Handling collision in PACStack without masking

Henry Donahue
Brown University
henry _donahue@brown.edu

Abstract

Backwards-edge control flow hijacking represents a significant threat to the security of C/C++ programs. Through the compromise of function return addresses, an adversary can redirect execution down illegitimate paths. Shadow stacks have proven to defend manipulation of return addresses, but at a cost. Shadow stacks keep a record of return addresses used to ensure they have not been hijacked at runtime with hardware-assistance or software control. Software shadow stacks incur high overheads or trade off security for efficiency. Hardware assisted shadow stacks are efficient and secure, but require specialized hardware, limiting widespread adoption–especially in legacy systems.

As a result of shadow stack's limitations, PACStack—an authenticated Call Stack (ACS)—was introduced using ARM's Pointer Authentication Codes (PACs). PACStack generates authentication codes for each return address using an amalgamation of the return addresses used to arrive to the current location. Authentication codes are generated and stored in the vacant first 16 bits of the register containing the return address, using the instruction `PACIA`. The shorter an authentication code, the more likely it is that two inputs will produce the same output; two inputs will collide. PACStack circumvents this issue by masking the register containing the authentication code and return address, preventing an adversary from recognizing collision.

This paper explores alterations to PACStack to prevent collision and thereby enhancing security against sophisticated collision-based attacks and nullifying the need to mask the register containing the authentication code and return address after the initial generation.

1 Introduction

Shadow stacks, an earlier and more developed defense to protect return addresses, keep a secured record of valid return address in a separate, integrity-protected memory area. Shadow stacks can be implemented using software, which tends to be less efficient, or hardware, which is more efficient but requires specific hardware support. PACStack aims to secure the integrity of call stacks without the hardware dependency or software overhead of a shadow stack [2,4].

PACStack differs from traditional return address defenses, i.e. shadow stacks, by utilizing the recently developed Pointer Authentication on ARMv8. PACStack only requires hardware support in the form of a single trusted/encrypted register and the `PACIA` and `AUTIA` instructions.

PACStack generates the current return address authentication code using a unique modifier each time. This modifier represents the path to the current location, which is unique. This ensures that each PAC is contextually bound not only to the pointer it authenticates but also to a unique execution instance, enhancing the overall robustness of the system against sophisticated attacks aiming to compromise the control flow integrity. If any address in this sequence is tampered with, the integrity check will fail, thus ensuring the security of the call stack. However, while this method significantly improves security, it is susceptible to collision attacks. If two different inputs collide, produce the same PAC, an adversary can switch a valid return address with a malicious one that has the same PAC. In a collision attack, an adversary observes the PAC values over time and identify two different return addresses that yield the same PAC. Such collisions could be used to manipulate the control flow of a program by substituting a legitimate return address with another that has the same PAC, yet leads to a malicious function or code snippet [5,6].

To address this vulnerability, PACStack generates a pseudo-random mask by calculating a PAC using a zero constant and the previous authenticated return address. This mask is then applied using an exclusive-OR (XOR) operation to the authentication code of the current return address. This mask is unique for each return address, thereby randomizing the masked value. As a result, even if an attacker calculates or predicts a PAC, the mask alters the PAC in a way that is unpredictable to the attacker. This removes the danger of PAC collision and reduces an adversary's options to solely guessing. By implementing PAC masking, PACStack effectively mitigates the risks associated with PAC collisions. Thus, PAC masking is a critical component of PACStack's strategy to enforce return address security [5].

If the authentication codes never collided, masking would not be necessary. The primary reason for PACStack not using a longer authentication code is the ARM design choice to have `PACIA` return a 16 bit authenticaion code. This choice was made to easily store the code in the unused 16 bits of an address.

There is however an instruction `PACGA` that generates a 32 bit authentication code. There is no corresponding authentication instruction. Instead, authentication can be done by recomputing the authentication code in the epilogue and making sure it is the same as it was in the prologue. Since a 32 bit hash has millions of possible output values, it is impossible to observe collision in any reasonable amount of time [7, 10].

2 Background

2.1 Return Oriented Programing

Return Oriented Progrmamming (ROP) is a software attack vector in which an adversary exploits a memory vulnerability to manipulate return addresses stored on the stack, thereby altering the program's backward-edge control flow. An adversary can chain together multiple gadgets, adversary-chosen sequences of pre-existing program instructions, that together perform a desired malicious operation [8, 12].

