Breaking Active-Set Backward-Edge CFI

Michael Theodorides and David Wagner
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{theodorides,daw}@cs.berkeley.edu

***Abstract*—Hardware-Assisted Flow Integrity eXtension (HAFIX) was proposed as a defense against code-reuse attacks that exploit backward edges (returns). HAFIX provides fine-grained protection by confining return addresses to only target call sites in functions active on the call stack. We study whether the backward-edge policy in HAFIX is sufficient to prevent code-reuse exploits on real-world programs. In this paper, we present three general attacks that exploit weaknesses in HAFIX and demonstrate these attacks are effective in case studies examining Nginx web server, Exim mail server, and PHP. We then propose improvements to HAFIX we believe will improve its effectiveness against code-reuse attacks.**

I. Introduction

Memory-safety vulnerabilities have been used to exploit systems for over two decades. Researchers have studied many defenses against these attacks, yet performance and other limitations of these defenses have meant that memory-safety exploits remain ubiquitous [1]. Data Execution Prevention (DEP) [2], which marks pages of memory used for data as non-executable, has caused a shift to code-reuse attacks which redirect program flow to code already present in a program [3].

Mitigations to code-reuse attacks have included stack canaries [4], address randomization (ASLR) [5], and control-flow integrity (CFI) [6]. Stack canaries and ASLR have limitations, and exploits have been demonstrated on both mitigations [7]. CFI is one promising defense. However, CFI seems to suffer from a performance/security tradeoff: full-strength CFI imposes a non-trivial performance overhead. Researchers have proposed coarse-grained CFI defenses that reduce the performance overhead by relaxing the security policy [8], [9], [10], but unfortunately these schemes have been demonstrated to be ineffective [11].

Recent research has suggested using hardware to implement CFI. In 2014, Davi et al. proposed an intriguing hardware-assisted approach with a novel policy for restricting return instructions [12]. Their design keeps track of an "active set" of return sites. Each function call adds the subsequent instruction to the set, and return instructions are only allowed to return to an instruction in the active set. In 2015, Davi et al. refined their design, which they dubbed Hardware-Assisted Flow Integrity eXtension (HAFIX) [13]. They also described two hardware implementations of HAFIX, one for the x86 Siskiyou Peak and one for SPARC LEON3, and showed that both implementations achieve excellent performance.

In this paper, we analyze the security of HAFIX's novel active-set policy for return instructions. This policy provides an interesting intermediate point between coarse-grained CFI and full-strength CFI with a shadow stack. Other researchers have studied coarse-grained CFI (where return instructions are allowed to target any location that follows a call instruction) and fully-precise CFI with a shadow stack (where each return instruction can only return back to the location after the matching call instruction), but the effectiveness of the active-set policy at preventing exploits in real-world programs has not been carefully studied before in the research literature. Our results help understand whether this policy will be sufficient at preventing exploits or if a shadow stack is a requirement for preventing exploits.

To shed light on this question, we examine real-world binaries that had vulnerabilities and evaluate whether HAFIX's active-set policy would have prevented exploitation of those vulnerabilities. We show that this policy is circumventable when an attacker has write access to arbitrary memory. We present three novel attacks, based on the attacker's ability to return to parent code in a child process after a fork, to earlier call sites in functions on the stack, and to the entry function of a program (typically main). Our results can be viewed as adding to the evidence that a shadow stack is a minimum requirement for CFI, and that weaker policies for return instructions are not sufficient.

To the best of our knowledge we are the first to evaluate the active-set policy on real-world programs. Previous research has speculated about potential weaknesses with HAFIX [14], but the extent that these weaknesses are effective at preventing actual exploits has not been studied.

II. Background and Related Work

A. Control-Flow Integrity (CFI)

Control-Flow Integrity [6] is a code-reuse defense that confines the program's execution to be consistent with its control-flow graph (CFG). The program is monitored at runtime to ensure its control flow follows a valid path in the CFG. Any deviation from the CFG produces an exception.

Control transfers can be split into two categories, forward edges (function calls and jumps, including indirect transfers) and backward edges (return instructions). Any CFI implementation must limit both forward and backward edges. Research suggests that at least some forward-edge policies can be enforced efficiently in software [15], but backward-edge policies can be more expensive [16]. This has motivated researchers to examine several different backward-edge policies and to consider hardware support for policy enforcement.

