Xd, Xn|SP	AUTIBZ	LDRAA Xt, [Xn|SP, #i]!	PACIAZ
AUTDZA Xd	AUTIZB Xd	LDRAA Xt, [Xn|SP, #i]	PACIZA Xd
AUTDB Xd, Xn|SP	BLRAA Xn, Xm|SP	LDRAB Xt, [Xn|SP, #i]!	PACIB Xd, Xn|SP
AUTDZB Xd	BLRAAZ Xn	LDRAB Xt, [Xn|SP, #i]	PACIB1716
AUTIA Xd, Xn|SP	BLRAB Xn, Xm|SP	PACDA Xd, Xn|SP	PACIBSP
AUTIA1716	BLRABZ Xn	PACDZA Xd	PACIBZ
AUTIASP	BRAA Xn, Xm|SP	PACDB Xd, Xn|SP	PACIZB Xd
AUTIAZ	BRAAZ Xn	PACDZB Xd	RETAA
AUTIZA Xd	BRAB Xn, Xm|SP	PACGA Xd, Xn, Xm|SP	RETAB
AUTIB Xd, Xn|SP	BRABZ Xn	PACIA Xd, Xn|SP	XPACD Xd
AUTIB1716	ERETAA	PACIA1716	XPACI Xd
AUTIBSP	ERETAB	PACIASP	XPACLRI

The Table 7 below details all new PAC instructions.

Some instructions are redundant for the sake of backward compatibility. When the PAC feature was specified, it was not possible to add all new instructions, with their variety of operands, in the HINT range. Instead, a few dedicated use cases were identified and the corresponding PAC instructions, with predefined operands, were reassigned in the HINT range.

Table 7: Detailed description of PAC instructions

Instruction	Description
AUTDA Xd, Xn|SP	Authenticate data address in Xd, using key A and modifier from Xn
AUTDZA Xd	Authenticate data address in Xd, using key A and modifier zero
AUTDB Xd, Xn|SP	Authenticate data address in Xd, using key B and modifier from Xn
AUTDZB Xd	Authenticate data address in Xd, using key B and modifier zero
AUTIA Xd, Xn|SP	Authenticate instr. address in Xd, using key A and modifier from Xn
AUTIA1716	Authenticate instr. address in X17, using key A and modifier from X16 *same as AUTIA X17,X16, as a HINT instruction for "trampoline" use case*
AUTIASP	Authenticate instr. address in X30, using key A and modifier from SP *same as AUTIA X30,SP, as a HINT instruction for "return" use case*
AUTIAZ	Authenticate instr. address in X30, using key A and modifier zero *same as AUTIZA X30, as a HINT instruction for (unsafe) "return" use case*
AUTIZA Xd	Authenticate instr. address in Xd, using key A and modifier zero
AUTIB Xd, Xn|SP	Authenticate instr. address in Xd, using key B and modifier from Xn
AUTIB1716	Authenticate instr. address in X17, using key B and modifier from X16 *same as AUTIB X17,X16, as a HINT instruction for "trampoline" use case*
AUTIBSP	Authenticate instr. address in X30, using key B and modifier from SP *same as AUTIB X30,SP, as a HINT instruction for "return" use case*
AUTIBZ	Authenticate instr. address in X30, using key B and modifier zero *same as AUTIZB X30, as a HINT instruction for (unsafe) "return" use case*
AUTIZB Xd	Authenticate instr. address in Xd, using key B and modifier zero
BLRAA Xn, Xm|SP	Branch with Link to @Xn with auth, key A, modifier Xm, ret addr => X30
BLRAAZ Xn	Branch with Link to @Xn with auth, key A, modifier zero, ret addr => X30
BLRAB Xn, Xm|SP	Branch with Link to @Xn with auth, key B, modifier Xm , ret addr => X30
BLRABZ Xn	Branch with Link to @Xn with auth, key B, modifier zero, ret addr => X30

Instruction	Description
BRAA Xn, Xm|SP	Branch to @Xn with auth, key A, modifier Xm
BRAAZ Xn	Branch to @Xn with auth, key A, modifier zero
BRAB Xn, Xm|SP	Branch to @Xn with auth, key B, modifier Xm
BRABZ Xn	Branch to @Xn with auth, key B, modifier zero
ERETAA	Exception return, authenticate ELR, using key A and modifier from SP
ERETAB	Exception return, authenticate ELR, using key B and modifier from SP
LDRAA Xt, [Xn|SP, #i]!	Load Xt with @Xn, authenticate Xn, using key A and modifier zero, *pre-indexed mode => leaves Xn|SP unauthenticated*
LDRAA Xt, [Xn|SP, #i]	Load Xt with @Xn, authenticate Xn, using key A and modifier zero
LDRAB Xt, [Xn|SP, #i]!	Load Xt with @Xn, authenticate Xn, using key B and modifier zero, *pre-indexed mode => leaves Xn|SP unauthenticated*
LDRAB Xt, [Xn|SP, #i]	Load Xt with @Xn, authenticate Xn, using key N and modifier zero
PACDA Xd, Xn|SP	Compute auth. of data address in Xd, using key A and modifier Xn
PACDZA Xd	Compute auth. of data address in Xd, using key A and modifier zero
PACDB Xd, Xn|SP	Compute auth. of data address in Xd, using key B and modifier Xn
PACDZB Xd	Compute auth. of data address in Xd, using key B and modifier zero
PACGA Xd, Xn, Xm|SP	Compute in Xd the auth. of Xn using the Generic Key and modifier Xm
PACIA Xd, Xn|SP	Compute auth. of instr. address in Xd, using key A and modifier Xn
PACIA1716	Compute auth. of instr. address in X17, using key A and modifier X16 *same as PACIA X17,X16, as a HINT instruction for "trampoline" use case*
PACIASP	Compute auth. of instr. address in X30, using key A and modifier SP *same as PACIA X30,SP, as a HINT instruction for "return" use case*
PACIAZ	Compute auth. of instr. address in X30, using key A and modifier zero *same as PACIZA X30, as a HINT instruction for (unsafe) "return" use case*
PACIZA Xd	Compute auth. of instr. address in Xd, using key A and modifier zero
PACIB Xd, Xn|SP	Compute auth. of instr. address in Xd, using key B and modifier Xn
PACIB1716	Compute auth. of instr. address in X17, using key B and modifier X16 *same as PACIB X17,X16, as a HINT instruction for "trampoline" use case*
PACIBSP	Compute auth. of instr. address in X30, using key B and modifier SP *same as PACIB X30,SP, as a HINT instruction for "return" use case*
PACIBZ	Compute auth. of instr. address in X30, using key B and modifier zero *same as PACIZB X30, as a HINT instruction for (unsafe) "return" use case*
PACIZB Xd	Compute auth. of instr. address in Xd, using key B and modifier zero
RETAA	Authenticated return, branch to @X30 with auth, key A, modifier SP
RETAB	Authenticated return, branch to @X30 with auth, key B, modifier SP
XPACD Xd	Strip PAC from data address in Xd
XPACI Xd	Strip PAC from instr. address in Xd
XPACLRI	Strip PAC from instr. address in X30 *same as XPACI X30, as a HINT instruction for "return" use case*

3.2.9. PAC cipher algorithms

The pointer authentication code in the upper bits of a signed pointer is built using a symmetric cipher algorithm. In practice, a specific algorithm is used instead of the usual AES algorithm because the requirements for the PAC algorithm are specific:

- 64-bit input, up to 64-bit output.
- Short entropy of the input message: 47 or 48 bits.
- Short entropy of the useful part of the output: 7 to 16 bits.
- Encryption only. The decryption operation – if defined – is not used.
- Must be fast to compute, within the acceptable duration of an instruction.

One single cipher algorithm is hardcoded in the CPU core and is used by all PAC instructions. However, the Arm architecture specification leaves the choice of this algorithm to the implementer.

On all Arm-designed CPU cores, the algorithm is QARMA64 [12]. This is a public algorithm which was co-designed by Arm and Qualcomm.

Unlike most classical symmetric cipher algorithms, there is no mandatory number of rounds or standard substitution box ("sbox") with QARMA64.

On all initial Arm-designed CPU cores, the algorithm is QARMA5, the 5-round variant of QARMA64 (FEAT_PACQARMA5). The Arm architecture specification also allows for QARMA3, the 3-round variant, which is faster, at the expense of security. No CPU core has been identified yet with FEAT_PACQARMA3.

The reference implementation of QARMA64 [13] proposes three distinct substitution boxes. The Arm-designed CPU cores use the sbox number two.

