Qualcomm Technologies, Inc.

Pointer Authentication on ARMv8.3

Design and Analysis of the New Software Security Instructions

Qualcomm Product Security
Qualcomm Technologies, Inc.
January 2017

Qualcomm Technologies, Inc.

Qualcomm Technologies, Inc.

5775 Morehouse Drive

San Diego, CA 92121

U.S.A.

Table of Contents

Introduction

ARM recently announced ARMv8.3-A, the 2016 additions to the ARMv8-A architecture [1]. These additions include Pointer Authentication instructions: "a mechanism for enhanced security associated with pointer authentication". It is very exciting to see the technology, refined and expanded through discussions and contributions from ARM and their partners, making it into the architecture as the new Pointer Authentication instructions.

The pointer authentication scheme introduced by ARM is a software security primitive that makes it much harder for an attacker to modify protected pointers in memory without being detected. In this document, we will provide more details about the Pointer Authentication mechanism, provide a security analysis, and discuss the implementation of certain software security countermeasures, such as stack protection and control flow integrity, using the Pointer Authentication primitives.

Problem Definition

A common problem in software security is memory corruption vulnerabilities such as buffer overflows. These vulnerabilities are often exploited by overwriting control data in memory (function pointers and return addresses) to redirect code execution to a location controlled by the attacker. There are three common ways to defend against memory corruption exploits:

1. Prevent corruption by putting sensitive data and pointers into read-only memory: This works very well for static tables of function pointers and other sensitive data. One still needs to make sure that any pointers to these read-only tables are also verified or protected. Unfortunately, it does not work for dynamic pointers such as return addresses on the stack or dynamically allocated objects containing function pointers.
2. Detect corruption by verifying the pointers before using them: This is how software stack protection (SSP) works. Control Flow Integrity (CFI) and other Return Oriented Programming (ROP) mitigations such as checking various properties of the target of jumps/returns also fall into this category.
3. Make it harder to find a target: This is achieved by randomization of some form. Randomization is a good generic defense that makes it harder to reliably exploit systems. Countermeasures in this category range from randomized stack/heap, which makes it harder to find a pointer to corrupt, to full Address Space Layout Randomization (ASLR), which makes it harder to identify where to jump to. Note that some detection countermeasures like stack protection also require unpredictability (i.e. random stack canaries) to be effective.

These techniques are complementary and most modern countermeasure designs use a combination of them.

Part of our product security work at Qualcomm Technologies, Inc. is introducing software security countermeasures into our platforms. This covers not only the applications processor that runs the main operating system, but also images for bootloaders, peripherals such as modem, WiFi and DSP, and other execution environments like hypervisors and TrustZone. Most of these images already support three baseline countermeasures: software stack protection, Data Execution Prevention (DEP/W^X), and a hardened heap. However, these images also have size and performance constraints that make more advanced countermeasures such as ASLR and software-based CFI infeasible to implement. We wanted a scheme where we could check the validity of pointers with minimal size and performance impact, and that resists memory disclosure vulnerabilities. This ruled out using XOR with a random value and other simple ways to obfuscate or scramble pointers. We needed a cryptographically strong algorithm.

Using authentication instead of encryption was an early design decision. With authentication, the actual pointer value is still available without having to know the secret keys. This has many advantages, from allowing branch prediction in the processor to debugging. Furthermore, with authentication, it becomes possible to know a when corruption happened rather than just jumping to a random location and hopefully crashing. The next section describes the design of Pointer Authentication.

Pointer Authentication

The basic idea behind Pointer Authentication is that the actual address space in 64-bit architectures is less than 64-bits. There are unused bits in pointer values that we can use to place a Pointer Authentication Code (PAC) for this pointer. We could insert a PAC into each pointer we want to protect before writing it to memory, and verify its integrity before using it. An attacker who wants to modify a protected pointer would have to find/guess the correct PAC to be able to control the program flow.

Not every pointer has the same purpose in a program. We want the pointers to be valid only in a specific context. In Pointer Authentication, this is achieved in two ways: Having separate keys for major use cases and by computing the PAC over both the pointer and a 64-bit context. The pointer authentication specification defines five keys: two for instruction pointers, two for data pointers and one for a separate general-purpose instruction for computing a MAC over longer sequences of data. The instruction encoding determines which key to use. The context is useful for isolating different types of pointers used with the same key. The context is specified as an additional argument together with the pointer when computing and verifying the PAC.