2.2 ARM Pointer Authentication

ARM's Pointer Authentication, introduced in ARMv8.3-A, is a crucial enhancement aimed at strengthening software security, particularly against exploits that manipulate pointers like return addresses or function pointers, like ROP. This mechanism embeds a PAC into unused bits of a pointer, and validating its integrity before use. Pointer Authentication (PA) adds a layer of cryptographic security that ensures pointers have not been tampered with by unauthorized changes [9].

The functionality of Pointer Authentication revolves around new ARM instructions that handle the code generation, insertion, verification, and management of PACs. The cryptography underlying these operations is robust, utilizing block ciphers (such as universal hashing) for generating short authentication codes that are computationally feasible to handle at runtime without significant performance overhead [1].

PACStack uses the following commands in it's implementation:

`PACIA` This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A. The authentication code is 16 bits long and stored in the high order bits of the address it is authenticating.

`AUTIA` This instruction authenticates an instruction address, using a modifier and key A. It validates the authentication code and then removes it, crashes if invalid code.

This paper uses those instructions as well as their mirror version, `PACIB` and `AUTIB`, which differ only in using key B. As well as,

`PACGA` This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of a seperate destination register.

The implementation of Pointer Authentication addresses a fundamental problem in software security: memory corruption vulnerabilities. By authenticating pointers before use, the mechanism drastically reduces the likelihood that an attacker can redirect program flow through corrupted pointers. This method is not only a preventive measure but also a detection mechanism, as any tampered pointer fails authentication, leading to an exception. As such, Pointer Authentication is a powerful tool for maintaining control flow integrity, protecting against sophisticated exploits, and ensuring that even if an attacker can read memory or execute arbitrary code, the integrity of pointer data remains intact, safeguarding the overall security of the system [1, 3, 5, 6, 9].

2.3 PACStack: An Authenticated Call Stack

PACStack enforces the security of a shadow stack with low overhead and only the need for a single trusted/encrypted register. It achieves this by using the ARM `PACIA` and `AUTIA` instruction. An amalgamation of the past return address is used as the modifier.

```
// prologue:
str X28 , [SP , # -32]! ; stack <- aret_{i-1}
stp FP , LR , [SP ,#16] ; stack <- frame-record
pacia LR , X28          ; LR <- aret_{i}
mov X28 , LR            ; CR <- aret_{i}
...
// epilogue:
mov LR , X28          ; LR <- aret_{i}
ldr FP , [SP , #16] ; skip ret_{i}' in frameRec
ldr X28 [SP], #32   ; CR <- aret_{i}' from stack
autia LR , X28        ; LR <- (ret_{i} or ret_{i}')
```

Listing 1: PACStack without masking

Now, since `PACIA` only generates an authentication code of 16 bit length, by the birthday paradox collision occurs after only 321 tokens are generated [6, 7, 10]. Once collision is found, valid authentication codes can in essence be fabricated by an adversary. PACStack masks the authentication code to prevents an adversary from recognizing collision.

```
// prologue:
str X28 , [SP , # -32]! ; stack  aret_{i-1}
```

```
stp FP , LR , [SP ,#16] ; stack  frame-record
mov X15 , XZR ; X15 <- 0
pacia LR , X28 ; LR <- aret_{i}^{unmasked}
pacia X15 , X28 ; X15 <- 0
eor LR , LR , X15 ; LR <- mask_{i} XOR aret_{i}^{unmasked}
mov X15 , XZR ; X15 <- 0
mov X28 , LR ; CR <- aret_{i}
...
// epilogue:
mov LR , X28 ; LR <- aret_{i}
ldr FP , [SP , #16] ; skip ret_{i}' in FP
ldr X28 , [SP], #32 ; CR <- aret_{i-1}' from stack
mov X15 , XZR ; X15 <- 0
pacia X15 , X28 ; X15 <- mask_{i}
eor LR , LR , X15 ; LR <- mask_{i} XOR aret_{i}
mov X15 , XZR ; X15 <- 0
autia LR , X28 ; LR <- (ret_{i} or ret_{i}*)
ret
```

Listing 2: PACStack with masking

3 Implementation

Two Hash PACStack extends the time until collision by breaking the return address into two halves and hashing one half with `PACIA` and the other with `PACIB`. It achieves the same level of defense and overhead as PACStack with masks. With the possibility of even a strong level of security. If an adversary could unmask authentication tokens in PACStack with masks, it reduces to PACStack without masks and collision will occur after 321 authentication tokens are generated. This number of tokens is derived from the result of the birthday paradox, $1.253 * 2^{b/2}$ [5,7,10]. Undoing the mask is infeasible but not outright impossible.

