Modular Control-Flow Integrity

Ben Niu Gang Tan
Lehigh University
ben210@lehigh.edu gtan@cse.lehigh.edu

Abstract

Control-Flow Integrity (CFI) is a software-hardening technique. It inlines checks into a program so that its execution always follows a predetermined Control-Flow Graph (CFG). As a result, CFI is effective at preventing control-flow hijacking attacks. However, past fine-grained CFI implementations do not support separate compilation, which hinders its adoption.

We present Modular Control-Flow Integrity (MCFI), a new CFI technique that supports separate compilation. MCFI allows modules to be independently instrumented and linked statically or dynamically. The combined module enforces a CFG that is a combination of the individual modules' CFGs. One challenge in supporting dynamic linking in multithreaded code is how to ensure a safe transition from the old CFG to the new CFG when libraries are dynamically linked. The key technique we use is to have the CFG represented in a runtime data structure and have reads and updates of the data structure wrapped in transactions to ensure thread safety. Our evaluation on SPECCPU2006 benchmarks shows that MCFI supports separate compilation, incurs low overhead of around 5%, and enhances security.

Categories and Subject Descriptors D.4.6 [*Software*]: Operating Systems—Security and Protection

General Terms Security

Keywords Control-Flow Integrity, Modularity, Separate Compilation

1. Introduction

Many software attacks hijack the control flow of a program to transfer to attacker-injected code, or to a dangerous library function as in return-to-libc attacks[15], or to some existing code snippet of the program as in Return-Oriented Programming (ROP [19]) attacks. These control-flow attacks can be mitigated by Control-Flow Integrity (CFI [3]). It rewrites the program to check indirect branches, which are return instructions, indirect jumps (jumps via a register or a memory operand), and indirect calls (calls via a register or a memory operand). A program's control flow is guaranteed to follow a given Control-Flow Graph (CFG), even under attack.

However, CFI has not seen wide adoption since its debut in 2005. One important factor is the lack of *separate compilation*. The term separate compilation refers to the ability of a compiler to separately compile modules of an application and link the compiled modules. In the context of CFI, it refers to the ability to perform instrumentation of modules separately, without considering other modules, and to link instrumented modules into a working executable. Unfortunately, past CFI techniques require all modules of an application, including libraries, to be available at instrumentation time. For instance, the classic CFI instrumentation [3] inserts identifiers (representing a class of indirect branches and targets) before branch targets and checks before indirect branches to ensure that they jump to targets specified in the CFG. The identifiers are embedded in instructions and cannot appear in the rest of the code. However, this property cannot be guaranteed without inspecting the whole program. Other CFI-instrumentation techniques also do not support separate compilation and the reasons will be discussed in the related-work section (Sec. 3).

The loss of separate compilation is a severe restriction in practice because libraries cannot be instrumented once and reused across programs. It implies that each program has to come with its own instrumented version of libraries. This is especially cumbersome for Dynamic-Link Libraries (DLLs), which are designed to be reused.

In this paper, we present *Modular Control-Flow Integrity* (MCFI), which extends CFI with the support of separate compilation. In MCFI, an application is divided into multiple modules (e.g., one module can be a library). Each module contains code, data, and auxiliary information that helps its linking with other modules and the generation of the module's CFG. Code of a module is instrumented separately for control-flow integrity. When modules are linked either statically or dynamically, their auxiliary information is combined and used to generate a new CFG, which is the new control-flow policy for the combined module after linking. The new policy may allow an indirect branch to target more destinations. For example, suppose a function named `f` in module $\texttt{M}_1$ contains a return instruction. $\texttt{M}_1$'s internal CFG allows the return instruction to return to any caller of `f` in $\texttt{M}_1$. After $\texttt{M}_1$ is linked with a second module $\texttt{M}_2$, the return instruction can also return to any caller of `f` in $\texttt{M}_2$. The important point is that the control-flow policy changes during linking, implying that the policy has to be updated during runtime when loading a dynamically linked library.

The dynamic nature of the control-flow policy when linking libraries poses two main challenges in designing MCFI:

- First, how to update the policy safely and efficiently at runtime in the presence of multithreading? We need to handle the case when one thread uses the current policy to decide whether an indirect branch is allowed while another thread concurrently updates the policy to a new one when loading code.
- Second, how to efficiently generate a new policy with high precision when modules are combined? There is a wide range of design choices. The challenge is to have an efficient methodology that generates a reasonably precise CFG.