Non-Arm-designed CPU cores are allowed to implement a proprietary algorithm. This is identified by FEAT_PACIMP, as found in the so-called *Apple Silicon* chips such as Apple M1 and M2. In the case of the Apple-designed CPU cores, no public information is available on the selected algorithm.

3.2.10. PAC cipher keys

Just like any symmetric cipher algorithm, the PAC algorithm uses a shared secret key. The key size shall not exceed 128 bits in length. The QARMA64 algorithm uses 128-bit keys.

Because the entropy of the input (47-48 bits) and the output (7 to 16 bits) is very low, the PAC algorithm includes an additional parameter to compute the signature, called the *modifier* in the Arm architecture reference manual. In other cryptographic contexts, this kind of parameter is also named *seed*, *salt,* or *tweak*. More details on the recommended usage of the modifier are found in section 3.2.13.

The Figure 10 below illustrates the computation process of a pointer signature.

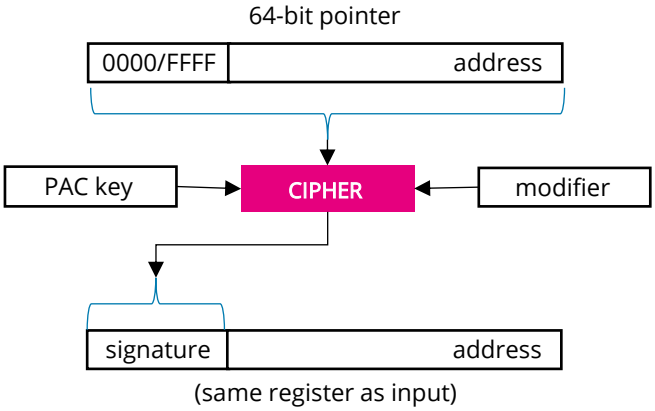

Figure 10: PAC algorithm inputs and output

The key is secret. As explained in section 3.2.6, the key is stored in a special CPU register which is accessible at EL1 only (in kernel mode). The application cannot read or modify the PAC key.

Key life cycle

The key is typically ephemeral. In most operating systems[4], a new set of random keys is allocated for each new process. Thus, two executions of the same application will use different signatures for the same address. This is done on purpose to make PAC values unpredictable. If a hacker observes the PAC values on his system, he cannot reuse them in an attack on another system, or even in another process of the same system.

The PAC keys are stored in the process context in kernel. Each time the process is scheduled on a core, the keys are loaded in the corresponding system registers, making sure the same PAC keys are used all along the execution of the process.

Instructions and data

Pointer to data and instructions are authenticated differently. In practice, they use different encryption keys.

Pointing to code or instructions cannot be determined from the pointer value alone (an address is just an address). Only the compiler knows the semantics of a pointer.

The difference between pointer to data and pointer to instructions is simply based on the PAC instruction. Building and authenticating a pointer to data is done by instructions such as PACDA and AUTDA. Building and authenticating a pointer to instructions is done by instructions such as PACIA and AUTIA. Note the “D” and “I” in the instruction to determine the type of pointed data.

Key registers

There are 5 different PAC cipher keys. There are 2 keys to sign pointers to instructions and 2 keys to sign pointers to data. In each category, there is a key named ‘A’ and a key named ‘B’. The fifth key is a bit special. It is used to implement a generic CMAC engine and is described in section 3.2.14.

PAC keys use 128 bits each. The Arm64 system registers, in contrast, are all 64 bits wide[5]. Thus, each PAC key is stored across two system registers, one for the most significant 64 bits, the other for the least significant ones. The Table 9, on page 45, lists all these registers.

User and kernel

Depending on the operating system, distinct PAC keys can be used in user and kernel mode. Since there is only one set of PAC keys for all exception levels, the APxxKeyHi/Lo_EL1 registers, the operating systems must allocate and store two sets of keys for each process and reload the PAC key registers when switching from user to kernel mode and vice versa.

Tests on the various operating systems have produced the following observations:

- On Linux, the key IA has distinct values between the user and kernel mode. The other 4 keys have the same value in the two modes. A probable explanation is that the kernel code is compiled with “pac-ret” option, using the IA key. The other PAC keys are unused in the kernel.
- On macOS, all 5 PAC keys have distinct values between the user and kernel mode.
- On Windows 11, all PAC keys have value 00000000-00000000 in all modes.

Hypervisor and monitor

For higher-level control, at hypervisor or monitor level, it is possible to trap at EL2 or EL3 on PAC instructions or access to the PAC key registers. More details on this can be found in section 3.2.17.

[4] Linux and macOS only. Windows 11 leaves all PAC keys to zero.
[5] The feature FEAT_SYSREG128 implements 128-bit system registers but was introduced later, in Armv9+.

Secret key, public modifier

The key is secret and specified by reference (the “IA” or “DB” in the instruction name). The compiler and the application do not have access to the value of the key.

The modifier, on the other hand, is explicitly provided to the instruction, as the content of a general-purpose register. Usages of modifiers are explained in section 3.2.13

3.2.11. Four address spaces

The size and position of the authentication code in a pointer depend on two criteria for this pointer:

- Pointer to data vs. pointer to instructions.
- Lower vs. upper address

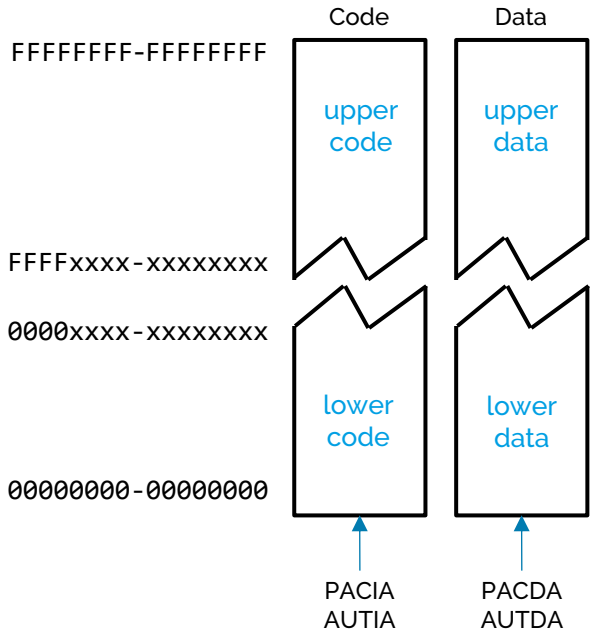

Figure 11: The four virtual address spaces for PAC

Data vs. pointer is implicit in the instruction (e.g., PACDA vs. PACIA).

Lower vs. upper address, is solely determined by the pointer value. Systems with 64-bit virtual addresses never use 2^{64} bytes. There are holes in the virtual address space. Specifically, most operating systems use lower addresses in the form 0000xxxx-xxxxxxxx and upper addresses in the form FFFFxxxx-xxxxxxxx. Although this is highly system-dependent, some operating systems use the former for applications and the latter for the kernel.

Why is this relevant to PAC?

Signing a pointer means overwriting the most significant bits in the address. Thus, the information lower vs. upper address is lost. Let’s assume that the authentication code uses the 16 upper bits of a pointer. Given the signed pointer ABCDxxxx-xxxxxxxx, ABCD is the authentication code. When the AUTxx instruction validates the signature and restores the original pointer value, which value should replace ABCD? 0000 (lower address) or FFFF (upper address)?

During the signature process (instruction PACxx), the information lower vs. address shall be preserved in order to restore the original value during AUTxx. The solution is to preserve the value of bit 55 in the pointer. If bit 55 is 0, AUTxx will restore 0000 in the pointer upper bits. If bit 55 is 1, AUTxx will restore FFFF.

Why use bit 55 and not bit 63 (most significant bit)?

The reason is Memory Tagging Extension (MTE). This is another optional Arm64 architectural feature. MTE tags each address to track the usage of the pointed data. The tag is an 8-bit value and is always located in the most significant byte of the address, or bits 63-56. To avoid interferences between PAC and MTE, the lower vs. upper address discriminator is bit 55, the last bit before the MTE tag, when one is present.

Combining these two criteria, data vs. instructions, lower vs. upper address, gives four distinct address spaces, as illustrated in Figure 11 above.

3.2.12. PAC format and location in pointers

The size and location of the PAC authentication code in a pointer is configurable. This configuration is stored in the special system register TCR_EL1, the Translation Control Register.

