Mitigating Replay Attacks in Cryptographic CFI

Ka Ying Chan
Brown University

Abstract

Control flow hijacking has been a prominent issue in software security since the early 2000s, with the rise of various techniques that leverage memory corruption vulnerabilities in applications. To mitigate control flow attacks, there has been a lot of research on control flow integrity (CFI), aiming to limit malicious transfers of execution flow. Cryptographic CFI is an approach to CFI based on cryptographic MACs. With the use of MACs, protection on control flow objects is hardened. However, cryptographic-based solutions are known to be prone to replay attacks, in which the adversary reuses valid MACs from previous execution.

In this paper, we focus on one specific scenario in which replay attacks would be successful even with previous proposed CFI defense mechanisms. The reason why existing mechanisms fail to eliminate replay attacks is mainly because of a lack of a unique identifier for each control flow object, which provides an opportunity for the adversary to use the MAC of an object for another. This paper proposed a framework, which adds a unique ID to each function and includes it in its return address MAC computation. This unique identifier makes sure that every MAC is only valid for its own function, so the adversary cannot reuse a MAC at an unintended location. The implementation of this proposed framework is proof-of-concept, and there is a discussion on its security and feasibility.

1 Introduction

Control flow hijacking is a form of attack where the adversary exploits vulnerabilities in an application to take control of its execution flow. With the ability to manipulate the control flow, one can redirect the program to execute malicious code or perform unauthorized actions. Some popular techniques used to achieve control flow hijacking include shellcode injections and return-to-libc, which rely on some memory corruption vulnerabilities in the program. To counter these attacks, several solutions have been proposed. For example, there are stack canaries that help detect stack overflows, non-executable memory that prevents shellcode injection, and address space layout randomization (ASLR) that aims to hide actual code location. Although these defense mechanisms are commonly used in today's software, there is a lot of work that proves that they can actually be bypassed [2, 5]. The aforementioned defense mechanisms are not complete solutions to the problem because they do not provide direct protection for control flow objects. As a result, control flow integrity (CFI) has been proposed. It works by generating a control flow graph (CFG) beforehand, which contains all valid execution paths identified, then using the CFG during execution to determine the validity of each control flow transfer [1]. Many different CFI implementations have been proposed [12, 13], and of course there are attempts to bypass these implementations [3, 4].

Among all implementations, cryptographic CFI (CCFI) stands out as it is different from a traditional CFI approach that is based on a set of predefined rules. Instead of using a CFG, CCFI provides protection for control flow objects with cryptographic message authentication codes (MACs) that cannot be generated by the adversary, thus they cannot direct the program execution flow to arbitrary targets [9] . However, the adversary can take the MAC of the desired target from previous execution to replace the MAC of the current target, directing the execution to somewhere else. This type of attack is called the replay attack. Replay attacks work because MACs from previous execution are generated with the secret key, which are considered valid by the application. To limit replay attacks, most cryptographic-based CFI solutions proposed some classifiers for control flow objects to make the computed MACs more specific to them [7, 8, 14]. Nonetheless, we discover that those classifiers are actually not unique enough to distinguish all types of objects. In the situation where two functions reside within the same function, they happen to have the same identifier in proposed implementations, meaning that they have the same MAC for their return addresses, and the adversary can transfer control flow within the same function.

To tackle this specific situation, this paper suggests a frame-

work that provides a unique ID number for each function as its identifier, and includes it in its return address MAC computation. The different IDs are embedded in the binary of the program, in their own function prologues and epilogues. Since the computation of each MAC now involves distinct information, the MACs of the function return address would be different even if they reside in the same function, hindering replay attacks in this scenario. To prove the effectiveness of this proposed framework, a small vulnerable program is designed with the described situation, and instrumentations for MAC computations are added to the binary manually. With the added instrumentations, the program can now recognize a control flow change which is not detected in the unprotected version, so malicious execution transfers by the adversary are successfully prevented. As the implementation is proof-of-concept, the paper also discusses further on the security this framework provides and the feasibility of it on real-world applications.

