Control-Flow Carrying Code

Yan Lin
Singapore Management University
yanlin.2016@smu.edu.sg

Xiaoyang Cheng
Nankai University, China
chengxiaoyangcxy@outlook.com

Debin Gao
Singapore Management University
dbgao@smu.edu.sg

ABSTRACT

Control-Flow Integrity (CFI) is an effective approach in mitigating control-flow hijacking attacks including code-reuse attacks. Most conventional CFI techniques use memory page protection mechanism, Data Execution Prevention (DEP), as an underlying basis. For instance, CFI defenses use read-only address tables to avoid metadata corruption. However, this assumption has shown to be invalid with advanced attacking techniques, such as Data-Oriented Programming, data race, and Rowhammer attacks. In addition, there are scenarios in which DEP is unavailable, e.g., bare-metal systems and applications with dynamically generated code.

We present the design and implementation of Control-Flow Carrying Code (C^3), a new CFI enforcement without depending on DEP, which makes the CFI policies embedded safe from being overwritten by attackers. C^3 embeds the Control-Flow Graph (CFG) and its enforcement into instructions of the program by encrypting each basic block with a key derived from the CFG. The "proof-carrying" code ensures that only valid control flow transfers can decrypt the corresponding instruction sequences, and that any unintended control flow transfers or overwritten code segment would cause program crash with high probability due to the wrong decryption key and the corresponding random code bytes obtained. We implement C^3 on top of an instrumentation platform and apply it to many popular programs. Our security evaluation shows that C^3 is capable of enforcing strong CFI policies and is able to defend against most control-flow hijacking attacks while suffering from moderate runtime overhead.

KEYWORDS

Control-flow integrity; Instruction-set randomization; Control-flow hijacking; Secret sharing

ACM Reference Format:
Yan Lin, Xiaoyang Cheng, and Debin Gao. 2019. Control-Flow Carrying Code. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3321705.3329815

1 INTRODUCTION

Control-Flow Integrity (CFI) introduced by Abadi et al. [1] provides attractive security features because of its effectiveness in defending against injected and code-reuse attacks (CRA), including advanced attacking techniques like Return-Oriented Programming (ROP) [5, 9, 37]. CFI techniques typically compute a Control-Flow Graph (CFG) statically [48, 50] or dynamically [28, 29] and instrument the binary code by adding checks before indirect branches. These checks ensure that any control transfers during execution never deviate from the CFG, even when the program is under attack.

An assumption made in most existing CFI approaches, including coarse-grained [48, 50] and fine-grained [28, 29, 44] ones, is that read-only data and code sections cannot be overwritten by attackers. For example, CFI proposed by Abadi et al. [1] relies on read-only tags inside the code segment, and numerous approaches use a table structure (made read-only) to store valid targets of indirect branches [28, 29, 50]. However, there are scenarios in which such page-level protection is unavailable, e.g., bare-metal systems which do not have a Memory Management Unit (MMU) and applications with dynamically generated code. Moreover, data race attacks [51], Rowhammer attacks [6] and Data-Oriented Programming (DOP) [21] have demonstrated that it is possible to gain arbitrary memory read and write access.

In this paper, we explore the possibility of enforcing CFI in the absence of such an assumption. Specifically, we look into encoding CFI policies into the machine instructions directly without relying on policies specified in additional data structures (i.e., the read-only table structures in existing CFI approaches) or inserting CFI checks into the code segment. The general idea is to embed a statically constructed CFG to the instructions, execution of which is conditioned on correct control flows. In this way, each intended instruction will carry a proof that can validate the control flow transfer. Unintended instructions cannot be executed as the proof in these instructions are not correct. Intuitively, instructions with CFG embedded can be seen as a proof-carrying code [27], where this proof is self-contained in the code rather than being encoded into a separate table. The challenge is how to embed the CFG into the instructions and how to correctly execute them at runtime.

To this end, we present Control-Flow Carrying Code, C^3, a general CFI method that embeds CFG into instructions and performs CFI checks automatically. We stress that with C^3, instructions in a program themselves carry the CFG information and its enforcement without relying on any additional data structure. Inspired by the framework of Instruction-Set Randomization (ISR) [22] where instructions of a program are encrypted with a secret key, C^3 encrypts each basic block in the program with a key derived from the CFG. More specifically, the key is derived from (the addresses of) valid callers of the basic block to ensure correct control flow transfers. At runtime, only the valid callers (their addresses) could enable the correct reconstruction of the key to decrypt the basic block. In this way, C^3 manages to embed and enforce CFI in the program instructions.

However, two challenges remain in making C^3 practical. First, a basic block may have multiple valid callers. These valid callers have different addresses, while the successor block has to be encrypted with a single key. How does C^3 enable the reconstruction of the single correct key by all the valid control flow transfers? To address this challenge, C^3 utilizes the secret sharing scheme [38] to make the key shared among valid callers.

Although secret sharing helps solve this important challenge at a high level, we encounter more challenges in its application in our setting. For example, secret sharing requires that *all (a variable number of)* callers of the basic block be on the same secret sharing curve. The implication is that once we have the curve fixed, addresses of these callers can no longer take arbitrary locations but have to be on the secret sharing curve determined. This imposes extra challenges in laying basic blocks in the text segment of the program. To address this, we design an algorithm to redistribute basic blocks to positions satisfying the secret sharing curve.

We have implemented C^3 that consists of two components, one that performs binary rewriting to redistribute and encrypt basic blocks, and the other as a plug-in to an existing instrumentation platform to assist runtime execution of the rewritten executable. We apply C^3 to a number of server and non-server applications on the Linux platform. Our experimental results demonstrate that C^3 effectively defends against control-flow hijacking attacks and at the same time, introduces realistic runtime performance overhead for server applications comparable to existing Instruction-Set Randomization (ISR) implementations on the same instrumentation platform. Similar to the arguments in ISR systems, we believe that such overhead could be significantly reduced with a hardware-assisted platform.

2 RELATED WORK

2.1 Control-Flow Integrity

CFI [1] forces control flow transfers in the program to follow the policy presented by the CFG. It can be classified into instrumentation-based and hardware-assisted ones.

2.1.1 Instrumentation-based CFI. This category of enforcement typically inserts CFI checks before each intended indirect branch during compiling or installation to consult a CFI checking module, e.g., MCFI [28] and πCFI [29]. Forwarding CFI [44] inserts checks before all forward-edge control-flow transfers to check whether the function signatures (return type and the number of arguments) are correct. The original CFI proposal [1] and its variants rewrite each indirect branch transfer in the binary to validate the ID of the control transfer target. BinCFI [50] instruments indirect branch transfers to jump to address translation routines that determine the targets of the transfers. CCFIR [48] instruments indirect transfers to limit them to flow only to a "springboard" section. O-CFI [26] combines fine-grained randomization and CFI by inserting checks before each indirect branch to check whether its target is within a valid boundary.

C^3 falls into this category using binary rewriting. The fundamental difference is that C^3 has the CFG and CFI policy embedded into every machine instruction without relying on additional metadata, and therefore works effectively without the assumption of keeping such metadata read-only.

2.1.2 Hardware-assisted CFI. Other CFI techniques make use of hardware features to record branch transfers where CFI checks are triggered. For example, kBouncer [32] and ROPecker [11] use Last Branch Record (LBR) which records a small number of the most recent control transfers with minimal overhead. Recent proposals show that an adversary can break heuristics used in these approaches by using long gadgets and launching "history-flushing" attacks [8]. To overcome the limitation in the size of LBR, some [17, 19, 20, 24] make use of Intel Processor Trace (IPT) to enforce CFI. However, control flow transfers recorded by IPT are in a compressed form and decoding it results in large performance overhead.