TCR_EL1 configures the PAC format at EL0 (user) and EL1 (kernel). The registers TCR_EL2 and TCR_EL3 serve the same purpose at EL2 (hypervisor) and EL3 (monitor), respectively.

The configuration in TCR_ELx specifies four different formats for the PAC, one per address space (see the previous section).

The format of the PAC depends on:

- Do we reserve the most significant byte, typically for MTE?
- Bit 55 is always reserved.
- In the 55 remaining bits, how many do we reserve for the PAC and the real address value?

The last criterion largely depends on how many bits we need for the virtual addresses. Most of the time, 47 or 48 bits are used for the virtual address, which leaves 8 or 7 bits, respectively, for the PAC.

This is illustrated in Figure 12 below. It must be noted that the PAC field is not necessarily contiguous. It can span before and after but 55.

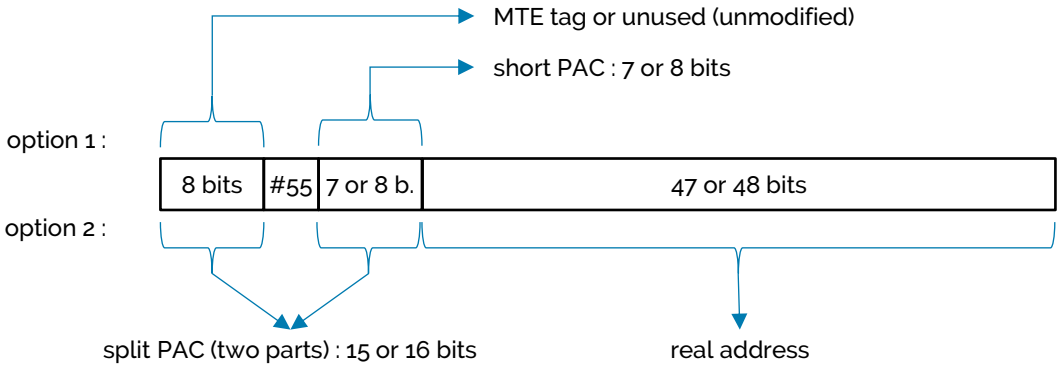

Figure 12: PAC format in a pointer

Bit mask from the cipher output

As mentioned in section 3.2.9, the PAC is computed using a symmetric cipher algorithm. This algorithm takes the original 64-bit pointer as input and produces a 64-bit output.

Only a few bits of this output value are copied into the authenticated pointer. These bits are copied at the same bit offset as in the output value of the algorithm, as if a mask was applied on unused bits.

PAC entropy in various operating systems

For security reasons, we should maximize the entropy of the PAC, and thus its size. This means that an operating system shall carefully consider if MTE shall be used on the current system and how many bits of virtual address are *really* needed.

However, the observation gives some mysterious results. The Table 8 below lists the size and position of the PAC field in the four address spaces on Linux, macOS and Windows [22].

Table 8: PAC size and position

OS version	Data lower		Data upper		Instructions lower		Instructions upper	
Linux 5.10	54-48	7 b.	63-56, 54-48	15 b.	54-48	7 b.	63-56, 54-48	15 b.
Linux 5.15+	54-48	7 b.	54-48	7 b.	54-48	7 b.	63-56, 54-48	15 b.
macOS 13	54-47	8 b.	63-56, 54-47	16 b.	63-56, 54-47	16 b.	63-56, 54-47	16 b.
Windows 11	63-56, 54-47	16 b.	63-56, 54-47	16 b.	63-56, 54-47	16 b.	63-56, 54-47	16 b.

These observations raise a few questions, currently without a satisfactory answer. It is unclear whether these configurations were adopted for good reasons or not.

- 7 or 8 bits is a very small size for the PAC. The entropy is low. Such a configuration should be adopted only as the result of a compromise with other constraints.
- Why do most operating systems use distinct configurations in the four address spaces?
- Why did the Linux kernel reduce the entropy of the PAC in version 5.15 (or immediately before)?
- Why use 48 bits for the virtual address on Linux, vs. 47 bits on macOS and Windows? 47 bits address 128 TB of memory. Isn't it enough for Linux?
- Why ignore the most significant byte in some configuration? On all observed systems, MTE was not implemented in the CPU core. Not using the most significant byte is a waste of PAC entropy.
- Even if MTE is supported on the CPU core, actually using it shall be configurable. Using MTE is described by Arm as a debug tool. It shouldn't be active by default.

If there is no strong reason to adopt the observed configurations, there is some room for improvement in the various operating systems.

3.2.13. Signature diversity

As seen in section 3.2.10, signing a pointer uses a modifier in addition to the key. The value of the modifier is directly computed in the generated code. This must be a value which is specific to a given usage of a pointer.

The authentication instruction must use the same modifier to validate the pointer.

The modifier is used to mitigate the reuse of signed pointers. An attacker may grab a signed pointer of something at some point and reuse it in some other context [16]. If the signature was solely based on the PAC key, this attack would work. However, when modifiers are carefully selected, reusing a signed pointer in another context won't work because the new context expects a different modifier value.

Using carefully selected modifiers is called signature diversity. This is well explained in [15].

There are two main classes of diversity: address diversity and semantic diversity.

Address diversity

The main purpose of pointer authentication is to protect pointers from buffer overflow in memory. Thus, the pointers are signed before being stored in memory. The modifier being a 64-bit value, address diversity means using the storage address of the pointer as modifier (this is the address where the pointer is stored in memory, not the pointer value itself).

Each time a pointer is stored somewhere, that storage address is used as modifier to compute the PAC. Each time a pointer is read from somewhere, that storage address is used as modifier to authenticate the pointer. If a signed pointer was copied from one memory location to another, its signature is no longer valid.

Example 1: In C++, the first element in an object instance is the address of the vtable of the class of that object (if this is a virtual class). A possible attack consists in hijacking the vtable address of a privileged object and copying it into the instance of a less privileged object. If the offsets in the vtable match (for instance when the two classes have a common superclass), the hacked object gains higher privileges.

On macOS, the pointer to the vtable is signed with address diversity. This means that the modifier is the address of the object instance. When a vtable address is copied from one object instance to another, its signature is no longer valid.

Example 2: During a function call, the returned pointer is stored on stack. The compiler always uses the value of SP, the stack pointer, to sign and authenticate the pointer. Different calls from the same point (same return address) occur in different states of the stack and generate different signatures. Similarly, performing a successful ROP attack would need to sign all injected code addresses with their respective storage address on stack, which is impossible to anticipate.

To be precise, unlike the first example, this is not *exactly* address diversity because the SP value which is used as modifier is the value of the stack pointer *before* pushing the return address on stack. Thus, this is not exactly the storage address of the return pointer. However, this is still a context-dependent stack state, and this is what really matters.

Semantic diversity

Semantic diversity goes one step further.

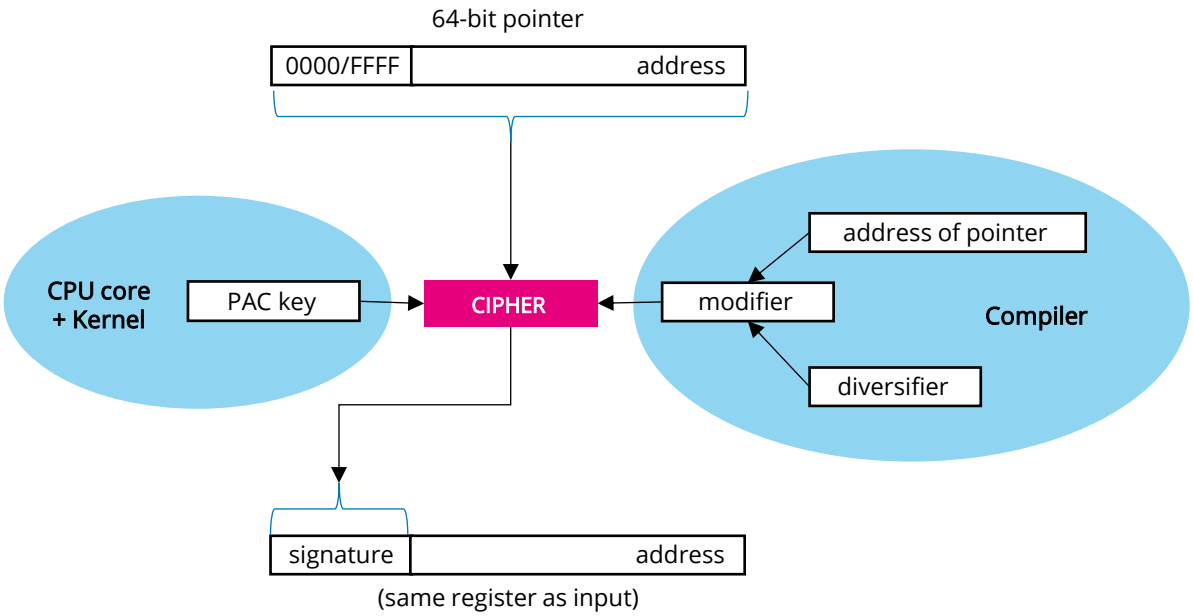

Figure 13: PAC semantic diversity

It starts with address diversity, meaning using the 64-bit storage address of the pointer as modifier.