The strongest (most restrictive, most secure) backward-edge policy involves validating return addresses with a shadow stack in protected memory. Each call instruction causes the return address to be pushed to the ordinary stack and to the protected shadow stack; a return instruction validates the return address against the value at the top of the shadow stack. However, shadow stacks impose a significant performance overhead in software, motivating researchers to study weaker, coarse-grained policies for backward edges [17]. One weaker alternative is to omit the shadow stack and check that every return instruction targets the location following some call instruction.

B. HAFIX: Hardware-Assisted Flow Integrity Extension

HAFIX is a hardware CFI implementation for backward edges; the performance overhead is just 2% [13]. Their design [12],[13, § 3] introduces an active-set policy that maintains a set of active functions (functions that are executing on the stack) and restricts returns to only target call-preceded instructions in active functions. This policy is used in the HAFIX x86 implementation.

Under HAFIX, the compiler assigns unique labels to each function, then uses the labels in the following three new instructions: (1.) **CFIBR**: CFIBR is inserted as the first instruction to each function to insert the function's label into the active set; (2.) **CFIRET**: CFIRET is inserted after each call instruction to check that the function's label is in the active set; (3.) **CFIDEL**: CFIDEL is inserted before each return instruction to remove the function's label from the active set. A state machine ensures that every function call and return must be followed immediately by a CFIBR or CFIRET. We illustrate the result of this transformation in pseudocode below:

```
//main - label=0
int main() {
    CFIBR 0 // insert label 0 into the active set
    ...
    foo();
    CFIRET 0 // ensure label 0 is in the active set
    ....
    CFIDEL 0 // delete label 0 from the active set
    return; }
//foo - label=1
int foo() {
    CFIBR 1 // insert label 1 into the active set
    ...
    CFIDEL 1 // delete label 1 from the active set
    return; }
```

In the example above, foo can return to main since main's label, 0, is present in the active set when foo's return instruction is executed. However, main cannot return to foo, as foo's label, 1, is not present in the active set when main returns. Before returning all functions remove their label from the active set to ensure future returns cannot target the function.

HAFIX also includes two additional instructions for handling recursive function calls. HAFIX does not support nested recursion. Our attacks (§ V) do not use recursion.

We emphasize that our results apply only to Davi et al.'s x86 implementation of HAFIX, but not to their SPARC implementation [13]. Their x86 implementation uses the active-set policy for return instructions, while their SPARC implementation uses a full shadow stack for return instructions. Other researchers have studied shadow stacks; this paper focuses solely on the active-set policy.

	Forward-edge policy	
Backward-edge policy	Coarse-grained	Fine-grained
Coarse-grained	broken [18], [20], [11]	broken [19]
Active set	broken	(this paper)
Shadow stack	broken [21], [19]	partially broken [19]

TABLE I
ATTACKS AGAINST VARIATIONS OF CFI

C. Attacks on CFI Implementations

Coarse-grained CFI has been bypassed [18], [11]. Carlini et al. found that fine-grained forward-edge CFI with a weak backward-edge policy (no shadow stack; allow returns to target any call-preceded instruction) can be bypassed [19]. Even a shadow stack (the strongest possible policy for backward edges) can be vulnerable to code-reuse attacks in some cases [19]. Table 1 summarizes attacks on various CFI policies. Although HAFIX was not studied by Carlini et al., their findings imply that coarse-grained forward-edge CFI with an active set for backwards edges can be bypassed, as that policy is strictly weaker than coarse-grained forward-edge CFI with a shadow stack. Previous research has not evaluated the active-set policy for backwards edges with a fine-grained forward-edge policy. We evaluate this combination in this paper.

III. THREAT MODEL

Attacker Goal. Our adversary seeks to use a vulnerability to execute arbitrary code with all program permissions.

Threat Model. In our model an attacker (1.) has full writable control of memory from a vulnerability at one point during program execution, (2.) has access to the program's code, and (3.) can bypass code randomization (ASLR). We verify that an attacker has control of memory in our case studies.

System Assumptions. We assume x86 HAFIX's active-set policy is deployed with the following additional defenses:

1) All indirect calls must follow the most restrictive static CFG for forward edges that still allows all feasible non-malicious execution [15]. (Thus, our attacks apply no matter what policy is applied to forward edges.)
2) Returns are restricted by the active-set policy: they can only target call-preceded instructions in active functions.
3) Data is non-executable and code is non-writable.

IV. ATTACKS AGAINST ACTIVE-SET POLICY

We present three general attacks on the active-set policy.

