This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.

Liljestrand, Hans; Nyman, Thomas; Ekberg, Jan Erik; Asokan, N.

Late breaking results: Authenticated call stack

Published in:

Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019

DOI:

[10.1145/3316781.3322469](https://doi.org/10.1145/3316781.3322469)

Published: 02/06/2019

Document Version

Peer reviewed version

Please cite the original version:

Liljestrand, H., Nyman, T., Ekberg, J. E., & Asokan, N. (2019). Late breaking results: Authenticated call stack. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019* [223] (Proceedings - Design Automation Conference). ACM. <https://doi.org/10.1145/3316781.3322469>

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

Authenticated Call Stack

Hans Liljestrand
Aalto University, Finland
Huawei Technologies Oy,
Finland
hans.liljestrand@aalto.fi

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Jan-Erik Ekberg
Huawei Technologies Oy,
Finland
Aalto University, Finland
jan.erik.ekberg@huawei.com

N. Asokan
Aalto University, Finland
asokan@acm.org

ABSTRACT

Shadow stacks are the go-to solution for perfect backward-edge control-flow integrity (CFI). Software shadow stacks trade off security for performance. Hardware-assisted shadow stacks are efficient and secure, but expensive to deploy. We present *authenticated call stack* (ACS), a novel mechanism for precise verification of return addresses using aggregated message authentication codes. We show how ACS can be realized using ARMv8.3-A pointer authentication, a new low-overhead mechanism for protecting pointer integrity. Our solution achieves security comparable to hardware-assisted shadow stacks, while incurring negligible performance overhead ($< 0.5\%$) but requiring *no additional hardware support*.

1 INTRODUCTION

Control-flow attacks are a prevalent threat; defences such as data execution prevention (DEP) can be circumvented using code-reuse attacks, such as return oriented programming (ROP). ROP is performed by corrupting function return addresses to alter a program's backward-edge control-flow. The gold standard for preventing ROP is the shadow stack [1], which securely stores a copy of each return address, thus allowing exact verification of returns. While software solutions can achieve reasonable performance [4], so far, only hardware-assisted shadow stacks, such as Intel CET [5], promise negligible overhead without security trade-offs. ARMv8.3-A recently introduced pointer authentication (PA) [2] which can be used to protect return addresses [8]. Current PA schemes are vulnerable to *reuse attacks*, where previously signed pointers are reused to bypass authentication [7].

In this paper, we propose *authenticated call stack* (ACS), a novel scheme to precisely protect return addresses, that achieves security comparable to hardware-assisted shadow stacks, with minimal overhead, and without integrity-protected memory. ACS aggregates message authentication codes (MACs) bound to return addresses. We show how ACS can be realized using ARM PA. The resulting system, PACStack, is efficient, can withstand strong adversaries with full memory access, and requires no additional hardware. Our contributions are:

- ACS, an aggregated MAC design that provides precise verification of return addresses (Section 4).
- PACStack, a LLVM-based ACS implementation using ARM PA to achieve negligible overhead ($< 0.5\%$) at no additional hardware cost (Sections 5 and 6).

2 BACKGROUND

ROP on ARM: In ROP, the attacker exploits a memory vulnerability to manipulate return addresses and alter the program's backward-edge control-flow. ROP allows Turing-complete attacks by chaining multiple returns into attacker-chosen code. The ARM architecture uses a dedicated *link register* (LR) to hold the current function's return address. For non-leaf function calls the LR value must be stored on the stack, thereby allowing ROP [6].

ARM PA: ARMv8.3-A introduced PA, which supports authenticating pointers [2]. Pointers are signed using a PA instruction to calculate a MAC, $H_K(V||M)$, using key K , a given input V and a 64-bit modifier M . The resulting MAC, the pointer authentication code (PAC), is typically embedded into the pointer itself. On a 64-bit ARM running a default Linux kernel, the virtual address (VA) is 39 bits, which, excluding other reserved bits, leaves 16 for the PAC. For example, the *pacia* and *autia* instructions generate a PAC based on the input pointer's VA and an arbitrary 64-bit modifier to authenticate the pointer value, VA. Additionally a generic PA instruction, *pacga*, outputs a separate 32-bit PAC based on an arbitrary 64-bit input and a 64-bit modifier. PA will be supported in Linux 5.0¹, where the kernel will set a process' keys at exec.

3 THREAT MODEL

In this work, we assume a powerful adversary with arbitrary read and write access to process memory, restricted by DEP. This threat model is consistent with prior work on control-flow attacks. We do not consider adversaries that can manipulate kernel memory, or directly alter CPU configuration and registers. Software-only shadow stacks cannot efficiently protect against such an adversary.

4 DESIGN

On ARM, the return address, ret_n , of a leaf-function can be securely stored in LR. Prior return addresses $ret_i, i \in [0, n-1]$ must however be stored on the stack. ACS protects these values by aggregating a series of iterative MACs $auth_i, i \in [0, n]$ that cryptographically bind all return values $ret_i, i \in [0, n]$ to the latest $auth_n$ (Figure 1). The topmost token, $auth_n$, is securely stored in LR to prevent its modification. To prevent pre-computation of hash values, we utilize a keyed MAC function H_K to generate $auth_i$:

$$auth_i = \begin{cases} H_K(ret_i || auth_{i-1}) & \text{if } i > 0 \\ H_K(ret_i || 0) & \text{if } i = 0 \end{cases}$$

On function call, LR cannot be used to hold $auth_i$; it is instead put in the *chain register* (CR in Figure 2). After verification of $auth_{i-1}$ and ret_i in the function epilogue, $auth_{i-1}$ is put in CR.