However, some parts of the modifier (typically the 16 upper bits) are replaced with a value which depends on the semantics of the pointed code or data. This smaller part (16 bits) is called a *diversifier*.

This is illustrated on Figure 13 above.

Let's go back to the first example of address diversity above, the vtable address in a C++ object on macOS. The reality is more complex. A 16-bit diversifier is computed from a hash of the fully qualifier class name of the object. This diversifier is added to the address storage of the pointer and the 64-bit result is used as PAC modifier.

> PAC with semantic diversity is an evolution of the *cryptographically enforced control flow integrity* which was proposed in 2014 by Dan Boneh et al [19]. At the time, they proposed a manual form of PAC using Intel AES-NI instructions.

An example of code generation is provided in section 3.2.15.

Compiler vs. kernel responsibilities

The Figure 13 also illustrates the various responsibilities.

The key is managed by the kernel and remains secret inside the CPU core and the kernel.

The modifier, on the other hand, is under the sole responsibility of the compiler, because it depends on the generated code (address diversity) but also on names from the source code (semantic diversity).

Type-safe authenticated pointers

Semantic diversity propagates semantic information from the source code to the pointer authentication.

In the case of the vtable address, if there is a mismatch during the link, or if sophisticated attack succeeds in having a code pointer signed in place (to pass address diversity), the semantic diversifier enforces the type of the signed pointer.

The sample generated code in 3.2.15 shows one step further. In addition to the address of the vtable being signed in each object instance, all virtual function pointers inside the vtable are individually signed with address diversity using the fully qualified named of the virtual function. Thus, the authentication succeeds only if the code pointer is exactly the expected virtual function, on a source code semantic perspective.

3.2.14. PACGA, a generic CMAC engine

The PACGA instruction is a specific form of PAC. In theory, this is not even "pointer authentication". In practice, we can call it an opportunistic use of the PAC encryption engine for CMAC.

A different kind of input and output

All other PACxx instructions *modify* a 64-bit value, adding an authentication code to this value, in the same register. They also use the fact that this 64-bit value is not *any* value. It is a pointer, a virtual address with unused (or predictable) upper bits which can be replaced.

PACGA, on the other hand, takes *any* 64-bit value as input, applies the PAC cipher algorithm on it, and produces a 32-bit output value in a distinct output register. This is illustrated by Figure 14 below.

One single generic key

The encryption key which is used in this process is the fifth key, the "generic key", or GA key, in register APGAKey_L1. This key is used only by the PACGA instruction.

Also note that there is no GB key.

A one-way operation

Unlike the other PACxx instructions, there is no corresponding AUTxx instruction. PACGA is a one-way instruction. For this reason, PACGA and the "generic key" cannot be used for pointer authentication.

So, what is the use of PACGA?

This is a generic CMAC engine. The main use case is checking the integrity of internal data. Some application routine typically chains PACGA instructions over a large memory area, by chunks of 64 bits.

The chaining is done using an accumulation of the 32-bit results. The accumulation can be as easy as XOR'ing all intermediate 32-bit results. Preferably, it can be another PACGA of the 64 bits which are formed from the current 32-bit output from PACGA on the current 64-bit data and the 32-bit result from the previous iteration.

This mechanism is a fast CMAC operation which detects data corruption, for instance from some buffer overflow issue.

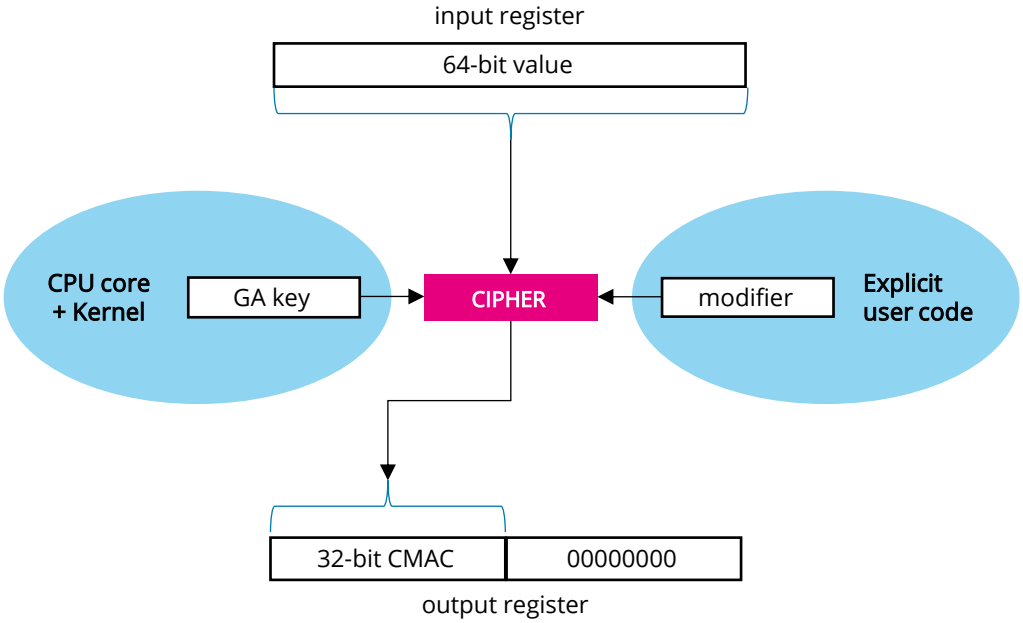

Figure 14: PACGA processing

Note: This mechanism can only detect *internal* memory corruption inside one application process. It is not suitable to compute a CMAC over external storage. The GA key, like all PAC keys, is randomly allocated per process by the operating system. It is ephemeral, per process, and secret from userland. Only the process instance which computed the CMAC can verify it later.

3.2.15. Compiler support

As already seen with BTI, the gcc and clang compilers support a common option named `-mbranch-protection`. To build a module with authenticated return addresses in function calls, use `-mbranch-protection=pac-ret` (this also included in `-mbranch-protection=standard`).

This is pretty much all that can be done on Linux. This is limited to return address, for basic ROP mitigation, without any JOP mitigation, basic address diversity without semantic diversity.

On macOS, the Apple fork of clang implements all the features which are missing on Linux.

Linux ABI (gcc, clang)

The sample code below illustrates the prolog and epilog of a function, with return address authentication.

We can see that the key IA is used to authenticate the return address. Using the option `+b-key` (`-mbranch-protection=pac-ret+b-key`), the IB key is used.

The basic `pac-ret` option protects the return address in *non-leaf* functions. These functions call other functions and, thus, must store their return address on stack. This is where the return pointer is at risk and must be signed.

Using `-mbranch-protection=pac-ret+leaf`, it is possible to protect *leaf* function as well. These functions do not call any other functions and, most of the time, the return address remains in register X30 without ever being stored in memory. There is no risk of being altered by a buffer overflow. However, the last part of a leaf function can still be used as gadget in a chain of attack. Having no authentication of X30 before returning could introduce some risks. This is where the `+leaf` option can help.

The code below illustrates the differences:

Source code

```
// non-leaf function
int func1()
{
    func0();
    return 0;
}

// leaf function
int func2()
{
    return 0;
}
```

gcc -mbranch-protection=pac-ret

```
func1:
  paciasp
  stp     x29, x30, [sp, -16]!
  mov     x29, sp
  bl      func0
  mov     w0, 0
  ldp     x29, x30, [sp], 16
  autiasp
  ret

func2:
  mov     w0, 0
  ret
```

-mbranch-protection=pac-ret+leaf

```
func1:
  paciasp
  stp     x29, x30, [sp, -16]!
  mov     x29, sp
  bl      func0
  mov     w0, 0
  ldp     x29, x30, [sp], 16
  autiasp
  ret

func2:
  paciasp
  mov     w0, 0
  autiasp
  ret
```

Note that this code is fully backward compatible because it uses PAC instructions in the HINT range only.