2.2 Instruction-Set Randomization

Instruction-Set Randomization (ISR) was initially proposed to fight against code-injection attacks [2, 16, 22, 34]. It encrypts instructions and provides a unique instruction set to every program. Injected code would first be decrypted to a random byte sequence and result in illegal instructions executed. Recently, researchers looked into using ISR to defend against CRA. Scylla [43] encrypts every instruction in a basic block with respect to its predecessor to defend against CRA that jumps to the middle of a basic block. Polyglot [40] encrypts the binary at the page granularity to defend against JIT-ROP [41]. SOFIA [15] uses ISR to enforce CFI for cyber-physical systems with instructions at a fixed length of 32-bit, and is probably the closest to our proposed technique. It enforces CFI via an integrity check of instruction blocks where the Message Authentication Code (MAC) is encrypted. However, their limitations include, e.g., supporting up to two callers of a basic block and requiring hardware assistance. C^3, on the other hand, does not have such limitations. DynOpVm [10] shares a similar idea with C^3 on using secret sharing for the purpose of defending against frequency analysis attacks on VM-based obfuscators. However, it leaves the original callers as plain-text in the executable and cannot defend against control-flow hijacking attacks.

2.3 Shamir's Secret Sharing

Secret sharing refers to the sharing of s among n parties so that only when the parties bring together their respective shares can the secret be reconstructed. When counting on all participants to combine the secret is impractical, Shamir [38] introduced a (t, n) threshold scheme which allows the secret to be shared among n participants while any t (but not fewer than t) of them are sufficient to recover the secret.

The essential idea of Shamir's secret sharing relies on the fact that we can fit a unique polynomial of degree $(t - 1)$ to any set of t points that lie on the polynomial curve. For example, two points are sufficient to define a straight line, and three points are sufficient to define a parabola. The first coefficient is usually used as the secret.

Secret-sharing schemes are important building blocks in cryptography in Byzantine agreement, threshold cryptography, access control and attribute-based encryption. In this paper, C^3 uses it to enforce CFI so that only valid transfers can reconstruct the secret.

3 OVERVIEW OF C^3

3.1 Threat Model and Assumptions

The proposed defense, C^3, is aimed to protect a vulnerable application against control-flow hijacking attacks such as ROP attacks. The application to be protected may have some vulnerabilities that can be leveraged by an attacker to inject an exploit payload (code or data). We focus on user-space attacks leaving kernel exploits out of our scope. Specifically, we assume that:

- The target program does not contain self-modifying or dynamically-generated code.
- Attackers could use attacks to bypass W$\oplus$X, such as Data-Oriented Programming [21], data race [51] and Rowhammer attacks [6], and could exploit information disclosure vulnerabilities to investigate the victim's process memory.
- Since the current implementation of C^3 is on top of the popular instrumentation platform Pin, we assume that attackers do not target Pin in their attacks and the partial memory segment managed by Pin (e.g., the code cache) is secure. This assumption can be removed if C^3 is supported by native hardware.

3.2 Embedding CFG to Instructions

Rather than consulting additional information stored in read-only memory, we propose to embed CFG to instructions. An instruction with CFG embedded can check the integrity of the control flow automatically during the execution without querying other data structures. In particular, C^3 embeds the CFG information by encrypting each basic block (an idea inspired by ISR [22]) with a key generated from control flow dependent information. At runtime, the basic blocks are decrypted using a key reconstructed from the actual control flow transfers taken. Only when the correct control flow paths are taken will the instructions be decrypted correctly.

In Figure 1, each node represents an encrypted basic block while edges indicate control flows. The solid edges represent valid control flows with S_i indicating the encryption key for basic block i. S_3 and S_4 are generated according to the valid control flow path $< 1, 3 >$, $< 2, 3 >$ and $< 3, 4 >$. When there is an invalid control flow transfer from node 2 to 4 denoted by the dotted edge, a wrong key S_4' is constructed which would result in illegal instruction faults.

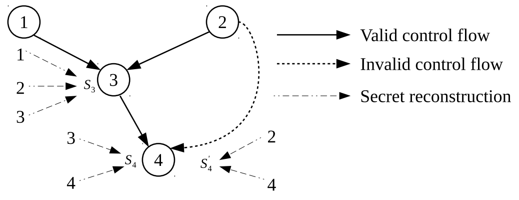


Figure 1: Example of secret reconstruction. Each circle represents a basic block of instructions.

Although the idea sounds straightforward, there are multiple design questions and challenges. First, what information do we use to generate the key? Such information needs to be both statically and dynamically available, and it shall allow enforcement of CFI. How do we deal with basic blocks involved in multiple control flows, which may lead to different keys constructed dynamically? A simple solution is to insert (constant) shares at the caller and callee with which the secret can be reconstructed at runtime. However, such an approach does not provide Control-flow integrity because an attacker can reuse the share at other caller sites.

Our solution is to use the addresses of the branch transfer instruction and its target as the shares since they capture the control transfer information precisely. To deal with basic blocks involved in multiple control flows, we use basic block redistribution and secret sharing [38] to encode the key. Figure 2 shows an overview of C^3, consisting of two components.

- **Embedding CFG.** C^3 transforms branch transfer instructions (indirect branches, conditional jumps, and direct calls) to have a secret share embedded, and then redistributes basic blocks to specific addresses so that all valid callers are on the same secret sharing curve. Finally, basic blocks are encrypted with the secret.
- **Enforcing CFI.** Whenever the program attempts a control transfer, C^3 obtains the caller and callee addresses and reconstructs the key to decrypt the callee basic block before control transfer takes place.

4 DETAILED DESIGN OF C^3

C^3 takes as input a binary executable (without source code) and outputs a modified executable with CFG embedded and CFI enforced.

4.1 Secret Sharing and Challenges

As discussed in Section 3, our approach of embedding CFG into instructions is to encrypt a basic block and to enable decryption with any correct control transfer. For a basic block with multiple callers, we can imagine that every valid caller shall contribute to the encryption key; however, in a concrete execution, only one valid caller is involved and the decryption key is reconstructed. This is where the idea of secret sharing comes to our design — only part of the ingredients

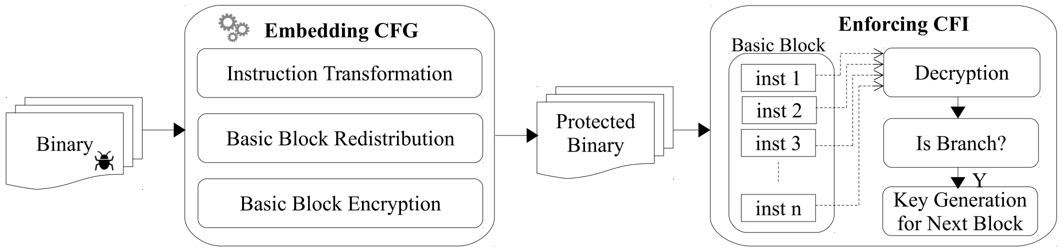


Figure 2: System overview of C^3.

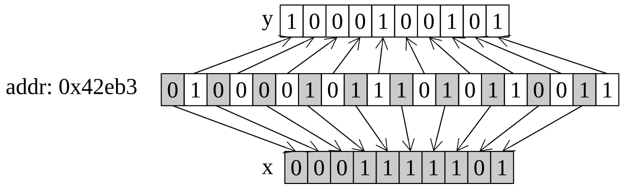


Figure 3: Extracting x and y for address 0x42eb3. C^3 obtains x and y from the lower-order odd- and even-index bits of an address.