The pointer authentication design has three main components: instructions, cryptography, and key management which will be described below.

Instructions

There are two main operations needed for Pointer Authentication: computing and adding a PAC, and verifying a PAC and restoring the pointer value. These are handled by the PAC* and AUT* sets of instructions respectively. If verification fails during the AUT instruction, the processor replaces the PAC with a specific pattern that makes the pointer value an illegal address. The actual error detection happens through the illegal address exception when an invalid pointer is dereferenced. This design decouples error handling from the instruction and removes the need to use additional instructions for error handling. The exception handler can distinguish between an illegal address exception and an authentication failure by checking the pattern that the AUT instruction uses to signal the error.

In addition to the PAC and AUT instructions, there are instructions for stripping the PAC (XPAC*), and for computing a 32-bit authentication code from two 64-bit inputs (PACGA). The PACGA instruction is useful for protecting sensitive data structures in memory. For instance, it can be used to compute an authentication code (AC) value that covers all the heap metadata by chaining together a couple of calls to the PACGA instruction.

The ARMv8.3-A architecture describes variants of the PAC and AUT instructions for each of the four keys and includes combined instructions such as verify-then-return (RETA*) and verify-and-branch (BRA*, BLRA*). A subset of instructions is encoded in NOP space (part of the instruction space that is treated as NOPs in earlier revisions of the architecture). This provides binary backwards compatibility, allowing older ARMv8 processors to run binaries compiled with these new instructions.

Cryptography

Cryptography is a critical part of Pointer Authentication. We need a fast and lightweight algorithm that is also cryptographically strong when truncated for very short tags. All existing candidates for this algorithm, such as SipHash [2] and PRINCE [3], would have to be used in constructions that have too high a latency for authenticating pointers, potentially impacting performance too much. One of the concerns is that the PAC must be a function of a secret string (or key) not controlled by the current environment, the pointer being tagged, and a local context. This would make the SipHash input (and thus processing time) too long, and a block cipher like PRINCE is designed to only admit a 128-bit key and a 64-bit input. One possible way to make the data fit in the PRINCE inputs would be to concatenate a 64-bit secret with the context to obtain the key to encrypt the pointer, and then truncating the ciphertext. However, this would give the attacker the ability to control the PRINCE key somewhat, which would be very dangerous for security because of the structure of PRINCE and its susceptibility to related key attacks.

Thus, we designed QARMA [4], a new family of lightweight tweakable block ciphers. Tweakable means that the permutation computed by the cipher on the plaintext is determined by a secret key and an additional user selectable tweak.

QARMA is targeted to a very specific set of use cases. Besides the case used here, generation of very short tags by truncation, it is also designed to be suitable for memory encryption, and the construction of keyed hash functions. It is meant to be used in fully unrolled hardware implementations. The algorithm belongs to the family of designs inaugurated by PRINCE in that it is a reflection design, and is related to the cipher MANTIS [5], but with a different design and a more conservative choice of S-Boxes than MANTIS to reduce the likelihood of certain classes of cryptanalytic attacks.

When used for Pointer Authentication, two inputs to QARMA are the pointer and the context. The PAC is the truncated output of QARMA. The size of the PAC is determined by processor virtual address size configuration and whether the “tagged address” feature is in use. The “tagged address” feature, which is different from PAC, allows software to add an 8-bit tag to a pointer without affecting address translation. PAC can use these bits if the tagged address feature is not enabled. Since the virtual address size can be configured to be between 32 and 52 bits and one bit (bit 55) is used for selecting the high or low half of the virtual address space, the size of PAC ranges from 11 to 31 bits when tagged addresses are disabled, and from 3 to 23 bits when tagged addresses are enabled. The PACGA instruction always generates a 32-bit AC from the QARMA output.