When compiling for Armv8.3-A and higher, the two instructions AUTIASP and RET are combined into a single RETAA. The code is no longer backward compatible because the RETAA instruction is not known to older CPU cores. The code is slightly more compact but not much: only one instruction less per function.

gcc -mbranch-protection=pac-ret -march=armv8.3-a

```
func1:
  paciasp
  stp     x29, x30, [sp, -16]!
  mov     x29, sp
  bl      func0
  mov     w0, 0
  ldp     x29, x30, [sp], 16
  retaa
```

This is basically all what can be done with PAC on Linux.

macOS ABI (clang)

On macOS, the situation is different. The compiler is a customized version of clang, available as a public fork of the LLVM project [14].

Apple defines two ABI's or architectures, named arm64 and arm64e (with a trailing 'e'). These architectures are used on macOS, iOS and maybe other variants such as tvOS and watchOS. The arm64 architecture is the standard compilation model for 64-bit Arm A-profile. The arm64e architecture adds complete support for PAC.

On iOS, arm64e is the default architecture. On macOS, the default is still arm64 and arm64e is in "preview mode" [23]. It is expected to become the default soon.

Let's focus on macOS in this paper.

The arm64e architecture is selected in one single compilation option `-arch arm64e`. PAC support in arm64e includes:

- ROP mitigation on stack overflow (same as Linux).
- JOP mitigation on buffer overflow: protection of all code and data pointers, function pointer, goto label addresses, C++ vtables, data structure pointers.

The protection of return addresses on stack is similar to the Linux use case, as seen below.

Source code

```
int func1()
{
    func0();
    return 0;
}
```

clang –arch arm64e

```
_func1:
  pacibsp
  stp     x29, x30, [sp, #-16]!
  mov     x29, sp
  bl      _func0
  mov     w0, #0
  ldp     x29, x30, [sp], #16
  retab
```

We can observe two specificities here:

- The PAC key IB is used instead of IA on Linux. On macOS, the key IA is used to protect function pointers. In other words, the arm64e architecture uses key IA for JOP mitigation and key IB for ROP mitigation.
- The combined instruction RETAB is used. On Mac systems, the oldest Arm-based processor is the Apple M1 which implements Armv8.5-A[6]. Consequently, there is no need for pre-Armv8.3-A CPU cores and PAC combined instructions are always used.

However, the most interesting part is the JOP mitigation. All data and code pointers are authenticated using semantic diversity, as explained in section 3.2.13. The two types of authentication are illustrated in a simple example, a call to a C++ virtual function.

```
class A
{
public:
    virtual void f();
    virtual void g();
};
```

[6] With some restrictions in the M1, see [23]. The Apple M2 chip seems to implement all Armv8.5-A mandatory features.

```
int func1(A& a)
{
    a.g();
    return 0;
}
```

In the function `func1()`, the object `a` can be of any subclass of the virtual class `A`. Consequently, calling `a.g()` shall be done through a lookup in the vtable of the actual class of the object.

The generated code for arm64e is listed below. The virtual function call sequence is isolated between horizontal lines.

```
_func1:                         ; x0 = first arg = address of object a
  pacibsp                       ; sign return address (key IB) with SP as modifier
  stp   x29, x30, [sp, #-16]!   ; save stack frame
  mov   x29, sp                 ; x29 = current stack frame
-------------------------------------------------------------------------------
  ldr   x16, [x0]               ; x16 = signed pointer to vtable of object's class
  mov   x17, x0                 ; x17 = address of signed pointer to vtable inside object
  movk  x17, #62866, lsl #48    ; add diversifier in upper 16 bits of x17
  autda x16, x17                ; authenticate (key DA) pointer to vtable
                                ; with x17 as modifier (semantic diversity)
                                ; and then x16 = plain pointer to vtable
  ldr   x8, [x16, #8]!          ; x8 = signed pointer to virtual function g() from vtable
  mov   x9, x16                 ; [*] x9 = address inside vtable of signed pointer to g()
  mov   x17, x9                 ; [*] x17 = same
  movk  x17, #19402, lsl #48    ; add diversifier in upper 16 bits of x17
  blraa x8, x17                 ; authenticate (key IA) pointer to g() and call it
                                ; with x17 as modifier (semantic diversity)
-------------------------------------------------------------------------------
  mov   w0, #0                  ; return 0
  ldp   x29, x30, [sp], #16     ; restore previous stack frame
  retab                         ; authenticated return to caller with SP as modifier
```

We observe the use of semantic diversity on all pointers, data pointers (address of vtable) and code pointers (address of virtual functions).

- The pointer to the vtable which is stored in the object instance is signed using a modifier which is the combination of the address where this pointer is stored (address diversity) and the value #62866 which is likely a hash of the fully qualified class name (semantic diversity).
- The pointer to the virtual function `g()` which is stored in the vtable at offset 8 is signed using a modifier which is the combination of the address where this pointer is stored (address diversity) and the value #19402 which is likely a hash of the fully qualified name of the virtual function (semantic diversity).

Let's add a remark about code optimization. The virtual call sequence uses 9 instructions with option `-O2`. However, the two instructions which are marked with `[*]` are redundant and could be avoided. With proper optimization, the virtual call sequence could use 7 instructions. A bug report has been filed about this[7].

[7] https://github.com/apple/llvm-project/issues/6307

Comparison with unauthenticated virtual calls

Without any form or pointer authentication, the generated code for arm64 is the following:

```
_func1:                         ; x0 = first arg = address of object a
  stp   x29, x30, [sp, #-16]! ; save stack frame
  mov   x29, sp                ; x29 = current stack frame
-------------------------------------------------------------------
  ldr   x8, [x0]               ; x8 = pointer to vtable
  ldr   x8, [x8, #8]           ; x8 = pointer to virtual function g()
  blr   x8                     ; unauthenticated call to g()
-------------------------------------------------------------------
  mov   w0, #0                 ; return 0
  ldp   x29, x30, [sp], #16    ; restore previous stack frame
  ret                          ; unauthenticated return to caller
```

We can see here that the virtual call sequence uses 3 instructions instead of the 7 instructions of a properly optimized sequence with semantic diversity on all pointers.

Other forms of authenticated virtual calls

The code pattern which is suggested by Arm to call an authenticated virtual method is the following (the temporary registers x8 and x9 are arbitrary choices here).

```
  ldr   x8, [x0]      ; x8 = signed pointer to vtable of object's class
  ldraa x9, [x8, #8]! ; authenticate vtable address with key DA and zero as modifier
                      ; x9 = signed pointer to virtual function g()
                      ; x8 = plain address inside vtable of signed pointer to g()
  blraa x9, x8        ; authenticate (key IA) pointer to g() and call it
                      ; with x8 as modifier (address diversity)
```

It is interesting to note that such an authenticated virtual call sequence uses the exact same number of instructions as a non-authenticated virtual call.

However, the security of such a sequence is not as good as the one for arm64e. First, the data pointer to the vtable is not diversified at all. It uses zero as modifier, which is the price to pay for using the instruction LDRAA. Second, the pointer to the virtual function is authenticated with address diversity only, not semantic diversity.

Consequently, the 3-instruction authenticated virtual call sequence remains vulnerable to complex reuse attacks as described in [16].

The 7-instruction authenticated virtual call sequence with double semantic diversity is probably the best usage of Arm pointer authentication code so far.

3.2.16. Code generation influence on ROP : gcc vs. clang

The PAC feature protects the code against stack overflows trashing the stack frame. However, which stack frame is trashed, exactly?

> The *stack frame* is the structure which is saved on stack by each function to link with the caller's function. At any time, the register X29 points to the stack frame of the currently executing function. On Arm64, the stack frame structure contains only two 64-bit values: the address of the caller's stack frame (previous value of X29) and the return code address (previous value of X30).
>
> Note that some simple functions which do not call other functions may have stack frame all. We call them *leaf functions*. They do not need to save their own context, X30 permanently points to the return address without risk of being trashed and X29 points to the caller's stack frame.

In practice, gcc and clang generate different layouts for the stack frame of the functions they compile. These details shall be perfectly understood by hackers who craft ROP attacks.

The PAC feature equally protects the two kinds of code generation. However, the mitigation occurs at different times after the stack overflow attack.

Let's have a look at the differences in the generated codes from the two compilers for the prolog and epilog of a function.