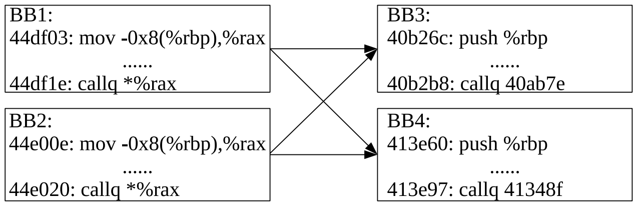


Figure 4: Multiple callers to multiple callees. BB1 and BB2 are callers to callee BB3 and BB4.

of the secret key is needed for correct reconstruction. C^3 uses Shamir's approach [38] due to its simplicity.

The next question is the degree of the secret sharing equation. A general guideline is to keep it small to minimize overhead. We can use a degree of two with the source and target addresses of the control transfer — the minimum information to fully describe a control transfer. However, this runs into the risk of a code pointer disclosure exploit that discloses both addresses and allows an attack to decrypt the basic block. To counter such an attack, we add one random value (called the master key) which is unknown to the attacker to construct the secret key. Specifically, we use a degree of three, with the secret sharing equation

$$y = a_0 \ + \ a_1 x + \ a_2 x^2 \ (mod\ M) \tag{1}$$

where a_0 is the secret key for encryption and decryption, and x, y are k-bit coordinates extracted from the source and target addresses and the master key. C^3 obtains x and y from the lower-order odd- and even-index bits of an address (see Figure 3 for an example). Reconstruction of the secret follows Equation 2 with $x = 0$.

$$y = \sum_{i=1}^{3} y_i \prod_{1 \le j \le 3, j \ne i} (x - x_j)(x_i - x_j)^{-1} \ (mod\ M) \tag{2}$$

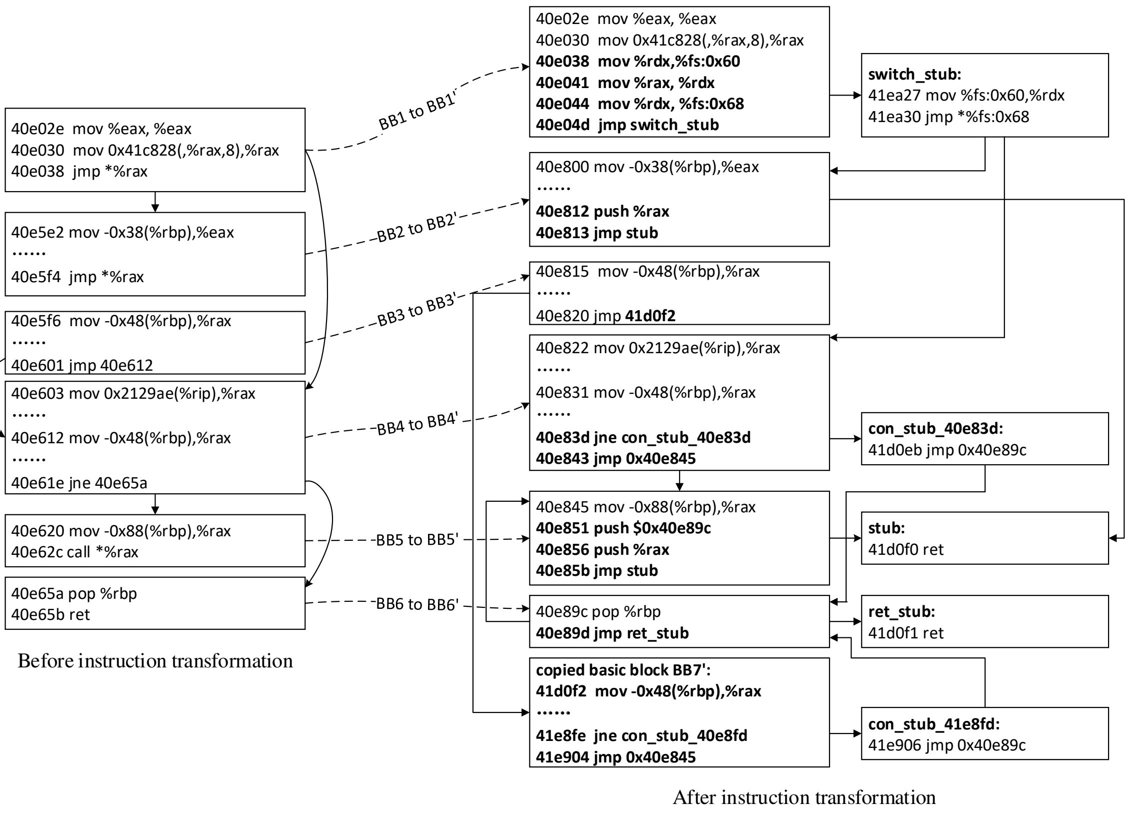
To support a basic block with multiple callers, we can simply relocate the caller instructions so that they all lie on the parabola. However, a more challenging issue is to support a set of basic blocks with the same (set of) callers. Figure 4 shows an example with $BB3$ and $BB4$ having the same set of callers $BB1$ and $BB2$. Following the secret sharing design we outline above, the two parabolas for $BB3$ and $BB4$ will have three intersection points — the master key, $BB1$, and $BB2$; however, different parabolas could have up to two intersections only. Therefore, C^3 not only needs to relocate the basic blocks to move them onto specific parabolas, but also needs to perform some special transformations to control transfer instructions; see the next subsection.

4.2 Instruction Transformation

In fact, the complexity shown in Figure 4 is not the only one that C^3 needs to handle.

- **CPX1: Multiple callers to multiple callees.** In such cases, secret sharing curves for the callees have three or more intersections (including the master key), which is not possible for parabolas. We add an intermediate block between the callers and callees so that multiple callees now have a single caller.
- **CPX2: Basic blocks that are not freely movable.** Examples of such blocks include targets of `ret` instructions which must follow the `call` instruction, and the default branch of conditional instructions which must follow the conditional branch instruction. They cannot be moved freely to other locations due to the implicit control flow. Our strategy is to transform the implicit control flows into explicit ones.
- **CPX3: Basic blocks with multiple entries.** Multiple entries will lead to different keys derived for the same basic block. Our strategy is to break it up into multiple basic blocks, each of which has a single entry.

In the rest of this subsection, we use an example (Figure 5) to explain how C^3 solves these complexities. Note that the transformation is via binary rewriting without source code of the program.



```
40e02e  mov %eax, %eax
40e030  mov 0x41c828(,%rax,8),%rax
40e038  jmp *%rax

40e5e2 mov -0x38(%rbp),%eax
……
40e5f4  jmp *%rax

40e5f6  mov -0x48(%rbp),%rax
……
40e601 jmp 40e612

40e603 mov 0x2129ae(%rip),%rax
……
40e612 mov -0x48(%rbp),%rax
……
40e61e jne 40e65a

40e620 mov -0x88(%rbp),%rax
40e62c call *%rax

40e65a pop %rbp
40e65b ret
```

```
40e02e  mov %eax, %eax
40e030  mov 0x41c828(,%rax,8),%rax
40e038  mov %rdx,%fs:0x60
40e041  mov %rax, %rdx
40e044  mov %rdx, %fs:0x68
40e04d  jmp switch_stub

40e800 mov -0x38(%rbp),%eax
……
40e812 push %rax
40e813 jmp stub

40e815  mov -0x48(%rbp),%rax
……
40e820 jmp 41d0f2

40e822 mov 0x2129ae(%rip),%rax
……
40e831 mov -0x48(%rbp),%rax
……
40e83d jne con_stub_40e83d
40e843 jmp 0x40e845

40e845 mov -0x88(%rbp),%rax
40e851 push $0x40e89c
40e856 push %rax
40e85b jmp stub

40e89c pop %rbp
40e89d jmp ret_stub

copied basic block BB7':
41d0f2  mov -0x48(%rbp),%rax
……
41e8fe  jne con_stub_40e8fd
41e904 jmp 0x40e845
```

```
switch_stub:
41ea27 mov %fs:0x60,%rdx
41ea30 jmp *%fs:0x68

con_stub_40e83d:
41d0eb jmp 0x40e89c

stub:
41d0f0 ret

ret_stub:
41d0f1 ret

con_stub_41e8fd:
41e906 jmp 0x40e89c
```

Figure 5: An example of instruction transformation by C^3. C^3 rewrites indirect and conditional branch transfer instructions to jump to a stub (bold), and the stub will then jump to the real targets. C^3 transforms basic blocks with multiple entries ($BB4$) by copying each entry to a new address ($BB4'$ and $BB7'$).