Key Management

QARMA uses 128-bit keys. The Pointer Authentication specification defines five keys. Four keys for PAC* and AUT* instructions (combination of instruction/data and A/B keys), and a fifth key for use with the general purpose PACGA instruction. The keys are stored in internal registers and are not accessible by EL0 (user mode), but otherwise are not tied to exception levels. The software (EL1, EL2 and EL3) is required to switch keys between exception levels as needed. It is expected that higher privilege levels control the keys for the lower privilege levels.

Control bits for each key are added that define the exception behavior. This allows keys to be generated or switched on demand.

The keys are expected to be ephemeral (per process for EL0 and per boot for EL1 to EL3), and key management, including generating good quality random numbers, is the responsibility of the software.

Sample Use Cases

This section describes different use cases we expect Pointer Authentication to be used for.

Software Stack Protection

Software stack protection is a countermeasure against stack-based buffer overflows. Typically, the compiler instruments selected functions, placing a random canary value between the return address and buffers on the stack when entering the function, and checking the canary for corruption before the function returns. While software stack protection is effective against certain types of buffer overflows, it can be defeated by memory disclosure vulnerabilities. The table below illustrates how the code is instrumented for software stack protection. Changes are highlighted in red:

	No stack protection	Software Stack protection

Function Prologue	`SUB sp, sp, #0x40`
	`STP x29, x30, [sp,#0x30]`
	`ADD x29, sp, #0x30`
	…
		`SUB sp, sp, #0x50`
		`STP x29, x30, [sp, #0x40]`
		`ADD x29, sp, #0x40`
		`ADRP  x3, {pc}`
		`LDR   x4, [x3, #SSP]`
		`STR   x4, [sp, #0x38]`
		…

Function Epilogue	…
	`LDP x29,x30,[sp,#0x30]`
	`ADD sp,sp,#0x40`
	`RET`
		…
		`LDR     x1, [x3, #SSP]`
		`LDR     x2, [sp, #0x38]`
		`CMP     x1, x2`
		`B.NE    __stack_chk_fail`
		`LDP x29, x30, [sp, #0x40]`
		`ADD sp, sp, #0x50`
		`RET`

The basic Pointer Authentication instructions PAC and AUT can be used to implement a stronger version of stack protection with much less overhead:

	No stack protection	With Pointer Authentication

Function Prologue	`SUB sp, sp, #0x40`
	`STP x29, x30, [sp,#0x30]`
	`ADD x29, sp, #0x30`
	…
		`PACIASP`
		`SUB sp, sp, #0x40`
		`STP x29, x30, [sp,#0x30]`
		`ADD x29, sp, #0x30`
		…

Function Epilogue	…
	`LDP x29,x30,[sp,#0x30]`
	`ADD sp,sp,#0x40`
	`RET`
		…
		`LDP x29,x30,[sp,#0x30]`
		`ADD sp,sp,#0x40`
		`AUTIASP`
		`RET`

The PACIASP and AUTIASP instructions used above are specialized versions of the more general PACIA X30, SP and AUTIA X30, SP instructions. They tag and verify the link register (X30) with the stack pointer as the context respectively. These specialized instructions are in NOP space which means the generated code remains binary compatible with older processors. If backwards compatibility is not needed, the combined Pointer Authentication instruction RETAA, which is equivalent to AUTIASP+RET, can be used to save an extra instruction.

Control Flow Integrity (CFI)

CFI is a property of a program where the control flow follows only legitimate paths determined in advance. While a secure CFI implementation needs to cover both function pointers and return addresses, existing CFI implementations generally focus on protecting function pointers and leave the return address protection to other countermeasure like strong stack protection or shadow/split stack implementations.

In CFI, functions that are reachable through indirect calls/jumps are classified based on compile time analysis or heuristics, and each call site is restricted to call only function pointers of the same class. This is typically implemented by computing and maintaining tables of function pointers in memory and adding checks before each indirect call/jump to make sure the target is within the table.

When using Pointer Authentication for CFI, a possible approach would be to pre-authenticate function pointers in memory at load and dynamic link time, based on the class (context) and location information for each pointer generated at compile-time. This allows validation of indirect function calls with a single AUT instruction at the call site using the context provided at compile-time.