Source code

```
void f(...)
{
    char buf[16];
    ...
}
```

gcc 12.2.0

```
f:
  stp  x29, x30, [sp, -64]!
  mov  x29, sp
  ...
  ldp  x29, x30, [sp], 64
  ret
```

clang 15.0.7

```
f:
  sub  sp, sp, #64
  stp  x29, x30, [sp, #48]
  add  x29, sp, #48
  ...
  ldp  x29, x30, [sp, #48]
  add  sp, sp, #64
  ret
```

We can notice the following differences:

- With gcc, the stack frame is *below* the local variables. In case of ROP attack, the stack overflow trashes the caller's stack frame. The attack is triggered at the end of the caller's function.
- With clang, the stack frame is *above* the local variables. In case of ROP attack, the stack overflow trashes the current stack frame. The attack is triggered at the end of the current function.

This is illustrated in Figure 15 below. The input is read into the `buf[]` local array. The red arrow indicates the trashed memory during an overflow of this local variable.

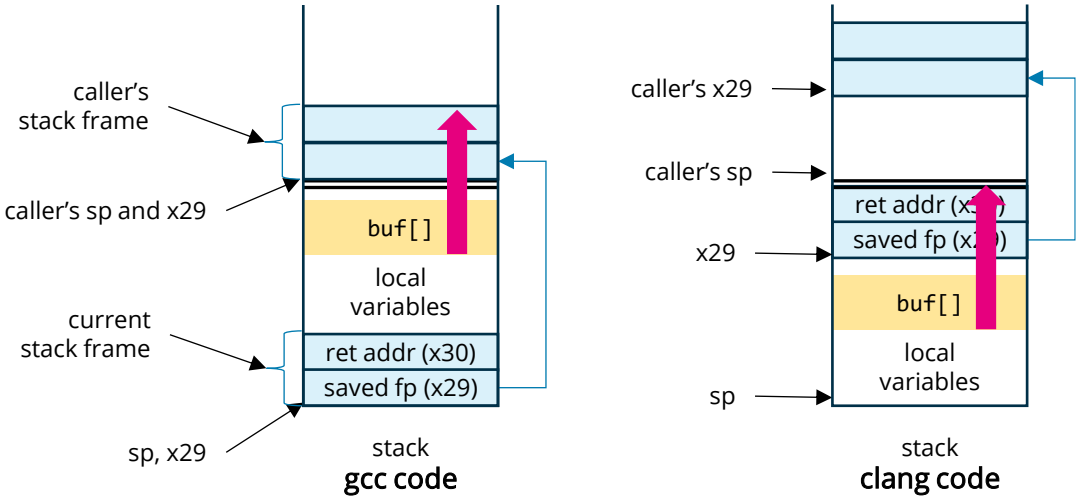

Figure 15: Stack frame layout with gcc and clang

All ROP and JOP attacks target a specific binary and the shared libraries it activates. Depending on the compiler which was used to build the binaries, the hacker must understand the generated code (although most gadget-finder tools will do that automatically).

Furthermore, since the on-stack protection of PAC is local to a function, it is quite possible to link together modules which were compiled by the two compilers. A gcc-compiled function can call a clang-compiled function and vice-versa. The designer of the attack (or his automation tools) must take this into account.

3.2.17. Virtualization support (EL2)

When a hypervisor executes at EL2, it can control the way the virtual machines use PAC. The hypervisor can intercept all PAC instructions and all accesses to PAC key registers.

To receive traps at EL2 when PAC is used in a virtual machine, the hypervisor needs to program a few special system registers. The trap handler in the hypervisor may then control, prevent, or emulate the PAC operation that the virtual machine wanted to execute.

Global control

HCR_EL2 is the Hypervisor Configuration Register. It contains the following configuration bits:

- API: When set to 0, all PAC instructions executing at EL0 or EL1 generate a trap at EL2. This covers all instructions from Table 6, page 25, except the XPACxx instructions which have no security impact.
- APK: When set to 0, any read or write access at EL0 or EL1 to any of the 10 PAC key registers generates a trap at EL2.

Fine-grained control

The fine-grained hypervisor trap registers are used to enable traps at EL2 on selected PAC instructions or selected types of access (read/write) on selected PAC key registers.

- HFGITR_EL2, the Hypervisor Fine-Grained Instruction Trap Register, configures traps on ERETAA and ERETAB instructions (authenticated return from exception).
- HFGRTR_EL2, the Hypervisor Fine-Grained Read Trap Register, configures traps on read access to each of the 5 PAC keys, selectively.
- HFGWTR_EL2, the Hypervisor Fine-Grained Write Trap Register, configures the equivalent traps for write access.

3.2.18. Monitor support (EL3)

EL3 is the highest level or privilege, where the *secure monitor* software executes. In most cases, the secure monitor software is part of the Arm Trusted Firmware package [8] but system vendors may also develop and provide their own.

Global control

The secure monitor must have the capability to intercept any action which relates to security, which includes pointer authentication code. Just like the hypervisor at EL2, the monitor configures its control over the system using some special system registers.

SCR_EL3 is the Secure Configuration Register. It contains the following configuration bits:

- API: When set to 0, all PAC instructions executing at EL0, EL1, or EL2 generate a trap at EL3. This covers all instructions from Table 6, page 25, except the XPACxx instructions which have no security impact.
- APK: When set to 0, any read or write access at EL0, EL1, or EL2 to any of the 10 PAC key registers generates a trap at EL3.

These traps are EL3 are generated only when security is at stake for the system. This means that some PAC operations do not trap in specific situations:

- PAC instructions which are “disabled” (see 3.2.19) do not generate a trap.

- If PAC instructions or key access are already trapped at EL2, this means that the hypervisor already handles PAC security operations locally in virtual machines. In that case, these operations do not trap at EL3.

Probable use case: PAC protection on macOS

On macOS, directly reading or writing any PAC key register at EL1 crashes the system.

This has been observed using tools in [20] which provides test kernel modules for Linux, macOS, and Windows. In these modules, there are commands to read and write the PAC key registers. Since these registers are EL1 registers, this is allowed by default and works on Linux and Windows. On macOS, on the other hand, simply trying to read the content of a PAC key register crashes the system.

There are only three possible technical reasons for this, HCR_EL2, HFGxTR_EL2, or SCR_EL3 configurations.

The EL2 registers are typically reserved for a hypervisor. Using PAC in a Linux or Windows virtual machine running on top of macOS works well. It is even possible to access the PAC key registers from the kernel of the guest operating system. So, EL2 registers are not likely the reason for the crash in macOS host system.

The only remaining reason is SCR_EL3.

Speculation: It is probable that macOS delegates the handling of PAC keys to the EL3 monitor or a trusted application running in a TrustZone TEE, through some specific API. This also means that the monitor software running on Mac is specifically crafted by Apple.

Possible application to Linux: PAC key management hardening

Compared to macOS and Windows 11, the Linux kernel is relatively permissive in terms of security. By default, there is no mandatory secure boot, no mandatory authenticated kernel modules. All vulnerabilities are publicly documented as CVE's [10], making unpatched system vulnerable.

Pushing critical security features outside the kernel, to a more protected environment, either the monitor or a trusted application in TEE, could be a significant improvement.

This would require, however, the definition of a standard API with the monitor and, consequently, in the Arm trusted firmware.

3.2.19. Handling software compatibility

The problem with the dispersion of Arm features is the distribution of binaries to heterogeneous systems. The main reason is the existence of instructions which are specific to a given optional feature. The pointer authentication code feature is no exception, with 48 new instructions.

There are two types of compatibility to solve.

The first type is backward compatibility, or how to execute protected binaries on older CPU cores which do not implement PAC.

The second type is interoperability. Compiling with advanced PAC protection means that all or most pointers to code and data are signed. On the other hand, modules which are compiled without PAC use plain pointers. How can software modules from these two types exchange pointer values?

Basic backward compatibility: HINT instructions

We have seen in section 3.2.8 that a few PAC instructions were allocated in the HINT range and act as NOP on older CPU cores. They are backward compatible. All other PAC instructions generate an *illegal instruction* fault on older CPU cores.

However, the backward compatible PAC instructions address limited use cases. In practice, the only supported use cases are function calls and *trampoline* calls through X16/X17.

For more advanced protection (not yet supported on Linux compilers), the only solution is to distribute two flavors of binary packages, one for pre-Armv8.3-A cores, one for more recent cores.

Advanced ABI interoperability: controlled disablement of PAC instructions

When advanced PAC protection is used, software modules exchange signed pointers. They can be function parameters, explicit pointers in data structures, implicit class, and virtual code pointers in object-oriented languages, etc.