2 Background

2.1 Cryptographic CFI

In the original CCFI paper, control flow objects (or pointers) are classified into four classes: function pointers, return addresses, method pointers, and vtable pointers [9]. Each class of pointers have their own classifiers. Function pointers are identified by their types and the pointer addresses. Return addresses are identified by their locations on the stack, i.e. frame pointers. Method pointers are identified by their pointer addresses and the method static addresses. And finally, vtable pointers are identified by the addresses they are stored at as the classes. MACs are computed using the AES encryption algorithm with input data of 128 bits (48 bits for the pointer the MAC is protecting + 80 bits for the classifier). In this paper, we focus on the protection on return addresses.
Upon a function entry, the expected return address for the function is stored on the stack at the stack frame pointer. This is when the MAC for the return address is being computed. It takes the return address and the frame pointer at the moment to form the input data, applies AES on the data to generate a MAC, then stores the MAC right next to the return address. On a x84-64 system, pointers can be safely truncated to 48 bits, so there are still 16 bits unused, which makes space for MAC storage. Upon the function return, the frame pointer points to the address the function is going to return to, so it takes the address at that time with the frame pointer to compute a MAC again, then verifies the computed MAC. If the MAC computed at function return is different than the one stored on the stack, that indicates a return address corruption and actions must be taken.

2.2 Replay Attacks

In cryptography, replay attacks in general refers to scenarios when the adversary intercepts a valid data transmission between two parties, records the data, then later retransmits that same data to pretend as the original sender. Since CCFI is cryptographic-based, replay attacks are relevant as well. Consider the following code snippet:

```
vul_func() {
    ...
    foo();
    ...
}

critical_func() {
    ...
    boo();
    ...
}
```

Figure 1 shows the stack layout during the execution of a program that contains the above functions vul_func and $critical\_func$. Let's say $critical\_func$ is called first, and inside $critical\_func$ the function boo is called. The return address of boo is stored at the address $0x1232$. After that, vul_func gets called and inside that foo is called. The return address of foo happens to be stored at the address $0x1232$ as well. Recall in CCFI, return addresses use their locations on the stack as their classifiers. In this case, the two functions boo and foo would have the same classifier, which implies that they have the same MAC. The adversary can capture the MAC of boo in earlier execution, then later on replace the MAC and return address of foo to the captured value. The MAC would still be valid, so the control flow is successfully directed from vul_func to $critical\_func$.

3 Related Work

Replay attack is not a new topic in cryptography, so it is taken into consideration by a lot of cryptographic-based CFI implementations. To mitigate cross-process replay attacks, RAGuard [14] adds the current process ID to MAC computations. To mitigate cross-function replay attacks, PAC It Up [8] assigns each function an ID number and adds the ID of the caller function to MAC computations. Zipper Stack [6] uses the latest MAC stored in a dedicated hardware register to compute the next MAC. PACStack [7] makes it even harder for the adversary as it combines all previous MACs on the stack in MAC computations, so the adversary's action is restricted as they need to change a chain of MACs to the desired value. Other than these MAC-based solutions, there are also other CFI solutions that utilize cryptography, and because of that replay attacks are relevant. Confidaent [11] performs authenticated encryption on both instructions and data, and on top