4.2.1 Transforming indirect call and indirect jump instructions. C^3 transforms an indirect call instruction into two `push` instructions (one to save the return address and the other to save the target address onto the stack) followed by a `jmp` instruction (jumping to a common stub); see $BB5$ and $BB5'$ in Figure 5. The stub block has a single `ret` instruction.

Although this simple transformation solves CPX1, it potentially enforces a relaxed CFI policy since multiple control transfer targets now go through the same common stub block. We stress that the same policy is used by existing coarse-grained CFI methods [48, 50]. Moreover, C^3 increases the difficulty of a stealthy attack since the valid targets are now encrypted. We could use a more complicated secret sharing curve to enforce a finer-grained policy, but C^3 chooses this solution due to its simplicity and enforcing a CFI policy not less secure than existing work. Note that a byproduct of pushing the return address on the stack (the first `push` in $BB5'$) is a solution to CPX2, as the return site can now be freely moved (explained later in the next subsection).

Indirect jumps are handled in the same way, except that we only need one `push` instruction since there is not a return address, e.g., $BB2$ in Figure 5. Additional complexity arises here when the indirect jump was generated due to `switch/case` statements during compilation, where local variables are sometimes accessed via `%rbp` directly without changing `%rsp`. In such cases, we cannot simply push the target address of the indirect jump onto the stack because doing so would overwrite the local variables. Instead, we make use of thread local storage to store the target; see the indirect jump in $BB1$ of Figure 5. In order to transform an indirect jump `jmp *0x8(%rax)` (the target is the address in memory) while having the same switch stub with `jmp *(%rax)`, we simply move the target of them to the temporary register `%rdx` as shown in $BB1'$.

4.2.2 Transforming conditional jump instructions. Conditional jumps usually have a fall-through branch to the instruction that immediately follows, forming an implicit control transfer (CPX2). We turn this into an explicit one by inserting a direct jump instruction as in $BB4'$ of Figure 5. Note that similar to indirect jumps, conditional jumps may be followed by multiple callees (CPX1); that is why we also add a stub block as shown in $BB4'$ of Figure 5.

4.2.3 Transforming return instructions. Handling return instructions (CPX1) is simple as we only need to add a common stub which then returns to the call site; see $BB6'$ in Fig 5. We can enforce a finer-grained CFI policy by classifying functions into indirectly-called and directly-called ones, of which the latter does not need the additional stub block to be inserted since any two of them cannot return to the same call site. We leave this security improvement as our future work.

4.2.4 Transforming basic blocks with multiple entries. The multiple entries of a basic block correspond to different sets of ingredients for the secret reconstruction, and therefore will result in different keys (CPX3). C^3 handles this by copying each entry (and subsequent instructions in the block) to a new address and updating the corresponding control flow instructions to the new addresses. For example, $BB4$ in Figure 5 has two entries, `0x40e603` and `0x40e612`, respectively. C^3 copies the second entry to a new address ($BB7'$) and directs the control flow from $BB3'$ to it.

4.3 Basic Block Redistribution

Redistributing basic blocks so that all callers of a control transfer are on the same secret sharing curve is an interesting and non-trivial problem. One can consider it as a directed graph traversal in which whenever a node is traversed, we pick a parabola and ensure that all its callers are on it by moving some or all the callers. However, if the traversal is not carefully designed, we could get into a failure where a node that has been previously moved on a parabola now needs to be moved again to satisfy another parabola — a mission impossible. Therefore, the key is to design a directed graph traversal algorithm that minimizes or eliminates such a risk.

C^3 uses a customized Depth First Search (DFS) algorithm. Intuitively, DFS fits our requirement in that it explores a branch to its ultimate leaves before backtracking or stepping into a new branch, which avoids unnecessary moving of caller nodes of branches already unexplored. We customize it with a "look ahead" capability which switches to another nearby branch when continuing exploring the current branch will get into a "mission impossible" case.

As shown in Figure 6 where shaded nodes denote those that had previously been moved (and therefore cannot be moved again) and hollow ones otherwise, continuing to traverse node A would run into a failure mode since node B will have two caller nodes fixed, making it impossible to find a parabola for node B (it already has three points determined including the master key). In this scenario, our "look ahead" function will traverse the sub-branch of node B before going back to traverse node A. This "look ahead" function is also used to decide the starting point. By default, C^3 picks a node with the largest number of callers as the starting point, and then uses the "look ahead" function to check whether this starting point and one of its callers target the same basic block. If they do, C^3 uses this basic block as the starting point. The detailed algorithm is shown in Appendix A.

Specifically, for a callee to be processed, we first check whether there is a prior basic block using the "look ahead" mechanism described above. Then, for each callee to be processed, we check whether there exists any of its callers that has a fixed address. If there is, we use this caller (with a fixed address) to determine the parabola; otherwise we randomly choose a caller to determine the parabola. The

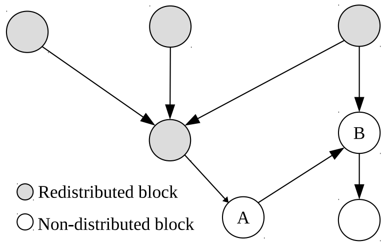


Figure 6: "Look ahead" DFS search. Traversing node A would result in node B having two caller nodes fixed. The "Look ahead" function of C^3 will traverse the sub-branch of node B first.

special and additional processing here is that for each (caller or callee) address, we need to check whether it will have the same x value with its callee, caller or the master key, since the same x value could result in a failure in calculating the inverse to compute the secret as in Equation 2. We generate a new random address if when detecting this problem.

Once a parabola is determined, we move all the callers onto it by randomly choosing an unused coordinate on the curve, which determines the new addresses of the callers. After that, we use the DFS approach to process other basic blocks.

Since the redistribution of basic blocks might turn a short jump instruction into a long jump, C^3 turns every direct jump into a long jump (with a four-byte displacement) before the redistribution process starts.

4.4 Encryption and Decryption

Before we present details of C^3 in encrypting a basic block, we note that completely separating code from data into different sections is a prerequisite for our encryption to work. This is because the encryption of any data may disrupt program execution when it is not decrypted at runtime. Fortunately, many linkers are configured to ensure such separation, and compiler optimizations like jump tables are also typically moved to a non-code section. C^3 does not include PLT calls in its protection as doing so will result in `.plt` section containing non-continuous addresses due to basic block redistribution (see the previous subsection), which in turn makes it impossible for the dynamic loader to update addresses in the Global Offset Table (GOT).

C^3 uses XOR as the encryption function due to its simplicity. The reconstructed secret s from secret sharing is used as the seed to a pseudo-random function generator to generate a 16-bit key for encryption. The length of the secret s is a configurable parameter which has an upper bound of 16 because going beyond that may result in distance between two instructions greater than 2^{31}. To fight against memory disclosure attacks that attempt to compromise the master key, C^3 stores the master secret key outside of the binary into a database file, an approach used in some ISR approaches [34]. We note that C^3 could also perform load-time encryption on the basic blocks using a session key (replacing the master key) to further improve security [2, 31]. Also note that when the binary rewriting process is performed remotely, we could make use of remote attestation [13] to securely distribute the master key. We leave both ideas as our further work.