Passing these pointers to code which was compiled without PAC support will likely crash the application. The code without PAC support simply dereferences pointers without authenticating them (and removing the signature). A signed pointer is likely not a valid address and the application will crash.

Fortunately, this case was anticipated in the Arm architecture.

However, this must be planned and handled in the operating system.

The solution is in the special system registers SCTLR_EL1, SCTLR_EL2, and SCTLR_EL3, the System Control Registers, per exception level.

In each of these registers, the four bits named EnIA, EnIB, EnDA, EnDB control the usage of the corresponding four PAC keys IA, IB, DA, DB. When one of these bits is set to zero, the corresponding PAC is *disabled*.

Disabling a PAC key means that the PACxx instructions don't compute a signature and don't modify the pointer. AUTxx instructions don't authenticate the pointer. These instructions do nothing, they act as NOP.

When a PAC key is disabled, the combined PAC instructions remain functional. They continue to load data (LDRAA), return to the caller (RETAA), branch somewhere (BLRAA). But no PAC is generated or verified in pointers.

When an application is built from a mix of modules with advanced PAC protection and modules without it, the kernel of the operating system shall detect this and disable PAC keys for that process. Each time the process is scheduled on a CPU core, the kernel shall reconfigure SCTLR_EL1.

If the advanced PAC usage is defined to use certain PAC keys for interchanged data and other PAC keys for local data, then the kernel does not need to disable the locally used keys.

For instance, the arm64e ABI on macOS uses the key IA to protect code pointers in data structures or vtables, and key IB to protect return addresses on stack. When scheduling a process which is a mix of protected and unprotected modules, it is necessary to disable the key IA (the signed pointers are used in all modules) but not IB (the signed return addresses are used in protected modules only).

This is why the SCTLR_ELx registers can selectively disable some keys but not all of them at once. That was cleverly anticipated.

Use case: arm64e vs. arm64 on macOS

This principle of selective key disablement is used on macOS. By default, a module is compiled for architecture arm64, without PAC protection. When a module is compiled for architecture arm64e, the code contains advanced PAC protection for code and data pointers. This was already explained in section 3.2.15.

An executable or shared library file can be built only from object modules of the same architecture, arm64 or arm64e.

All system libraries and the kernel are built for arm64e. They use full PAC protection.

When an application is built for arm64e, it can only be linked against arm64e libraries. If the developer builds his application for arm64e, he expects full protection. Having unprotected arm64 libraries in the process memory space would create security holes. This is the reason for this restriction.

On the other hand, when an application is built for arm64, no PAC protection is expected by this application, and it can be linked against arm64 and arm64e libraries.

The architecture, arm64 or arm64e, is identified in each library and executable file. Whenever the operating system schedules a process with an arm64 application on a CPU core, the SCTLR_EL1 is reconfigured to disable the PAC keys. Thus, the arm64e libraries in the application no longer modify or authenticate the pointers and become fully interoperable with the arm64 application.

The PACGA case

As a final note, the SCTLR_ELx registers do not provide support for disabling the key GA. As described in section 3.2.14, the PACGA instruction is not used to authenticate pointers. This is a generic CMAC engine. Either you explicitly use it, or you don't. But there is no interoperability issue.

3.2.20. Comparative operating system support

Reaching the end of this analysis of pointer authentication features, let's briefly compare the levels of support in the three main operating systems, in decreasing level of support.

macOS

The LLVM fork for macOS implements the most sophisticated usages of PAC when the target is arm64e. The return addresses on stack are protected. All code pointers are protected, C++ vtables are protected at two levels: the code pointers in the vtable and the pointers to the vtable in all object instances. The code and data pointers are protected using advanced semantic diversity to prevent polymorphic attacks (replacing a valid signed pointer with another valid signed pointer, typically executing more dangerous operations).

The kernel and all system libraries are built for arm64e. Applications, on the other hand, are still built for arm64, meaning without PAC protection. This needs to be improved.

The advanced PAC protection in the arm64e architecture requires a specific ABI. This ABI only differs in the way PAC is used. The interoperability of arm64e and arm64 binaries is managed through the proper identification of the architecture of each binary, object file, library, application, and the controlled disablement of PAC instructions in arm64 applications.

New random PAC keys are allocated per process.

All 5 PAC keys are segmented between kernel and user mode.

PAC keys are not accessible in the kernel (probably managed in the monitor or TEE).

Linux

The gcc and clang compilers for Linux can only generate PAC protection for the return addresses on stack (ROP mitigation). There is no protection for code and data pointer (no JOP mitigation), no extended ABI, no semantic diversity. The only option is `-mbranch-protection=pac-ret`.

The kernel is compiled with PAC protection in recent versions of some distributions (Ubuntu stock kernel for instance).

New random PAC keys are allocated per process.

Only the IA key is segmented between kernel and user mode. The 4 other PAC keys are identical.

PAC keys are directly managed in the kernel.

Windows

Windows support for Arm64 is still in infancy.

As of mid-2023, it is still not possible to download a complete Arm64 Windows 11 system binary. Windows 11 for Arm64 is typically bundled with specific devices only, the Surface tablets for instance.

The PAC keys are not managed by the kernel. All PAC keys have value 00000000-00000000.

It is possible to read and write the PAC key registers from a kernel module. However, without proper support from the process management in the kernel, this is useless to protect applications.

There is little doubt that PAC support and application protection will be implemented in Windows 11. However, it is unclear when this will happen.

4. Further developments

Hopefully, the detailed descriptions in this paper have convinced the reader of the power of the Control Flow Integrity features in the Arm64 architecture.

However, we have also seen that this power is not fully exploited in the various software environments.

The following list enumerates the topics where there is room for improvement. We strongly encourage the software vendors and open-source community to embrace these tasks to make all software environments even more secure and resistant to malware code injection.

- On gcc and clang, define a complete PAC coverage strategy. The currently simple “pac-ret” implementation is notoriously insufficient. Add JOP mitigations using PAC on code and data pointers. The PAC implementation in the Apple fork of LLVM is the way to go.
- On Linux, manage backward compatibility, instead of avoiding it. Limiting the PAC implementation to “pac-ret” is probably a good way to avoid compatibility issues. The apparent compatibility issues of a more complete ABI with signed pointers virtually everywhere can be easily solved with the identification of secure executables and libraries (improve the compilers and binutils), combined with “disabled” PAC keys in non-secure processes (improve the kernel). The macOS “arm64e” concept is the way to go.
- On macOS, make “arch64e” the default for all applications.
- On Windows, fully support PAC, define a system-wide ABI for PAC, allocate and manage PAC keys in the kernel. It seems that everything still needs to be done to fully support PAC+BTI.

Appendix: Relevant CPU system registers

The general system control registers are documented in the Arm Architecture Reference Manual [1], in section D19.2 (as of version J.a, April 2023).

The Table 9 below lists a selected set of system registers which are relevant to PAC and BTI. The “sections” column refers to the sections in this white paper where the use of the corresponding system register is explained.

The ID_AA64xxx registers are used to check the support for the various Arm features for PAC and BTI.

Table 9: Relevant CPU system registers for PAC and BTI

Register name	Full name	See sections
APDAKeyHi_EL1	Pointer Authentication Key A for Data (bits[127:64])	3.2.10
APDAKeyLo_EL1	Pointer Authentication Key A for Data (bits[63:0])	3.2.10
APDBKeyHi_EL1	Pointer Authentication Key B for Data (bits[127:64])	3.2.10
APDBKeyLo_EL1	Pointer Authentication Key B for Data (bits[63:0])	3.2.10
APGAKeyHi_EL1	Pointer Authentication Generic Key (bits[127:64])	3.2.10, 3.2.14
APGAKeyLo_EL1	Pointer Authentication Generic Key (bits[63:0])	3.2.10, 3.2.14
APIAKeyHi_EL1	Pointer Authentication Key A for Instructions (bits[127:64])	3.2.10
APIAKeyLo_EL1	Pointer Authentication Key A for Instructions (bits[63:0])	3.2.10
APIBKeyHi_EL1	Pointer Authentication Key B for Instructions (bits[127:64])	3.2.10
APIBKeyLo_EL1	Pointer Authentication Key B for Instructions (bits[63:0])	3.2.10
HCR_EL2	Hypervisor Configuration Register	3.2.17
HFGITR_EL2	Hypervisor Fine-Grained Instruction Trap Register	3.2.17
HFGRTR_EL2	Hypervisor Fine-Grained Read Trap Register	3.2.17
HFGWTR_EL2	Hypervisor Fine-Grained Write Trap Register	3.2.17
ID_AA64ISAR1_EL1	AArch64 Instruction Set Attribute Register 1	2.4.2, [20]
ID_AA64ISAR2_EL1	AArch64 Instruction Set Attribute Register 2	2.4.2, [20]
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1	2.4.2, [20]
SCR_EL3	Secure Configuration Register	3.2.18
SCTLR_EL1	System Control Register (EL1)	3.2.19
SCTLR_EL2	System Control Register (EL2)	3.2.19
SCTLR_EL3	System Control Register (EL3)	3.2.19
TCR_EL1	Translation Control Register (EL1)	3.2.12
TCR_EL2	Translation Control Register (EL2)	3.2.12
TCR_EL3	Translation Control Register (EL3)	3.2.12

Viewing or manipulating these registers for experimentation can be done using tools in the *arm-cpusysregs* project [20]. These tools were developed to write this white paper. Be aware that using them can be dangerous for the stability of the system and shall be reserved for experimentation on test systems.