Binding Pointers to Addresses

The Pointer Authentication instructions can be used to fix the value of a pointer at a given address by a PAC using the address of the pointer as the context. This makes sure that the pointer value remains unmodified, effectively making that location read-only. This is useful for pointers that do not change much over the lifetime of the program, such as pointers to read-only tables or data structures.

Security Properties

This section describes security properties, including strengths and weaknesses of Pointer Authentication. Note that most of this discussion assumes that data execution prevention (DEP) is enabled so that code is not writable and data is not executable.

Arbitrary memory read

Most software countermeasures that rely on secrets are vulnerable to exploit primitives that disclose memory contents. This includes ASLR, software stack protection and other schemes that obfuscate pointer values using XOR with a secret.

Pointer Authentication is designed to resist memory disclosure attacks. The PAC is computed using a cryptographically strong algorithm, so reading any number of authenticated pointers from memory would not make it easier to forge pointers.

The keys are stored in processor registers, and these registers are not accessible from usermode (EL0). Therefore, a memory disclosure vulnerability would not help extract the keys used for PAC generation.

Arbitrary memory write

An arbitrary memory write exploit targeting writable code pointers in memory would have to guess or brute-force the PAC or find a non-control data exploit that modifies the behavior of the system. A kernel exploit that changes the effective-uid of an attacker controlled process to root, and a remote exploit that changes the “authenticated” flag in memory to true to bypass a password check are examples of data-only attacks. Pointer Authentication extensions do not provide a generic mechanism to protect against these attacks. However, authenticating sensitive data pointers and protecting sensitive data structures using PACGA can help limit attacker capabilities.

One interesting property of Pointer Authentication is that corruption can be detected even if the modification happens outside the processor core, e.g. through DMA attacks. This does not help too much in the general case where the code or other read-only regions may be targeted, but it may provide protection when external access is limited to certain regions of the physical address space, such as with an SMMU/IOMMU.

Guessing and forging PAC values

The difficulty of guessing the PAC of a target pointer depends on the size of the PAC, which is system configuration dependent. While it can be as low as 3 bits (1 in 8 chance), the Linux kernel uses a 39 bit virtual addresses space (512GB) on AArch64 by default. This allows for a 24-bit PAC (1 in 4M) in Linux. With the maximum 52-bit virtual address size configuration PAC will have 11 bits (1 in 2048) when tagged addresses are disabled which is still comparable to some ASLR implementations. Systems that require more security, such as the TrustZone kernel and applications, could modify the virtual address size configuration for these execution modes to increase the PAC size to up to 31 bits.

The QARMA algorithm is explicitly designed to handle short/truncated authentication codes. We do not expect any shortcuts in the algorithm where leaking or guessing a tag would make subsequent guesses easier. Indeed, assuming a successful guess of a tag implies that only a few bits of the QARMA output are known, instead of the whole ciphertext – and any reduction in time or space complexity of a cryptanalytic attack on QARMA would require a large amount of known (or even chosen) text pairs, which is beyond the amount of information collectible from a process because of effective memory size. Therefore, the difficulty of guessing does increase exponentially. Any exploit that needs to modify or forge more than one pointer will find it prohibitively expensive. This effectively raises the bar in a significant way for the attacker.

Pointer substitution attacks

We expect that most attacks against Pointer Authentication will be in the form of substituting one authenticated pointer with another. For instance, if an authenticated function pointer that points to *puts()* and prints out an attacker supplied string could be replaced with an authenticated pointer that points to the *system()* function, this would allow execution of attacker supplied commands.

The Pointer Authentication design specifies different keys for different usage categories these keys are part of the instruction encoding and determined at compile-time. The context argument can be used to further classify pointers into groups to restrict which pointers can be substituted for each other. This is not much different from fine-grained Control-Flow-Integrity [6] where each indirect call-site can only call a subset of functions based on a compile-time generated control-flow graph. The context argument can be used to enforce similar restrictions.

During architecture discussions, a very interesting pointer substitution attack, devised by an ARM partner evaluating the proposal, was described to us by ARM. The attack assumes, in addition to a memory corruption exploit, a primitive that can read stack contents and the ability to trigger different functionality on the same stack. Web browsers would probably satisfy these requirements. Since the stack pointer is used as the context when computing PAC for the return addresses, it becomes possible to collect multiple authenticated pointers bound to different stack addresses. These addresses can then be used to form a sequence of gadgets (bound to their original position in stack) to execute a ROP payload. Given the power of primitives already available to the attacker, however, it is hard to evaluate the difficulty of carrying out this attack as opposed to some other way of compromising the environment. This confirms once again that Pointer Authentication is not a silver bullet and that browser security is hard.