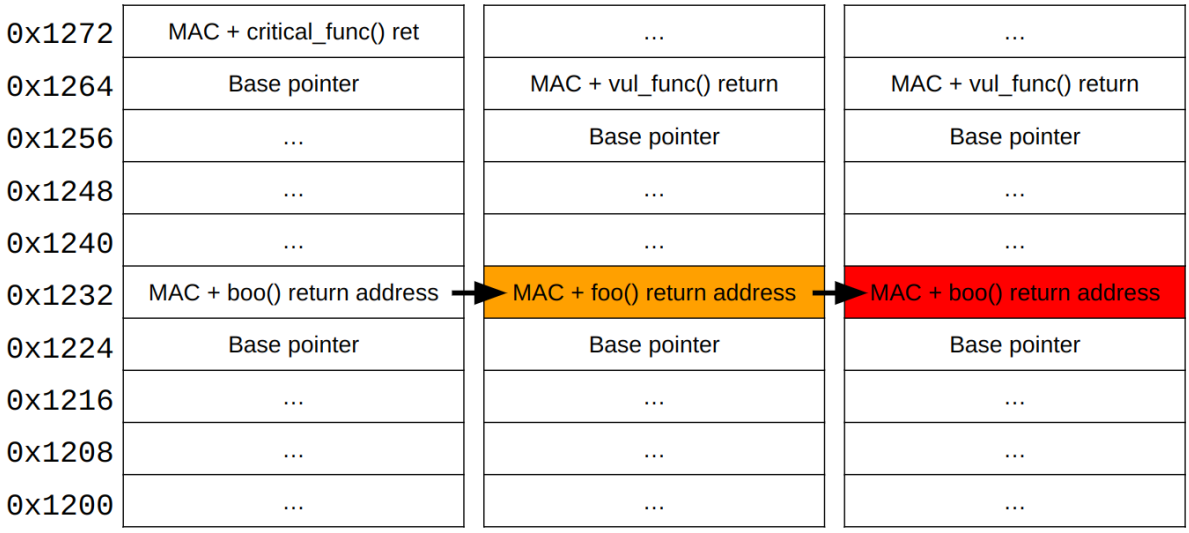

Figure 1: Stack layout during a replay attack. In this scenario, the adversary can replace the MAC of foo() with the MAC of boo() since their return addresses happen to be stored at the same location on the stack during program execution.

of that builds CFI in the form of xored masks with each instruction. EC-CFI [10] uses Intel hardware features to encrypt each function with a different key, and derives decryption key at runtime from CFG comparison.

4 Attack Model

In this paper, we focus on a specific situation when the adversary attempts to divert the control flow within a single function. Consider the following code snippet:

```
vul_func() {
    ...
    foo();
    ...
    boo();
    ...
}
```

Inside the function *vul_func*, two functions *foo* and *boo* are called. Since they reside within the same function, they share the same amount of local variables on the stack. Thus, it is guaranteed that the return addresses of *foo* and *boo* would end up at the same address on the stack during execution. As *foo* and *boo* have the same caller and they are called by the same process, even with the proposed identifiers in Section 3 (PID, caller ID, etc.), they would have the same MAC, and therefore a replay attack is possible.

5 Design

The main objective of the proposed framework is to make sure every return address can be uniquely identified, and the uniqueness is brought over to its MAC computation as well. To achieve this, every function in the program will be assigned a unique ID number, and the MAC will be computed as follows:

```
MAC = hash(message, secret_key)
message = return_address ^ frame_pointer
        ^ caller_id ^ callee_id
```

The input message for the MAC consists of four components, which are the return address we want to protect, and three identifiers to make it unique: (1) frame pointer where the return address is stored; (2) the ID of the parent function who calls the function that has the return address, and; (3) the ID of the function who is called. These components are combined together as one message using XOR. So for instance, for a program that contains the code snippet in Section 4, the input message of the MAC for the function *foo* would be the XOR result of the return address of *foo*, the stack location where the address is stored, the ID of *vul_func* (caller), and the ID of *foo* (callee). To instrument MAC generation and verification, there are three areas that need to be modified - call sites, function prologues, and function epilogues.

5.1 Call Sites

Call sites are where functions are being called. Since the MAC computation requires the ID of the caller, which is something that needs to be passed by the caller to the callee, the call

```
1. movq $<caller_id>, %r15
2. call <callee>
```

Figure 2: Right before calling the callee function (line 2), the caller passes its ID into a register (line 1).

sites need to be modified so that the caller can put its ID in a register, and the callee can retrieve this value from the register during MAC generation and verification. With that being said, we want to make sure that this register being used here would not be used by the program itself - we want to reserve this register for the sole purpose of passing ID numbers to callees. See Figure 2 for pseudocode.