A. Return-to-Active-Function Attack

The active-set policy allows return instructions to target any active function on the call stack. This property can be used by an attacker to directly return to any function in the call stack, bypassing any code residing in intermediate functions on the stack. These intermediate functions may contain code that is critical for secure execution. We show an example below.

```
int main(int argc, char *argv[]) {
    char *path;
    ...
    foo(path, argv);
    execl(path, argv); // (*)
    ...
}
int foo(char* path, char *argv[]) {
    ...
    vulnerable();
    if (validate(path) != 0) { exit(1); }
    ...
}
int vulnerable(char * argv[]) {
    char buf[1024];
    ...
    memcpy(buf, argv[1], strlen(argv[1]));
    ...
}
```

An attacker can leverage the vulnerability in vulnerable() to overwrite the return address to point to the statement marked (*) in main and overwrite the path variable to refer to a program of their choosing. By returning directly to main, the attacker bypasses the path variable validation that would have caused the program to exit.

B. Return-to-Parent-After-Fork Attack

Event loops and forked processes are common in server software. Servers often have a main process that waits for requests and forks a child process on each new request. In benign execution it is usually not possible to execute code that was designed for the parent process in the child process. The active-set policy allows an attacker who has compromised a child process to return to a function higher in the call stack (in the parent's region of the call stack) and execute code designed for the parent within the child process. This may enable a powerful attack, as often many unsafe library calls occur in code designed to be executed by the parent.

Davi et al.'s x86 HAFIX implementation is intended for bare metal code and does not support multiple processes or fork. This attack is not applicable to that implementation, but it is applicable to any system that uses the active-set policy and supports multiple processes.

C. Back-Call-Site Attack

A consequence of assigning unique labels to functions as opposed to individual call sites is that attackers who control a return address can return to call sites that appear earlier than the original call site if they are in the same function. This enables attackers to reach points in active functions that have already completed execution and are not intended to be re-executed. We show an example below.

```
int main() {
    char path[1024];
    ...
    strcpy(path, "/usr/bin/whoami");
    execl(path, arg);
    ...
    vulnerable();
}
int vulnerable() {
    char buf[1024];
    ...
    memcpy(buf, input, strlen(input));
}
```

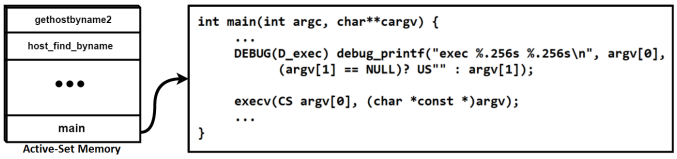

Fig. 1. The active set for the Exim mail server during execution of the vulnerability with the execv call in main.

The vulnerability in the vulnerable function can enable an attacker to execute execl in main with malicious arguments by overwriting the path variable and return address to target the execl in main.

An attacker can also return directly to call sites that occur later in an active function than the original call site, bypassing code occurring in between the original call site and the attacker's chosen call site.

D. Return-to-Main Attack

The back-call-site attack can be combined with a return-to-active-function attack targeting the main function. Programs typically start and complete execution in the main function. As a result, main is marked active throughout the duration of the program, and all code (other than dead code) is reachable via some path starting in main. Suppose an attacker wants to reach code in function g, g is reachable via some path from function f, and main calls f. Then an attacker controlling any return address can always return to any call site in main that precedes the call to f and from there reach g.

V. Evaluation and Case Studies

A. Motivation and Methodology

To understand the applicability of our attacks to real programs we select three programs with reported memory vulnerabilities and attempt to develop attacks on these programs under HAFIX. We select our programs by searching CVE databases for CVEs of open-source programs. We reproduce the vulnerabilities inside gdb to obtain an accurate backtrace and identify which functions are active at the point of the vulnerability. We also use gdb to write to memory and emulate an attacker's control over memory, and to verify that an attacker has full-writable control of memory for all programs.

B. Exim Mail Server

We examine a buffer overflow in the Exim mail server [22]. The vulnerability results from a heap based buffer overflow in the gethostbyname functions in glibc 2.2–2.18.

Control over Memory. A security advisory [22] explains how an attacker can turn the gethostbyname buffer overflow into a write-anything-anywhere primitive. This satisfies our requirement that an attacker has full control of memory.