Key management concerns and key reuse attacks

An attacker can forge pointers if she can guess the key or gains control of a process that has the same key as the target process.

The first concern is about quality of the randomness used to generate the key. While the ARM architecture does not include an entropy source, we would like to assume that all modern ARMv8.3-A based designs would already include a high-quality entropy source accessible at early boot. Qualcomm Technologies chips that support ARMv8 already have a hardware RNG available.

The second issue is more common, especially in UNIX-like systems, where the *fork()* system call creates a complete duplicate of a process. This means that the child processes must now have the same keys as its parent for it to continue running. Note that this issue exists for other countermeasures including stack protection and ASLR.

In a privilege separation design where one of the processes remain privileged while the other drops privileges, compromising the unprivileged process would allow an attacker to create authenticated pointers that would work in the privileged process.

In a worker process model where several worker processes are spawned as needed by a master process to service requests, a remote attacker can gain multiple, potentially unlimited, tries to brute-force PAC values since all workers will have the same key, including the ones that are forked to replace the ones that die due to address faults.

In both cases, using the fork+exec model where the child process reloads itself right after fork makes sure the all processes start fresh with a new key (and new address space layout with ASLR).

Since implementing fork+exec requires changes to the design and existing code, and may impact the performance/latency of the service, kernels should consider implementing special handling for PAC exceptions: When one process receives a PAC exception, the kernel should kill not only the faulting process, but all other processes that have the same key. This is doable since the kernel is responsible for managing the keys for the processes that use Pointer Authentication.

Interpreters and Just-in-Time Compilation (JIT)

Scripts and bytecode interpreters provide a good target for attackers to inject data into a system which is then interpreted as instructions. PAC does not protect against data-only attacks, but it does make it harder to jump to the entry point of the interpreter with attacker controlled data.

Most of these runtime environments also support JIT compilation where native code is dynamically generated from the script/byte-code for improved performance. Some JIT implementations use writable and executable (RWX) memory regions that break DEP, while others pretend to adhere to DEP/W^X requirements by using aliased mappings, accessing the same physical memory range from two virtual addresses, one writable, the other executable. While the RWX case is probably worse, both schemes provide a target for injecting code into the system, potentially bypassing Pointer Authentication and other countermeasures.

Randomizing the location of the writable JIT region is one mitigation that may help against these attacks. Turning off JIT should also be considered as an option in security critical environments.

Conclusion

In this document, we have described the design of the ARM Pointer Authentication extensions newly introduced in ARMv8.3-A specification. We have presented some common use-cases that we expect to see in toolchains and provided a brief security analysis of the scheme. The scenarios we presented in this document are some of the examples where we thing the Pointer Authentication instructions would be useful. By providing a way for quickly verifying the integrity of pointers and data in memory, these instructions will create opportunities for additional techniques and applications to emerge.

References

[1] ARM Connected Community Blog, 2016, “ARMv8-A architecture – 2016 additions”, available at https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions

[2] Aumasson, et al., 2012 “SipHash: a fast short-input PRF” INDOCRYPT 2012, full version of the paper available at https://131002.net/siphash/siphash.pdf

[3] Borghoff, et al., 2012, “PRINCE – A Low-latency Block Cipher for Pervasive Computing Applications” ASIACRYPT 2012, full version of the paper available at https://eprint.iacr.org/2012/529.pdf

[4] Avanzi, 2016, “The QARMA Block Cipher Family”, full version of the paper available at https://eprint.iacr.org/2016/444.pdf

[5] Beierle, et al., 2016, “The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS”, CRYPTO 2016, full version of the paper available at https://eprint.iacr.org/2016/660

[6] Abadi, et al., 2005, “Control-Flow Integrity: Principles, Implementations and Applications” available at http://research.microsoft.com/pubs/64250/ccs05.pdf