4.5 Transitioning from Unprotected to Protected Code

C^3 supports partial protection of a program that contains protected (CFG embedded) and unprotected (e.g., system or third-party libraries without CFG embedded) code. However, the transitioning from unprotected to protected code needs special attention since CFI checks will fail as the caller is not on the secret sharing curve of the callee. Such transitioning typically occurs in two scenarios.

- **Returning to protected code.** This happens when protected code calls an external library function and subsequently returns from it.
- **Calling to a function in protected code.** This happens when the external library function (e.g., qsort, bsearch) calls a comparison function in the protected code.

We handle these cases by adding a dummy block before each return target and function entry in the protected code, since we cannot accurately identify calls to a library function and functions called by the library. This dummy block has only one instruction that jumps to the actual target, and is encrypted with a key generated from its address. C^3 transfers control to the dummy block when detecting a control transfer from unprotected to protected code, the range of which is recorded into a (secure) database.

In this way, C^3 ensures that these dummy blocks cannot be invoked by control flow transfers in the protected code and provides the same level of protection compared with existing CFI techniques.

5 IMPLEMENTATION

We implemented C^3 on an Ubuntu 64-bit system supporting inputs of ELF binary executables without source code.

5.1 Binary Rewriter

We developed our custom binary rewriter in 6,500 lines of Python code with the help of the disassembly engine Capstone [36]. The binary rewriter takes as input the ELF executable to be protected and the configuration of k. Embedding CFGs to an executable consists of three stages.

Before we embed control flow information, we first obtain the static CFG. We do this by modifying a recent work typearmor [45] (which builds on Dyninst [3]).

Secondly, we use Capstone [36] to disassemble the binary. C^3 uses the algorithm described in Appendix A to select basic blocks and then compute the secret for each of them. Note that the redistribution algorithm will likely distribute basic blocks apart from each other, and many NOP instructions need to be inserted into the `.text` section.

In the last stage, we update the corresponding section information including program entry point, program header, section header, items in relocation table, `.dynamic`, and `.dynsym` sections. In addition, some instructions need to be updated to maintain the original control flow:

- **Direct jumps:** We transform all indirect branch transfers to jump to the stub first; see Section 4.2. Therefore, there are only direct jumps in the `.text` section now. The target address of a direct jump is specified as a relative offset from the address of the jump instruction, which needs to be recomputed after basic block redistribution.
- **PC-relative addressing mode:** We also need to patch instructions with PC-relative addressing mode, which are often used to generate position-independent code. The new x86-64 architecture natively supports PC-relative addressing, e.g., `lea ox200000 (%rip), %rbp` adds `0x200000` to the program counter and saves it to `%rbp`. To ensure correctness, C^3 updates these instructions by recomputing the new offset using the new program counter and the address of the redistributed target.
- **Function pointers:** They are usually absolute addresses of indirect call targets that are loaded into registers. To fix these instructions, the absolute address of the callee should be patched at the instruction that loads its address into the corresponding register. This is done by identifying all possible function pointers with the help of the symbol table and patching them to the redistributed addresses. The same goes to global function pointers where C^3 updates the address in data section.
- **Data pointers:** They need to be patched, too, because the starting offset of the data section has changed. C^3 patches them by adding the new offset to the original value.
- **Jump tables/virtual tables:** C^3 updates the base address of the jump table by adding the new offset to it. Patching virtual tables follows the same mechanism.

5.2 Execution Environment

We make use of Pin [25] to implement the execution environment with 1,100 lines of code in C++. It first reads from the secure database the master key and the protection range and then installs a callback that intercepts the loading of all images to obtain ranges of the unprotected memory.

We then use the instrumentation callback at instruction granularity to detect a branch and compute the key for the next basic block. The decryption of basic blocks is performed by installing a callback that replaces Pin's default mechanism of fetching code from the target process. If the instruction fetched is within the range of protected code, we reconstruct the key from secret sharing parabola for decryption.

For code transitioning described in Section 4.5, we make use of `PIN_SetContextReg` to set the value of `%rip` register to the address of the dummy block which has just one instruction that jumps to the actual target, and then use the `PIN_ExecuteAt` API to direct execution to it. For the transition from protected code to unprotected code, C^3 stops

Table 1: Comparison with existing CFI techniques.

Exploits	BinCFI [50]	CCFIR [48]	IFCC [44]	kBouncer [32]	ROPecker [11]	C^3
Göktas et al [18]	✓	✓				
Davi et al. [14]	✓			✓	✓	
Conti et al. [12]	✓	✓	✓			
Hu et al. [21]	✓	✓	✓	✓	✓	

the decryption and let the code execute as normal. Similar to other CFI approaches, the attacker can use gadgets in unprotected code to construct code-reuse attack, which C^3 cannot defend against.

To avoid performing frequent key reconstruction for direct branch transfer instructions, we cache the key for subsequent use. Therefore, each direct branch transfer instruction corresponds to only one key reconstruction.

6 EVALUATION

We first analyze the security of C^3 and then measure its performance overhead with real-world applications.

6.1 Security

C^3 mitigates code-injection attacks in the same way Instruction-Set Randomization defeats them — when control flow is redirected to injected code, C^3 will decrypt it into random bytes. The attacker could not prepare the correct encrypted code since she does not know the master key.

C^3 also mitigates most Code-Reuse Attacks (CRA) due to three reasons. First, C^3 generates a wrong key when an invalid control transfer happens, which results in a random byte stream to be executed. Second, redistributing and encrypting basic blocks makes it harder for attackers to analyze and locate gadgets, which defeats most static CRA. Finally, the encrypted basic blocks result in little information revealed even when an attacker manages to dump the execution memory, which defeats most dynamic CRA.

6.1.1 Comparison with existing CFI techniques. A number of recent proof-of-concept exploits have shown how existing coarse-grained CFI techniques can be bypassed [12, 14, 18]. Although C^3 also enforces a coarse-grained policy, its unique handling of basic blocks (encryption) provides a new defense to make these exploits unsuccessful. Table 1 compares various CFI techniques with C^3 on the CFI policy enforced and defense capability against the exploits.

As shown in Table 1, existing instrumentation-based CFI methods [44, 48, 50] do not insert checks for unintended control-flow transfers, making them vulnerable to the exploit proposed by Conti et al. [12]. Such an exploit would not work on C^3 as all instructions (intended or unintended) are encrypted. The exploit proposed by Hu et al. [21] succeeds on all existing CFI methods as they rely on the assumption that W$\oplus$X is effective. Moreover, the content in the CFI table inserted by BinCFI provides sufficient information about useful gadgets if there is memory disclosure. However, since C^3 does not have this problem because it does not insert any metadata. The first three CFI approaches in Table 1 also suffer from TOCTOU attack — time of checking values of `esp/rsp` and time of executing `ret`, when the return address is stored in memory which could be modified by another thread. Under the protection of C^3, even if the address is modified by another thread, control flow will transfer to cipher-text which will result in program crashing.

Exploits that use call-preceded gadgets [14, 18] cannot succeed on C^3 since basic blocks are redistributed to random addresses. We performed experiments to verify the effectiveness of C^3 on defending against CRA that uses call-preceded gadgets using the test application `ndh_rop` from ROPgadget[1], a publicly available test set for ROP attacks. Our experiments verified that the payload that successfully exploits `ndh_rop` failed to run on C^3. Upon further investigation, we realized that it generated an illegal instruction fault when the return instruction directs control flow to the first call-preceded gadget. This is because this address is an invalid instruction which does not carry a valid proof to reconstruct the correct decryption key.

Compared with fine-grained approaches, e.g., Lockdown [33], which uses binary instrumentation to enforce CFI for different modules, C^3 can achieve better security as the attacker cannot make use of memory disclosure to traverse the memory of the victim program due to encryption of instructions. Basic block redistribution performed in C^3 can also be seen as effectively making the coarse-grained CFI policy finer-grained since the attacker cannot find the addresses of gadgets.