Waiting for two independent hash functions to collide however does turn time until collision into $(1.253 * 2^{b/2})^2 = 1.57 * 2^b$. This is because an adversary needs both hash functions to collide at the same time for them to gain the ability to abuse collision.

```
// prologue:
mov   x1,lr ; Move link register (lr) to x1
and   x1,x1,#0xFFFFFF000000 ; Mask the upper 24 bits of x1
and   lr,lr,#0x000000FFFFFF ; Mask the lower 24 bits of lr
pacib lr,x28 ; Generate authentication code for lr, using modifier x28
pacia x1,x28 ; Generate authentication code for x1, using modifier x28
stp   lr,x1,[sp,#-16]! ; Store lr x1 on the stack, update stack pointer
str   x28   ,[sp,#-16]! ; Store x28 on the stack, update stack pointer
eor   x28,lr,x1     ; Exclusive OR operation between lr and x1, result stored in x28 - use both halves for feedback
...
// epilogue:
ldr     x28   ,[sp],#16
ldp     lr ,x1,[sp],#16  //encrypted pair
autib   lr ,x28
autia   x1 ,x28
orr     lr ,lr,x1     //restore return address
ret
```

Listing 3: Two Hash PACStack

The `PACGA` PACStack implementation uses `PACGA` to generate a 32 bit authentication code for a given address using a given modifier. The modifier here is the same as in regular PACStack with or without masking, the amalgamation of past return addresses. This implementation avoids collision as an adversary would now have to observe $1.253 * 2^{32/2} = 82117$ PACs before a collision occurs.

```
// prologue::
pacga x1,lr,x28
stp   lr,x28,[sp,#-16]!
str   x1,     [sp,#-16]!
mov   x28,lr ; feedback into next modifier
...
// epilogue:
ldr   x1     ,[sp],#16
ldp   lr,x28,[sp],#16
pacga x2,lr,x28
cmp   x1,x2
b.eq  pac_pass ; x1 and x2 must match
autia lr,x28 ; "decrypt" an unencrypted return addr => should always fail
pac_pass:
ret
```

Listing 4: PACGA PACStack

During the epilogue, the 32 bit authentication code is recomputed and compared to the initial code. If they are different, an error will occur.

4 Performance Overhead

To measure overhead without writing an LLVM pass and comparing benchmarks, each of the various authenticated call stack implementation's runtime is measured on a basic ARM assembly sum function `sum.s` that sums all numbers from 0 to n: $\sum_{i=o}^{n} = \frac{n(n+1)}{2}$. Although an imperfect evaluation of overhead, it suffices to generate primitive results.

Table 1: Average Execution Times by method

Function Name	Average Time (seconds)
sum.arm64 (benchmark)	0.581
sum-pac.arm64	1.268
sum-pac-mask.arm64	2.580
sum-pac2.arm64	2.618
sum-pacga.arm64	1.414

The Two Hash PACStack has nearly the same runtime as PACStack with masking. This difference is likely due to `AUTIA/B` having more overhead then `PACIA/B`. PACStack with masking calls `PACIA` 3 times and `AUTIA` once. Two hash PACStack calls AUTIA/B and AUTIA/B twice. The increase overhead could also be from the two hash PACStack method having one more instruction per cycle.

The PACGA PACStack method has a runtime in between PACStack with no masks and PACStack with masking.