5.2 Function Prologues

Function prologues are right at the entry of functions, before executing any instructions in there. The return address is loaded onto the stack, so MAC generation happens here. At this point, the stack frame pointer would be pointing at the expected return address of the current function, so we can retrieve these two pieces of information. The caller ID can be retrieved from the register the caller used to pass the value. The callee ID is included as part of the computation instructions as a constant. There are instructions that perform XOR for all the data, and a keyed-hash is performed on the combined data using a secret key provided by hardware to compute the MAC. After that, the MAC would be stored in the upper 32 bits of the return address (as return address can be truncated into 32 bits). See Figure 3 for pseudocode.

5.3 Function Epilogues

Function epilogues are right at the exit of functions, before returning to the address stored on the stack. At this point, the stack frame pointer would be pointing at an address that the current function will be returning. Before returning, MAC verification is required. We can retrieve the four components needed just like how we retrieve information in function prologues. One thing that is a little different is that some extra instructions are needed to make sure we are only retrieving the return address, not MAC + return address, as it can affect the computation. Similarly, all the data is combined by XOR, and a keyed-hash is performed on the data with the secret key. After computing the MAC, we will compare this with the MAC stored on the stack in the upper 32 bits of the return address. If the MACs do not match, then we will not do the return, but instead jump to a handler to handle this address corruption. The handler construction is of choice. See Figure 4 for pseudocode.

```
1. movq %rsp, %r12
2. movq (%rsp), %r13
3. xor %r13, %r12
4. xor %r15, %r12
5. xor $<callee_id>, %r12
6. hash %r12, $<secret_key>
7. shl $32, %r12
8. xor %r12, (%rsp)
```

Figure 3: Line 1-2 retrieve the frame pointer and the return addresses respectively. Line 3-5 combine all components using XOR. Line 6 computes the MAC. Line 7-8 store the MAC in the upper 32 bits of the return address.

```
1. movq %rsp, %r12
2. movq (%rsp), %r13
3. and $0xffffffff, %r13d
4. xor %r13, %r12
5. xor %r15, %r12
6. xor $<callee_id>, %r12
7. hash %r12, $<secret_key>
8. shl $32, %r12
9. movq (%rsp), %r13
10. mov $0xffffffff00000000, %r14
11. and %r14, %r13
12. cmp %r12, %r13
13. jne handler
14. xor %r12, (%rsp)
```

Figure 4: Line 1-2 retrieve the frame pointer and the return addresses respectively. Line 3 clears the upper part of the return address to exclude the stored MAC. Line 4-6 combine all components using XOR. Line 7 computes the MAC. Line 8-10 get the computed MAC and the stored MAC ready for comparison. Line 11-12 compare the MACs. If the comparison fails, jump to the handler on line 13. Line 14 undoes the MAC stored on the stack, gets the return address ready for use.

6 Proof-of-concept Implementation

To demonstrate the proposed framework, a small vulnerable program is developed. Just like the attack model described in Section 4, two functions are called inside the *main* function. In one of the callee function, it attempts to read beyond the size of the buffer, which gives the adversary an opportunity to overwrite the return address with a buffer overflow.
To perform the instrumentations, the vulnerable program written in C is converted into x86-64 assembly code, and the modifications mentioned in Section 5 are added manually at each function's call site, prologue, and epilogue. For the sake of demonstration, the hash step is replaced by an XOR operation. This is because it is difficult to perform hashing with few assembly instructions. In real-world scenarios, a cryptographic keyed-hash algorithm should be used. See the Availability Section for the program and complete code implementation.

7 Discussion

7.1 Security

To see if the framework achieve its expected security goal, we construct a buffer overflow attack on both the original version and the instrumented version of the program. For the original unprotected version, we overwrite the return address of *message* with the return address of *order* obtain from previous execution. For the instrumented version, we overwrite the MAC and the return address of *message* with the MAC and the return address of *order* from previous execution.
From Figure 5, we can see that the string "Ordered" is printed twice, which means the execution flow is reverted back to the return address of the function *order*. From Figure 6, the string "Ordered" is only printed out once, and when we try to revert the execution flow, MAC verification fails, hence we end up jumping to the handler. The results demonstrate that the proposed framework is able to protect return addresses and prevent malicious control flow transfers within a single function.
The framework is effective due to the uniqueness of the MACs. Even when multiple functions reside within the same function, they are all provided with distinct IDs, which differentiates their MAC computations. Since the function IDs exist in the binary within the instructions, the adversary cannot modify that (unless they have arbitrary write privilege that can modify the program data, which is way harder to gain), thus cannot destroy the uniqueness of MACs.