One may argue that the attacker could dump the protected code and do offline analysis to decrypt it. However, even if the attacker dumps the protected code and obtains the master key and the address of a basic block, she still has to try all possible encryption keys to see whether the basic block can be decrypted into valid instructions. We performed such experiments and realized that there are usually multiple such keys which have to be further tested on the resulting caller blocks for validity checks, and such checks have to carry on for callers of the callers, which makes it difficult for offline analysis to decrypt the protected code.

6.1.2 CFI effectiveness with AIR. Zhang and Sekar [50] propose using *Average Indirect target Reduction (AIR)* for measuring the strength of CFI, which has become a common method of evaluation [23, 33, 47]. It computes the average number of machine code instructions that are eliminated as possible targets of indirect control transfers.

The formula used by Zhang and Sekar is shown in Equation 3, where n is the number of indirect branch instructions in the program, and S is the total number of instructions to which an indirect branch can transfer control flow, whose value is the same as the size of code in a binary. $|T_j|$ is the

[1]https://github.com/JonathanSalwan/ROPgadget

Table 2: Average indirect target reduction.

Programs	# of valid targets		AIR
	ends with indirect branch	ends with direct branch	
`vsftpd` ($k = 9$)	25	0	99.84%
`Pure-FTPd` ($k = 10$)	172	0	98.95%
`ProFTPD` ($k = 11$)	506	0	99.23%
`httpd` ($k = 11$)	171	0	99.74%
`Nginx` ($k = 11$)	125	0	99.81%
`lighttpd` ($k = 10$)	35	0	99.79%
`Memcached` ($k = 10$)	62	0	99.62%
average			99.57%

possible number of targets to which indirect branch j can transfer control flow after CFI enforcement.

$$\frac{1}{n}\sum_{j=1}^{n}(1 - \frac{|T_j|}{S}) \qquad (AIR) \qquad (3)$$

In the case of C^3, $|T_j|$ is the possible number of targets that can be interpreted as valid basic blocks for indirect branch j. Since we substantially increase the size of the `.text` section, instead of enumerating all possible addresses (which requires testing millions of addresses), we randomly choose $16,384$ addresses for effective testing when $k \leq 10$ and $65,536$ addresses for other k values. We consider all basic blocks ending with indirect transfer instructions as valid, and those ending with direct transfer instructions valid if their targets are in the `.text` section.

The results are shown in Table 2 with server applications. Interestingly, there are few addresses that can be interpreted as valid basic blocks, and all of them end with indirect transfer instructions. This is because C^3 extends the displacement in direct branches to four bytes, which makes the probability that a random sequence be interpreted as a valid direct branch small. On average, C^3 achieves an AIR value of 99.57%, comparable to existing CFI approaches [33, 50].

6.1.3 JIT-ROP. JIT-ROP [41] is an attack against fine-grained randomization. It assembles ROP gadgets "on-demand" without knowing the memory layout by exploiting the disclosure of a single code pointer. Specifically, the adversary traverses the memory space that the leaked pointer points to, searches for gadgets and cross-page transfer instructions to find new code pages and other useful gadgets. However, under C^3, a read performed from a code page yields cipher-text, which the adversary cannot disassemble without knowing the decryption key. As such, an adversary cannot use JIT-ROP to disclose new code pages to find gadgets.

To verify our intuition, we tried to use the ROP gadget finding tool peda[2] to identify gadgets in the protected binary `nginx-1.4.0` after the loading phase, simulating the full disclosure of the code segment. Many gadgets ending with `ret` are found, which are chained together to form an attack payload. However, the gadgets found were based on encrypted basic blocks, which become invalid instructions and lead the execution into an illegal instruction fault.

6.1.4 Blind ROP. Blind ROP [4] uses the response from the victim process (crash vs. no crash) as a side channel to incrementally guess the position of a gadget. It assumes that the adversary can disassemble the code pages to find the required gadgets. Since the code pages are encrypted with C^3, Blind ROP will not succeed. We applied the exploit script provided by Bittau et al.[3] to `nginx-1.4.0` protected by C^3, and found that it made all worker threads "stuck" as they were all running into an infinite loop of locating gadgets. Blind ROP uses a conservative implementation to incrementally populate the stack to find a stack-based stop gadget to avoid hanging. However, with C^3, every attempt in transferring control to this stack-based stop gadget results in a failure due to incorrect decryption of the callee block.

6.1.5 Control-Flow Bending. Control-Flow Bending (CFB) [7] bypasses conventional CFI that statically generates CFGs. CFB abuses certain functions whose executions may change their own return addresses to point to any call-preceded site which allows the attacker to "bend" the control flow. C^3 mitigates CFB attacks by preventing the attacker from locating call-preceded basic blocks — thanks to redistributing and encrypting of basic blocks.

Although C^3 successfully defends against these existing advanced control-flow-hijacking attacks, we acknowledge that it is not necessarily effective against an attack specifically crafted for C^3. We further discuss this possibility in Section 7.

6.2 Performance overhead

We evaluated C^3 with three FTP servers (`vsftpd`, `ProFTPD`, and `Pure-FTPd`), three web servers (`Nginx`, `lighttpd`, and `Apache`), a distributed memory caching system (`Memcached`), and some common applications (image processing tools `sam2p`, `GraphicsMagic`, and `ImageMagics` and `bzip2`). All programs are executed with their default settings on a desktop computer with an Intel i7-4510u CPU with 8GB of memory running x64 version of Ubuntu.

To benchmark web servers, we configured Apache Benchmark[4] to issue 2,000 requests with 100 concurrent connections. For FTP servers, we configured pyftpbench benchmark[5] to open 20 connections and request 100 files per connection with over 100MB of files requested. To benchmark `Memcached`, we used memslap[6]. We ran each experiment 10 times, ensuring that the CPUs were fully loaded throughout the tests, and report the median.

Since C^3 is implemented on top of the dynamic instrumentation platform Pin, we measure the performance of C^3 in terms of the additional execution overhead compared to these programs executing on an un-modified Pin v.3.5. To

[2]https://github.com/longld/peda

[3]http://www.scs.stanford.edu/brop/

[4]httpd.apache.org/docs/2.4/programs/ab.html

[5]http://code.google.com/p/pyftpdlib

[6]http://docs.libmemcached.org/bin/memslap.html

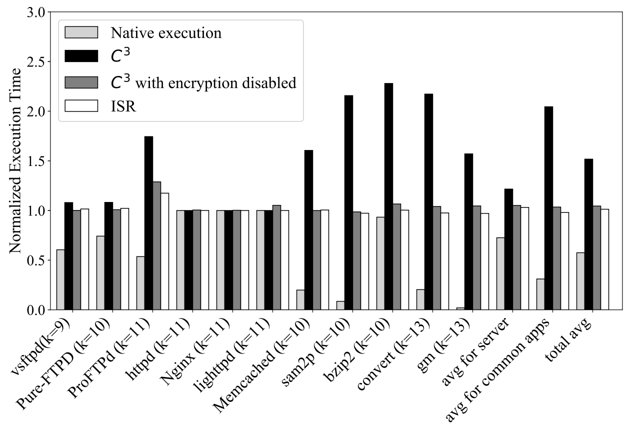


Figure 7: Overall overhead of C^3. The result is normalized to the baseline execution time on un-modified Pin.

enable a better understanding of the results, we also report the execution overhead of another system that is built on top of Pin, namely Instruction-Set Randomization implemented by Portokalidis et al. [34].

6.2.1 Execution Time. We report, in Figure 7, the execution time of each program under four settings: native execution, ISR [34], C^3 with encryption disabled, and C^3 with encryption turned on. Results are normalized to a baseline for its execution on un-modified Pin. k was chosen to be the minimum that successfully distributes the basic block for secret sharing, whose values are shown in brackets.