© 2019. Copyright held by the owner/author(s). This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA, <https://doi.org/10.1145/3316781.3322469>.

¹kernelnewbies.org/Linux_5.0

Figure 1: ACS is an aggregated MAC of tokens $auth_i$ bound to previous tokens and the corresponding return addresses ret_i .

Figure 2: ACS keeps the topmost $auth_n$ secure in LR, or at function call, in CR. Function prologues generate a new $auth_n$, and epilogues validate $auth_{n-i}$ and ret_n against $auth_n$.

Authenticated return-address variant To avoid storing both $auth_i$ and ret_i we define a combined $aret_i$:

$$aret_i = auth_i || ret_i, \text{ where}$$

$$auth_i = \begin{cases} H_K(ret_i || aret_{i-1}) || ret_i & \text{if } i > 0 \\ H_K(ret_i || 0) || ret_i & \text{if } i = 0 \end{cases}$$

We secure $aret_n$ in LR, and store $aret_i, i \in [0, n-1]$ on the stack.

Irregular stack unwinding In some cases, notably C++ exceptions and C setjmp / longjmp, the call-stack is unwound in an irregular manner. To preserve ACS integrity, the unwinding is modified such that the associated data (e.g., jmp_buf) is tied to the corresponding $auth_i$, i.e., $auth_{buf} = H_K(jmp_buf || auth_i)$.

5 IMPLEMENTATION

We implement PACStack, an ACS solution on top of LLVM 7.0, using PA to efficiently compute $auth$ and $aret$ values. We use Mark Rutland’s Linux kernel patches².

pacga-based instrumentation: Our first variant uses pacga as H_K to generate and verify $auth_i$ in the function prologue and epilogue, respectively. Call-sites are instrumented to pass $auth_i$ via CR, set to x28. x28 is a callee saved register, i.e., its value is preserved across calls, allowing PACStack to call non-instrumented code.

autia- / pacia- based instrumentation: This variant using pacia and autia to authenticate return addresses avoids additional store / load operations as CR is stored on the stack instead of LR.

²lwn.net/Articles/752116/

³www.math.utah.edu/~mayer/linux/bmark.html

⁴A comprehensive security analysis in full paper at pacstack.github.io

⁵community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a

6 EVALUATION

Performance: We use nbench-byte-2.2.3³ on a 960board Kirin 620 HiKey running Linux kernel v4.18.0 and BusyBox v1.29.2 for evaluation. Because the hardware lacks PA support we simulate PA overhead based on estimated costs of four cycles [3, 7]. Our functional evaluation was done on a Linaro FVP simulator that supports PA instructions. Our current PACStack instrumentation is based on the autia / pacia variant and incurs a < 0.5% overhead.

Security evaluation⁴: ROP on the pacga-variant, requires modifying the stack to inject a ret'_n and $auth'_{n-1}$, such that $auth_n = H_K(ret'_n || auth'_{n-1})$. Because the attacker cannot control $auth_n$, the only course of action is correctly guessing the 32-bit $auth'_{n-1}$, which yields a success probability of 2^{-32} . The second variant is subtly different: an attacker must correctly guess both $auth'_{n-i}$ and $auth'_{n-2}$. First to inject an $aret'_{n-1}$ that passes authentication against and replaces $aret_n$ in LR, but still returns to ret_n . And second, to inject an $aret'_{n-2}$, such that the return to $aret_{n-1}$ passes authentication and a return to ret'_{n-1} is performed. The guesses cannot be done separately, thus the attack probability on a system with 16-bit PAC size is 2^{-32} . The security of PACStack depends on the cryptographic properties of PA and the integrity of the topmost $auth_n$.

7 CONCLUSION

We showed how a general-purpose hardware security mechanism (ARM PA) can provide security similar to hardware-assisted shadow stacks, *without requiring additional hardware support or sacrificing CFI precision*. Other general-purpose primitives are being rolled out⁵. Creative uses of such primitives hold the promise of significantly improving software protection.

REFERENCES

- [1] Martín Abadi et al. 2009. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (Nov. 2009), 4:1–4:40.
- [2] ARM Ltd. 2017. ARMv8 Architecture Reference Manual, for ARMv8-A architecture profile (ARM DDI 0487C.a). https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.
- [3] Roberto Avanzi. 2017. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Trans. Symmetric Cryptol.* 2017, 1 (2017), 4–44.
- [4] Nathan Burrow et al. 2019. SoK: Shining Light on Shadow Stacks. *arXiv:1811.03165v2 [cs.CR]*. <https://arxiv.org/abs/1811.03165v2>
- [5] Intel. 2016. Control-flow Enforcement Technology Preview. <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>.
- [6] Tim Kornau. 2009. *Return Oriented Programming for the ARM Architecture*. Ph.D. Dissertation. Ruhr-Universität Bochum.
- [7] Hans Liljestrand et al. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. (to appear) Usenix SEC 2019, arXiv: 1811.09189 [cs.CR]. <https://arxiv.org/abs/1811.09189>
- [8] Qualcomm. 2017. Pointer Authentication on ARMv8.3. <https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf>.