5 Future Work

PACGA PACStack could be further developed to store half of it's output in the trusted register with the return address, just as in PACStack with or without masking. The other half of the code could be stored anywhere else. For verification in the epilogue, it would need to be stitched back together and then compared to the recomputed authentication code. This would result in the same design as PACStack without masking but with a 32 bit authentication code. As the overhead for PACGA PACStack is quite low, adding in this additionally functionality would most likely still keep it's overhead below PACStack with masking. An adversary would have access to half of the authenticating code bits and could override them or use them to inform guessing. In the case of overriding, the comparing step would fail. Guessing would still be ineffective because knowing half of an authentication code output would still not help an adversary solve a 32 bit hash function.

Two Hash PACStack could be further developed to split the return address in a more complex/harder to unwind way then just a bottom and top half. Instead of storing one inside the trusted register and one outside, store 8 bits of each authentication codes in the trusted register and the other 8 bits else where.

Due to just learning ARM assembly, LLDB and clang, this functionality was unable to be implemented. But, the working implementation show proof of concept and are valid implementations in their own right. These proposed continuations would further reinforce the designs.

6 Conclusion

PACStack, an authenticated call stacks, achieves security on-par with shadow stacks without the necessary hardware support or compromising security for overhead. Due to the short length of the authentication code output, masking is required to prevent an adversary from exploiting collision. This paper explores alternative ways to generated and store message authentication codes to prevent collision and nullify the need to mask message authentication codes.

The two alternative methods to masking provide either similar or better overhead while providing the same level of security. Additionally, they provide stronger guarantees of security than a masking operation. Masking could possibly be reverse engineered and bypassed at runtime. Whereas there is no way to bypass hash functions besides collision, but collision becomes incredibly unlikely with two 16 bit/a 32 bit authentication code. Both Two Hash and PACGA PACStack leverage security through the use of hash functions with longer output values, extending time until collision and alleviating the need for masking.

Availability

The project's code can be found on the following GitHub repository. There is a detailed README with instructions on how to run and clear descriptions of each file. `https://github.com/henrypdonahue/2951U`.

A virtual environment using ARMv8 or an Apple silicon device is needed. Apple does not allow users to compile ARM64e binaries by default. The following stack overflow discussion walks a user through disabling this to allow the user to compile with the needed instructions by disabling settings in recovery mode. [11]

Acknowledgments

I would like to thank Professor Vasileios P. Kemerlis for two wonderful semester at Brown University. CSCI 1650 and 2951U have exposed me to the fascinating world of software security. I also would like to thank my fellow classmates in CSCI 2951U for their insightful comments, productive peer review/study sessions and comradery.

References

[1] ARM. Arm a-profile a64 instruction set architecture, 2024.

[2] Burow, N., Zhang, X., and Payer, M. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)* (May 2019), IEEE.

[3] Cai, Z., Zhu, J., Shen, W., Yang, Y., Chang, R., Wang, Y., Li, J., and Ren, K. Demystifying pointer authentication on apple m1. In *32nd USENIX Security Symposium (USENIX Security 23)* (Anaheim, CA, Aug. 2023), USENIX Association, pp. 2833–2848.

[4] Li, J., Chen, L., Xu, Q., Tian, L., Shi, G., Chen, K., and Meng, D. Zipper stack: Shadow stacks without shadow, 2020.

[5] Liljestrand, H., Nyman, T., Gunn, L. J., Ekberg, J.-E., and Asokan, N. PACStack: an authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 357–374.

[6] Liljestrand, H., Nyman, T., Wang, K., Perez, C. C., Ekberg, J.-E., and Asokan, N. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 177–194.

[7] Mashtizadeh, A. J., Bittau, A., Mazieres, D., and Boneh, D. Cryptographically enforced control flow integrity, 2014.

[8] Pappas, V., Polychronakis, M., and Keromytis, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy* (2012), pp. 601–615.

[9] Qualcomm. Pointer authentication on armv8.3 design and analysis of the new software security instructions, 2017.

[10] SMART, N. *Cryptography Made Simple*. Information Security and Cryptography. Springer, 2016.

[11] STACKOVERFLOW. How to enable the arm pointer authentication code (pac) on macos? Stack Overflow.

[12] ZHANG, J., HOU, R., SONG, W., MCKEE, S. A., JIA, Z., ZHENG, C., CHEN, M., ZHANG, L., AND MENG, D. Raguard: An efficient and user-transparent hardware mechanism against rop attacks. *ACM Trans. Archit. Code Optim. 15*, 4 (nov 2018).