Acronyms

ABI	Application Binary Interface
ANSSI	Agence Nationale de la Sécurité des Systèmes d'Information (France)
ASLR	Address Space Layout Randomization
AV	Anti-Virus
BTI	Branch Target Identification
CCA	Confidential Compute Architecture
CFI	Control Flow Integrity
CISA	Cybersecurity & Infrastructure Security Agency (USA)
CMAC	Cipher-based Message Authentication Code
CVE	Common Vulnerabilities and Exposures [10]
ELx	Exception Level 0, 1, 2, or 3
FP	Frame Pointer
HPC	High-Performance Computing
IDS	Intrusion Detection System
IP0/IP1	Intermediate Pointer 0 or 1
IPS	Intrusion Prevention System
ISA	Instruction Set Architecture
IT	Information Technology
JOP	Jump-Oriented Programming
LLVM	The infrastructure behind the clang compiler, no longer an acronym
OS	Operating System
PAC	Pointer Authentication Code
PC	Program Counter
ROP	Return-Oriented Programming
SIEM	Security Information and Event Management
SP	Stack Pointer
TEE	Trusted Execution Environment (a lightweight operating system in TrustZone)
ZTA	Zero-Trust Architecture

References

[1] *Arm Architecture Reference Manual for A-profile architecture*
https://developer.arm.com/documentation/ddi0487/

[2] *Applying PAC+BTI techniques to real code* – Arm developer
https://developer.arm.com/documentation/102433/0100/Applying-these-techniques-to-real-code

[3] *Arm Neoverse V1 Software Optimization Guide* – Arm developer
https://developer.arm.com/documentation/PJDOC-466751330-9685/latest/

[4] *Arm Neoverse V2 Core Software Optimization Guide* – Arm developer
https://developer.arm.com/documentation/PJDOC-466751330-593177/latest/

[5] *Arm Neoverse N2 Software Optimization Guide* – Arm developer
https://developer.arm.com/documentation/PJDOC-466751330-18256/latest/

[6] *TrustZone for Cortex-A* – Arm
https://www.arm.com/technologies/trustzone-for-cortex-a

[7] *Confidential Compute Architecture* – Arm
https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[8] *Arm Trusted Firmware* – Open-source secure software for Arm systems
https://www.trustedfirmware.org/

[9] *Trusted Firmware Source code* – Project arm-trusted-firmware
https://github.com/ARM-software/arm-trusted-firmware

[10] *Common vulnerabilities and exposures database*
https://www.cve.org/ (formerly https://cve.mitre.org/)

[11] *Pointer authentication on Armv8.3, design and analysis of the new software security instructions* – Qualcomm Technologies Inc.
https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf

[12] *The QARMA block cipher family* – Roberto Avanzi, Qualcomm Product Security
https://eprint.iacr.org/2016/444.pdf

[13] *Learning QARMA64* – Reference implementation of QARMA64
https://github.com/Phantom1003/QARMA64

[14] *Apple fork of LLVM project*
https://github.com/apple/llvm-project

[15] *Pointer Authentication implementation* – Apple fork of LLVM project
https://github.com/apple/llvm-project/blob/apple/main/clang/docs/PointerAuthentication.rst

[16] *PAC it up: Towards Pointer Integrity using ARM Pointer Authentication* - Hans Liljestrand, Carlos Chinea Perez, Thomas Nyman, Jan-Erik Ekberg, Kui Wang, N. Asokan
https://www.usenix.org/system/files/sec19fall_liljestrand_prepub.pdf

[17] *In-kernel control-flow integrity on commodity OSes using Arm pointer authentication* – Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, Taesoo Kim – Samsung Research, Georgia Institute of Technology
https://arxiv.org/pdf/2112.07213.pdf

[18] *Capability Hardware Enhanced RISC Instructions (CHERI)* – University of Cambridge
https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

[19] *Cryptographically Enforced Control Flow Integrity* – Ali Jose Mashtizadeh, Andrea Bittau, David Mazieres, Dan Boneh – Stanford University (2014)
https://arxiv.org/abs/1408.1451

[20] *Tools to explore Arm64 CPU system registers and PAC* – Project arm-cpusysregs
https://github.com/lelegard/arm-cpusysregs

[21] *Arm features for A-profile* – Project arm-cpusysregs
https://github.com/lelegard/arm-cpusysregs/blob/main/docs/features.md

[22] *Pointer authentication code format* – Project arm-cpusysregs
https://github.com/lelegard/arm-cpusysregs/blob/main/docs/pac-format.md

[23] *Apple architectures arm64 and arm64e on macOS* – Project arm-cpusysregs
https://github.com/lelegard/arm-cpusysregs/blob/main/docs/arm64e-on-macos.md

[24] *US cybersecurity director: The tech ecosystem has 'become really unsafe'* – Yahoo! finance, 05/01/2023
https://finance.yahoo.com/news/us-cybersecurity-director-the-tech-ecosystem-has-become-really-unsafe-222118097.html

[25] *Cybersécurité : "On fait tout pour éviter" des cyberattaques sur le réseau électrique", assure le directeur de l'ANSSI* (in French) – France info, 13/12/2022
https://www.francetvinfo.fr/internet/securite-sur-internet/cyberattaques/cybersecurite-on-fait-tout-pour-eviter-des-cyberattaques-sur-le-reseau-electrique-assure-le-directeur-de-l-anssi_5541561.html

[26] *Morris worm, one of the oldest computer worms* – Wikipedia
https://en.wikipedia.org/wiki/Morris_worm

[27] *Log4shell, a zero-day vulnerability in Log4j* – Wikipedia
https://en.wikipedia.org/wiki/Log4Shell

[28] *Message authentication code* – Wikipedia
https://en.wikipedia.org/wiki/Message_authentication_code

[29] *Complete Guide to Stack Buffer Overflow (OSCP Preparation)* – Stefano Lanaro, 2021
https://steflan-security.com/complete-guide-to-stack-buffer-overflow-oscp/

[30] *Smashing The Stack for Fun and Profit* – Aleph One
https://insecure.org/stf/smashstack.html

[31] *Buffer-Overflow Vulnerabilities and Attacks* – Syracuse University
https://web.ecs.syr.edu/~wedu/Teaching/IntrCompSec/LectureNotes_New/Buffer_Overflow.pdf

[32] *A Beginner's Guide to Buffer Overflow* - Raj Chandel, 2021, Hacking Articles
https://www.hackingarticles.in/a-beginners-guide-to-buffer-overflow/

Maisons-Laffitte

22 rue Guynemer
78600 Maisons-Laffitte, France
+33 1 80 83 54 90

Barcelona

Office 201-202, Parc UPC-RDIT
Parc Mediterrani de la Tecnologia
Esteve Terradas, 1
08860 Castelldefels, Barcelone, Espagne

Duisbourg

Schifferstraße 196
47059 Duisbourg, Allemagne

Grenoble

4 Place Robert Schuman
38000 Grenoble, France

Massy

Immeuble Carnot Plaza
14 avenue Carnot
91300 Massy, France

Sophia Antipolis

Les Aqueducs B3
535 route des Lucioles
06560 Valbonne, France

SIPEARL

SiPearl is building the first energy-efficient HPC-dedicated micro-processor designed to work with any third-party accelerator (GPU, AI, quantum). It will help Europe solving major challenges in medical research, artificial intelligence, security, energy management and climate while reducing its environmental footprint.

European Processor Initiative

Funded by the European Union