Being consistent with results reported in the original paper [34], ISR does not incur observable slow down compared with execution on un-modified Pin since there is no additional instrumentation. C^3 presents very similar results when encryption is disabled for the same reason. With encryption turned on, C^3 experiences less than 10% overhead for server applications while non-server applications generally suffer from significantly higher overhead. Note that when compared with their respective native executions, several server applications on C^3 have very small runtime performance, although the average runtime overhead is about 70%.

To gain a better understanding of contributions to such overhead and why non-server applications perform worse, we conduct the next finer-grained analysis of C^3 to see which components of C^3 are the main contributors to the overhead. We first identify the following three main tasks of C^3 that potentially contribute to the performance overhead:

- **Key Reconstruction (KR).** This is performed for every branch in the program, be it a direct branch (whose key reconstruction is denoted as dKR) or an indirect branch (whose key reconstruction is denoted as iKR).
- **Decryption.** Since C^3 uses XOR operation as in ISR [34], decryption incurs minimal overhead as confirmed in Figure 7 in which ISR only results in a small overhead.
- **Execution Redirection (ER).** This happens when execution transitions from unprotected code to protected code. Since it requires saving and restoring the entire register state [30], it could result in significant overhead.

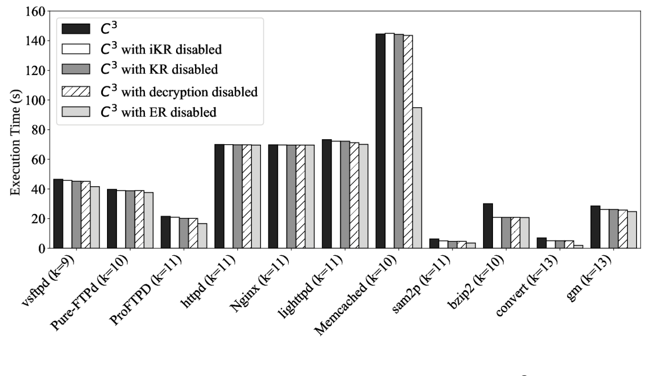


Figure 8: Detailed overhead of C^3.

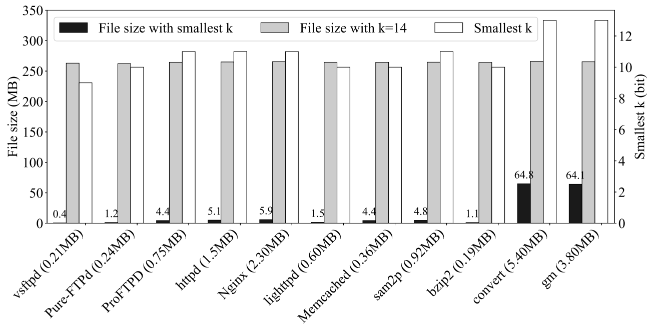


Figure 9: File size with different secret sizes.

Figure 8 shows the overhead of C^3 with certain components disabled to more accurately attribute the overhead to the corresponding components. This time, the overhead is presented in seconds without normalization (to visualize the small differences). We have two important observations.

First, iKR (whose contribution can be seen by comparing the bars for C^3 and those for C^3 with iKR disabled) contributes more overhead than dKR (whose contribution can be seen by comparing the bars for C^3 with iKR disabled and those for C^3 with KR disabled). This is mainly due to optimizations C^3 implements for direct branches, where key reconstruction is done only once and results are cached for subsequent decryption. Such optimization does not apply to indirect branches since the control transfer target changes in each indirect branch. Therefore, applications with more indirect branches suffer higher overhead on C^3.

Table 3 records the number of indirect branches, direct branches, and transitions from unprotected to protected code. Note that `bzip2` has a larger number of indirect branches executed, which explains its higher overhead on C^3.

Our second observation from Figure 8 is on execution redirection ER. We found that ER contributes significantly to the performance overhead for `vsftpd`, `proftpd`, `memcached` and the non-server applications except `bzip2`, which can be explained by the numbers in the last column of Table 3.

Table 3: Number of various branches executed. We report the number of indirect branches executed in iKR #; dKR # is the number of direct branches executed; the number of transitions from unprotected to protected code is reported in ER #.

Programs	iKR #	dKR #	ER #
`vsftpd`	3.99×10^6	2.43×10^7	1.38×10^6
`Pure-FTPd`	1.09×10^6	6.29×10^6	2.49×10^5
`ProFTPD`	5.07×10^6	5.67×10^7	1.68×10^6
`httpd`	1.16×10^5	4.82×10^5	9.97×10^4
`Nginx`	3.43×10^3	2.51×10^5	4.70×10^3
`lighttpd`	1.75×10^5	2.28×10^6	6.31×10^4
`Memcached`	2.37×10^7	1.54×10^8	7.64×10^6
`sam2p`	2.22×10^7	1.33×10^8	1.22×10^5
`bzip2`	1.36×10^8	7.64×10^9	5.83×10^4
`convert`	2.55×10^7	6.81×10^7	4.36×10^5
`gm`	1.14×10^6	4.25×10^7	1.21×10^5

6.2.2 Space. The redistribution of basic blocks in C^3 makes use of a potentially large address space with gaps among various basic blocks; see Section 4.3. The resulting size of the binary executable mainly depends on the length of the secret, i.e., k. For example when $k = 12$, the address of an instruction can be as big as 2^{24}.

Figure 9 shows the resulting file sizes after C^3 processing with two settings — one using a smallest possible setting of k (which varies among different programs) and the other with $k = 14$. We argue that although the size of the binary increases significantly with bigger values of k, storage is cheap and it is usually not an issue with hard-disk space. That said, a larger k also results in slightly bigger runtime overhead as more instructions are executed to extract the values of x and y from an address, and key reconstruction could also require slightly more instructions executed.

7 DISCUSSION

7.1 Return-into-Pin

C^3 provides CFI protection on the application but not the dynamic instrumentation platform, i.e., Pin. An attacker, in theory, could perform an attack by returning into instructions in Pin so that control is diverted directly into the gadgets found in Pin. We call such an attack "return-into-Pin". Such a control transfer would circumvent C^3, enabling the attacker to successfully execute control-flow hijacking attacks. Our design of C^3 is compatible with other isolation hardening solutions, such as Software-based Fault Isolation (SFI) [46], though, which can instrument memory writes to check whether the application attempts to write to a page "owned" by Pin. Another (probably better) defense is to implement the execution environment in a more isolated layer such as the OS layer, the hypervisor layer, the hardware layer, or even inside SGX [13].

That said, once instructions are in Pin's code cache, Pin will not instrument them but jump there directly, which improves the performance of C^3. Meanwhile, such optimization does not hurt security since Pin uses a local hash table for each individual indirect branch transfer, which will contain only the correctly decrypted targets. Any new targets will result in a hash table miss and basic block decryption.

7.2 Return-into-libc

In general, CFI does not defend against all return-into-libc attacks. Specifically, C^3 does not encrypt instruction sequences in the `.plt` section, and so any return instructions can transfer control to entries in the `.plt` section. In order to protect these library function calls, one could statically compile the libraries into the application.

7.3 Length of the Keys

Brute-force attacks have been introduced to reconstructing the encryption keys in ISR [42], which is also applicable to C^3. However, since we use a different key for encrypting each basic block, such brute-forcing will be ineffective because a successful attack typically requires the reconstruction of keys for multiple basic blocks. To this end, we believe that using XOR as the encryption algorithm for improved performance is justifiable, although C^3 can definitely use a more secure encryption scheme. We currently use a 32-bit master key since it is unique for the entire program. C^3 could improve its security with a longer master key of, say, 80 bits.

