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

Published: 02/06/2019

*Document Version*
Peer reviewed version

*Please cite the original version:*

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
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 auth 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 $auth_{i-1}, i \in [0, n]$ 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.

¹kernelnewbies.org/Linux_5.0
6 EVALUATION

Performance: We use nbench-byte-2.2.3\(^3\) 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 cost 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 evaluation\(^4\): ROP on the pacga-variant, requires modifying the stack to inject a ret\(_i\) and auth\(_{n-1}\), such that auth\(_n\) = H\(_K\)(ret\(_i\)||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\(_{n-1}\) and auth\(_{n-2}\). First to inject an ret\(_i\) that passes authentication against and replaces ret\(_n\) in LR, but still returns to ret\(_n\). And second, to inject an ret\(_n\) such that the return to ret\(_n\) 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 topest 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 out\(^5\). Creative uses of such primitives hold the promise of significantly improving software protection.

REFERENCES


---

\(^1\)www.net/Articles/752116/
\(^2\)www.math.utah.edu/ mayer/linux/bmark.html
\(^3\)A comprehensive security analysis in full paper at pacstack.github.io
\(^4\)community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv8a
\(^5\)