7.2 Feasibility

As the implementation is proof-of-concept, we will discuss how feasible it is to apply adequate changes to the framework for practical application. First, the framework requires every

```
cslab00a ~/vul_prog $ ./vprog_0 < expl_0
Ordered
Ordered
```

Figure 5: Caption

```
(gdb) r < expl_2
Starting program: /ifs/CS/replicated/home/kchan40/vul_prog/vprog_2 < expl_2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Ordered

Program received signal SIGSEGV, Segmentation fault.
0x0000000096d9004 in handler ()
```

Figure 6: Caption

function in the program to have a unique ID. This can be done by the compiler iterating through all functions and generating a random 4-byte or 8-byte value for each. In fact, it is done in PAC It Up [8] already. Second, the instrumentations can be done manually given that the program is relatively small, but in real life, software can contain a lot more functions, so there needs to be a way to automate the instrumentation process. This can be done with LLVM passes which modifies the way the compiler generates function prologues and epilogues, and locates the call sites to add extra instructions. Lastly, the keyed-hash algorithm used in MAC computation are of choice, but there are requirements it needs to fulfill. The security of CCFI relies on the fact that hashed MACs cannot be reversed, so the hash algorithm used needs to be cryptographic. In addition, for ideal performance, a low-latency algorithm is preferred.

8 Conclusion

In this paper, we proposed a framework which aims to mitigate replay attacks in CCFI. We focus on the situation when the adversary attempts to change the program execution flow within one single function. CCFI secures return addresses by computing MACs for the addresses together with their classifiers. To harden protection, we increase the uniqueness of MACs by assigning all functions with distinct IDs and include the frame pointer, the caller ID, and the callee ID in MAC computations. To perform MAC generation and verification, instrumentations need added at all call sites, function prologues, and function epilogues. The proof-of-concept implementation demonstrates that the proposed scheme is able to achieve its intended security goal. The framework is also feasible on real-world software with corresponding modifications.

Availability

The vulnerable toy program and the implementation in x86-64 assembly code can be found in this GitHub repository: https://github.com/kchan2/csci2951u_replay

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery. https://doi.org/10.1145/1102120.1102165.

[2] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. pages 227–242, 05 2014. https://www.ieee-security.org/TC/SP2014/papers/HackingBlind.pdf.

[3] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow bending: On the effectiveness of Control-Flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

[4] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of Coarse-Grained Control-Flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi.

[5] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016. https://ieeexplore.ieee.org/document/7783743.

[6] Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng. Zipper stack: Shadow stacks without shadow, 2020. https://arxiv.org/pdf/1902.00888.

[7] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. PACStack: an authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 357–374. USENIX Association, August 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[8] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association. https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

[9] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. CCS '15, page 941–951, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2810103.2813676.

[10] Pascal Nasahl, Salmin Sultana, Hans Liljestrand, Karanvir Grewal, Michael LeMay, David M. Durham, David Schrammel, and Stefan Mangard. Ec-cfi: Control-flow integrity via code encryption counteracting fault attacks, 2023. https://arxiv.org/pdf/2301.13760.

[11] Olivier Savry, Mustapha El-Majihi, and Thomas Hiscock. Confidaent: Control flow protection with instruction and data authenticated encryption. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 246–253, 2020.

[12] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

[13] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 927–940, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2810103.2813673.

[14] Jun Zhang, Rui Hou, Junfeng Fan, Ke Liu, Lixin Zhang, and Sally A. McKee. Raguard: A hardware based mechanism for backward-edge control-flow integrity. In *Proceedings of the Computing Frontiers Conference*, CF'17, page 27–34, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3075564.3075570.