7.4 Fine-grained CFI Enforcement

C^3 can be extended to enforce fine-grained CFI. For example, C^3 can enforce the fine-grained CFI policy for forward-edge indirect branch transfer instructions enforced in TypeArmor [45] by classifying functions and indirect call instructions into different clusters according to the number of arguments they can accept, and then encrypting basic blocks with the more accurate set of control transfers derived. We note that enforcing a finer-grained CFI policy could likely reduce the execution time and space overhead of C^3 due to fewer valid control transfers on average and consequently less secret sharing and block redistribution needed.

7.5 Other limitations

First, C^3 relies on static analysis and rewriting of binaries. The current implementation does not support dynamically generated code or self-modifying code.

Second, C^3 prevents attackers from directly reading the code and finding useful gadgets. However, code pointers in data areas such as stack and heap are still vulnerable to indirect memory disclosure. For example, if the protected binary has a format string vulnerability, the attacker can print out the valid memory locations for return instructions, which may allow an attacker to use, e.g., call-preceded gadgets. This is a rather general limitation shared by other techniques performing binary rewriting [47, 48, 50].

Third, C^3 renders caching and pipelining less effective. It is a limitation for most ISR approaches, excluding those performing decryption when there are I-cache misses and store plain text in the I-cache.

Lastly, C^3 requires symbol names in the executable to enable patching function and data pointers after basic block redistribution. It also requires that data and code be completely separated to enforce instruction encryption. For binaries that do not contain symbol information, we can use external tools, e.g., Unstrip[7] and others [35, 39], to restore the symbol information. Similarly, there are approaches to identify data embedded within code [49, 50].

8 CONCLUSION

We present C^3, a new CFI technique that embeds the CFG into instructions to perform CFI checks without relying on additional data structure like the read-only table used in existing CFI approaches. It encrypts each basic block with a key that can be reconstructed by any of its valid callers with the help of a secret sharing scheme. During execution, C^3 reconstructs the key when a branch transfer instruction is encountered. Our evaluation shows that C^3 can effectively defend against most control-flow hijacking attacks with moderate overhead.

ACKNOWLEDGEMENT

We thank our shepherd, Hong Hu and the anonymous reviewers for their valuable comments and suggestions that have helped improve our paper.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[2] Elena Gabriela Barrantes, David H Ackley, Trek S Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 281–289.

[3] Andrew R Bernat and Barton P Miller. 2011. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. ACM, 9–16.

[4] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[5] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 30–40.

[6] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 987–1004.

[7] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity.. In *Proceedings of the 24th USENIX Security Symposium*. 161–176.

[8] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses.. In *Proceedings of the 23rd USENIX Security Symposium*. 385–399.

[9] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 559–572.

[10] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. 2019. DynOpvm: VM-based software Obfuscation with Dynamic Opcode Mapping. In *Proceedings of the 17th International Conference on Applied Cryptography and Network Security*.

[11] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, Huijie Deng, et al. 2014. ROPecker: A generic and practical approach for defending against ROP attack. In *Symposium on Network and Distributed System Security*.

[12] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*. ACM, 952–963.

[13] Intel Corporation. 2019. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx/.

[14] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection.. In *Proceedings of the 23rd USENIX Security Symposium*.

[15] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. 2017. SOFIA: Software and control flow integrity architecture. *Computers & Security* 68 (2017), 16–35.

[16] Jianming Fu, Xu Zhang, and Yan Lin. 2015. An Instruction-Set Randomization Using Length-Preserving Permutation. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 376–383.

[17] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding control flows using intel processor trace. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 585–598.

[18] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*. IEEE, 575–589.

[19] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 173–184.

[20] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1470–1486.

[21] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*. IEEE, 969–986.

[22] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 272–280.

[23] Yan Lin, Xiaoxiao Tang, Debin Gao, and Jianming Fu. 2016. Control flow integrity enforcement with dynamic code optimization. In *International Conference on Information Security*. Springer, 366–385.

[24] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and efficient cfi enforcement with intel processor trace. In *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture*. IEEE, 529–540.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM conference on Programming language design and implementation*. ACM, 190–200.

[26] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity.. In *Symposium on Network and Distributed System Security*, Vol. 26. 27–30.

[7] http://paradyn.org/html/tools/unstrip.html

[27] George C Necula. 2002. Proof-carrying code. design and implementation. In *Proof and system-reliability*. Springer, 261–288.

[28] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*. ACM, 577–587.

[29] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*. ACM, 914–926.

[30] Heidi Pan, Krste Asanović, Robert Cohn, and Chi-Keung Luk. 2005. Controlling program execution through binary instrumentation. *ACM SIGARCH Computer Architecture News* 33, 5 (2005), 45–50.

[31] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. 2013. ASIST: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 981–992.

[32] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing.. In *Proceedings of the 22nd USENIX Security Symposium*. 447–462.

[33] Mathias Payer, Antonio Barresi, and Thomas R Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 144–164.

[34] Georgios Portokalidis and Angelos D Keromytis. 2010. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 41–48.

[35] Rui Qiao and R Sekar. 2017. Function interface analysis: A principled approach for function recognition in COTS binaries. In *Proceeding of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 201–212.

[36] Nguyen Anh Quynh. 2014. Capstone: Next-gen disassembly framework. *Black Hat USA* (2014).

[37] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 552–561.

[38] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[39] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks.. In *Proceeding of the 24th USENIX Security Symposium*. 611–626.

[40] Kanad Sinha, Vasileios P Kemerlis, and Simha Sethumadhavan. 2017. Reviving instruction set randomization. In *International Symposium on Hardware Oriented Security and Trust*. IEEE, 21–28.

[41] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. IEEE, 574–588.

[42] Ana Nora Sovarel, David Evans, and Nathanael Paul. 2005. Where's the FEEB? The Effectiveness of Instruction Set Randomization.. In *Proceedings of the 15th USENIX Security Symposium*.

[43] Dean Sullivan, Orlando Arias, David Gens, Lucas Davi, Ahmad-Reza Sadeghi, and Yier Jin. 2017. Execution Integrity with In-Place Encryption. *arXiv preprint arXiv:1703.02698* (2017).

[44] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM.. In *Proceedings of the 23rd USENIX Security Symposium*. 941–955.

[45] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*. IEEE, 934–953.

[46] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1994. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, Vol. 27. ACM, 203–216.

[47] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. 2015. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 331–340.

[48] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[49] Mingwei Zhang, Michalis Polychronakis, and R Sekar. 2017. Protecting COTS binaries from disclosure-guided code reuse attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 128–140.

[50] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries.. In *Proceedings of the 22nd USENIX Security Symposium*. 337–352.

[51] Mingwei Zhang and R Sekar. 2015. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. (2015).

A BASIC BLOCK REDISTRIBUTION ALGORITHM

Algorithm 1 Basic Block Redistribution

```
procedure REDISTRIBUTION(callee, master_key, k, p)
    if callee not in key_block then
        $priority\_callee = Look\_Ahead(callee)$
        if priority_callee then
            Redistribution($priority\_callee, master\_key, k, p$)
        $callers$ = callee_caller[$callee$]
        $moved\_callers$ = find_moved_callers($callers$)
        if len(moved_callers) == 0  then
            $caller$ = random_choose_caller($callers$)
        if len(moved_callers) == 1 then
            $caller$ = moved_callers[0]
        compute_key($callee, caller, master\_key, k, p$)
      # move all callers of this basic block to be on the curve
        for i in callers do
            if i not in redistributed_block then
                move_caller($callee, i, master\_key, k, p$)
      # DFS: process callees of this basic block
        for i in caller_callee[callee] do
            Redistribution($i, master\_key, k, p$)
      # backtracking
        for i in callers do
            for j in caller_callee[i] do
                Redistribution($j, master\_key, k, p$)
```