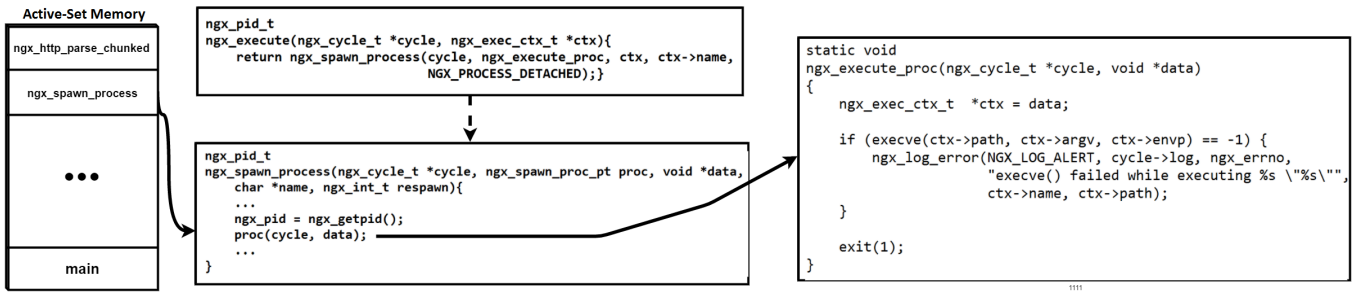

Fig. 2. The active set for Nginx is shown on the left. The ngx_spawn_process function contains a call to ngx_execute_proc which calls execve. ngx_execute_proc is called using the proc variable which can reference ngx_execute_proc when ngx_spawn_process is called from ngx_execute.

Exploitation. We found that an attacker can bypass active-set CFI and perform an exec with an arbitrary command. We successfully spawned a shell while monitoring the program in gdb to ensure the active-set policy is respected. Our attack works by invoking an execv call in the main function. Because main is active when the gethostbyname vulnerability occurs, an attacker can use their control over memory to (1.) overwrite the return address to target the execv and (2.) overwrite the argument that is supplied to execv. Figure 1 shows the active functions at the point where the attacker controls memory and the call-preceded execv call in main.

C. Nginx Web Server

We study a integer overflow vulnerability in Nginx server 1.4.0 that was reported in CVE-2013-2028 [23].

Control over Memory. An integer signedness vulnerability in the decoding stage of Nginx allows an attacker to overflow an integer and trigger a stack-based buffer overflow. The overflow can be used to control arguments of a memcpy call, allowing an attacker to write arbitrary values to arbitrary locations [19]. Memory can be arranged after executing memcpy to return the process to accepting further requests without a crash.

Exploitation. We find an attacker can execute arbitrary code in the presence of active-set CFI. One of the functions in the active set when the memory vulnerability occurs, ngx_spawn_process, invokes a function pointer, proc, which can be overwritten by any value of the attacker's choice. An attacker with control over memory can (1.) overwrite the return address to target the proc function call in ngx_spawn_process, (2.) overwrite the proc function pointer to reference the ngx_execute_proc function, and (3.) overwrite the structure in memory used to hold the arguments for the execve call in ngx_execute_proc. Figure 2 summarizes our exploit and shows the active functions during the exploit. Overwriting the proc function pointer to reference the ngx_execute_proc function does not result in a forward-edge CFI exception as there exists another function, ngx_execute, that sets proc to ngx_execute_proc in non-malicious execution.

D. PHP: Stack Buffer Overflow

We investigate a stack buffer overflow in the sockets extension of PHP 5.3.6 that was reported in CVE-2011-1938 [24].

Control over Memory. An attacker has full writable control of memory in the presence of active-set CFI. A memcpy call in the sockets function of php allows an attacker to trigger a stack overflow. The overflow can (1.) overwrite the arguments to a memcpy call in main and (2.) overwrite the return address to target the memcpy in main. The memcpy call in main is followed by an error condition check that returns when errors are detected. Memory can be overwritten to force a return through this error path to create a write-what-where gadget.

Exploitation. We found that an attacker can execute arbitrary code despite active-set CFI. An attacker can leverage their control over memory to inject a php script of their choosing. The stack overflow occurs during execution of a php script, so the active set contains the required functions for execution of a php script. To execute an arbitrary php script, an attacker (1.) overwrites the existing php script in memory and (2.) overwrites the return address to target the php_execute_script function that executes php scripts.

E. Results

Table II summarizes our results. We believe the attacks we demonstrate are general and can be applied to other software.

VI. Enhancing HAFIX

A. Adoption of Call-Site Labels

To prevent the back-call-site attack, we propose assigning unique labels to individual call sites instead of functions. A compiler would then insert CFIBR instructions immediately before call sites instead of inserting CFIBR instructions at the beginning of functions. This modification restricts returns to target only the original call site in an active function. The attacks we demonstrate on Nginx and Exim servers are not possible under this modification.

B. Deactivation of Parent Function upon Fork

To prevent the return-to-parent-after-fork attack, we propose augmenting fork to clear the child process's active set before executing the child's code. The programs we evaluate do not contain programmer-intended paths in a child process that lead to functions made active in the parent process. We believe this holds true for most programs, however for compatibility we propose implementing this feature as an opt-out compiler option with a default of deactivating active parent functions.

Application	Attack techniques used	Exploitable with active set	Exploitable with shadow stack
Nginx web server	Return-to-parent, back-call-site	Yes	No
Exim mail server	Return-to-main, back-call-site	Yes	No
PHP	-	Yes	Undetermined (No write-what-where gadget found)

TABLE II
A SUMMARY OF OUR ATTACKS. THE SECOND COLUMN INDICATES THE ATTACK METHODS WE USE IN OUR EXPLOITS.

Our proposed compiler option can be modeled after the -fno-stack-protector option used in gcc to disable canaries.

C. Replacement of Active Functions with a Shadow Stack

We were unable to find exploits that work in the presence of a shadow stack for the three programs in our case studies. Therefore we believe the adoption of a LIFO shadow stack will be significantly stronger than an active set. Fortunately Intel plans to add hardware support for shadow stacks in their upcoming Control-flow Enforcement Technology (CET) [25]. Recent research projects, including a successor to HAFIX, also present hardware support for shadow stacks [14], [26].

VII. CONCLUSION

Many variants of CFI have been considered in the research literature. Our work shows that the active set policy for backward edges can be defeated, no matter what forward-edge policy is used. These results suggest that the active set policy is too permissive and CFI needs to use a full shadow stack.

ACKNOWLEDGMENT

We thank Nicholas Carlini and the anonymous reviewers for feedback that improved the paper. This work was supported by the AFOSR under MURI award FA9550-12-1-0040, by Intel through the ISTC for Secure Computing, and by the Hewlett Foundation through the Center for Long-term Cybersecurity.

REFERENCES

[1] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 CWE/SANS top 25 most dangerous software errors," *Common Weakness Enumeration*, vol. 7515, 2011.

[2] PAX-TEAM, "PaX SEGMEXEC documentation," 2004. [Online]. Available: https://pax.grsecurity.net/docs/segmexec.txt

[3] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[4] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security*, 1998, pp. 63–78.

[5] PAX-TEAM, "PaX ASLR (adress space layout randomization)," 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[6] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[7] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *European Workshop on System Security*. ACM, 2009, pp. 1–8.

[8] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attack," in *NDSS 2014*.

[9] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *USENIX Security*, 2013, pp. 447–462.

[10] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.

[11] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security*, 2014, pp. 401–416.

[12] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Design Automation Conference*. ACM, 2014, pp. 1–6.

[13] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: hardware-assisted flow integrity extension," in *Design Automation Conference*. ACM, 2015, p. 74.

[14] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced Control-Flow Integrity," in *ACM Conference on Data and Application Security and Privacy*, 2016, pp. 38–49.

[15] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *USENIX Security*, 2014, pp. 941–955.

[16] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 555–566.

[17] A. Kanuparthi, J. Rajendran, and R. Karri, "Controlling your control flow graph," in *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 43–48.

[18] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *USENIX Security*, 2014, pp. 385–399.

[19] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security*, 2015, pp. 161–176.

[20] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE Symposium on Security and Privacy*, 2014, pp. 575–589.

[21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *ACM Conference on Computer and Communications Security*, 2015, pp. 901–913.

[22] "Qualys security advisory CVE-2015-0235." [Online]. Available: https://www.qualys.com/2015/01/27/cve-2015-0235/ghost-cve-2015-0235.txt

[23] "CVE-2013-2028." Available from MITRE, CVE-ID CVE-2013-2028., May 2 2013. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-2028

[24] "CVE-2011-1938." Available from MITRE, CVE-ID CVE-2011-1938., May 9 2011. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1938

[25] B. Patel, "Intel releases new technology specifications to protect against ROP attacks," Nov 2016. [Online]. Available: https://software.intel.com/en-us/blogs/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks

[26] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *Design Automation Conference*. ACM, 2016, p. 163.