PLDI'14, June 9 - 11 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2784-8/14/06. . . $15.00.
http://dx.doi.org/10.1145/2594291.2594295

To address the first challenge, MCFI represents the CFG in separate tables outside of the code region. To achieve a smooth transition from the old policy to the new one, MCFI designs table-check and table-update transactions, which are inspired by Software Transactional Memory (STM [20]). However, generic STM algorithms would incur high performance overhead. Instead, we propose a lightweight STM implementation that performs table transactions efficiently in MCFI's context.

To address the second challenge, MCFI augments modules with auxiliary type information and uses the type information for CFG generation. MCFI takes a module's source code and compiles it using a modified LLVM [1] compiler to acquire the type information.

We highlight our contributions below:

- To the best of our knowledge, MCFI is the first efficient CFI instrumentation that supports separate compilation. It addresses the challenge of how to support dynamically linked libraries for CFI in the presence of multithreaded code, using a novel approach based on transactions. We believe MCFI can greatly improve CFI's practicality.
- We describe a simple yet effective way of generating CFGs for C programs based on type matching. It is efficient and can be used during dynamic linking. It generates relatively precise CFGs, while breaking only small portions of C programs. Our empirical evidence shows that C programs can be made to be compatible with our CFG-generation process with no or small changes to the source code.
- In evaluating MCFI, we have implemented a compilation toolchain, which instruments C programs for x86. Our experiments on SPECCPU2006 benchmarks show that MCFI imposes about 5% execution-time overhead on average.

The remainder of this paper is organized as follows. We first discuss the CFI background in Sec. 2 and related work in Sec. 3. Sec. 4 summarizes MCFI's approach, with details in Sec. 5 and Sec. 6. We discuss MCFI's toolchain in Sec. 7 and present evaluation in Sec. 8. Finally, we conclude in Sec. 9.

2. Background: CFI

A CFI policy for a program is a CFG whose nodes are instructions, and whose edges connect instructions to allowed next instructions in control flow. Code *C* respects its CFI policy *P* if and only if all control transfers in *C*, when executed, respect the graph *P*.

We next discuss the classic CFI enforcement [3]. Hereafter, we use the term CFI to refer to the classic CFI. For enforcement, it uses a combination of static and dynamic checks. For a direct branch, a static verifier is used to check if its targets are allowed by the CFG. For an indirect branch, CFI inserts runtime checks into the program to ensure that the control transfer is consistent with the CFG. The focus of CFI is how to instrument indirect branches, discussed next.

First, the set of indirect branch targets is partitioned into *equivalence classes*. Two target addresses are equivalent if there is an indirect branch that can jump to both targets according to the CFG. An indirect branch is allowed to jump to any destination in the same equivalence class. If two indirect branches target two sets of destinations and those two sets are not disjoint, the two sets are merged into one equivalence class. This results in some CFG precision loss.

After partitioning, each equivalence class is assigned a unique number. An address's *Equivalence-Class Number* (ECN) refers to the number given to the equivalence class to which the address belongs. The *branch ECN* of an indirect branch refers to the ECN of the equivalence class whose addresses the branch can jump to. The *target ECN* of an indirect branch is a dynamic notion and refers to the ECN of the equivalence class in which the actual destination is when the indirect branch runs in a specific state.

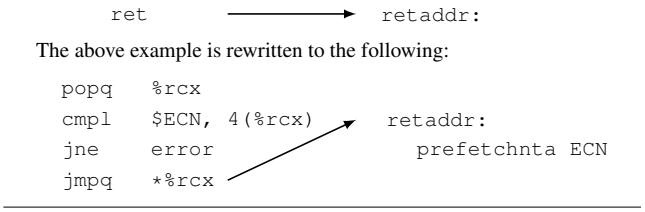

Figure 1. An example of the classic CFI instrumentation.

For an indirect branch to respect the CFI policy, its branch ECN must be the same as its target ECN. This is enforced by the CFI instrumentation. Fig. 1 shows the instrumentation of a return instruction that jumps to the address `retaddr`:

(a) At the target, `retaddr`, CFI inserts a side-effect-free instruction with the target address's ECN embedded in. The example uses the `prefetchnta` instruction, which performs memory prefetching.

(b) The return instruction is rewritten to (1) pop the return address to a temporary register (`%rcx` in this case); (2) retrieve the target ECN and compare it with the branch ECN using `cmpl` and `jne` (the address `%rcx+4` points to the middle of `prefetchnta`, if the correct return address is on the stack); (3) if they are the same, the control is transferred.

In this technique, ECNs are embedded in the non-writable code section. Consequently, an ECN must be unique and cannot overlap with the encoding of instructions in the rest of the code. Otherwise, it would be possible for an indirect branch to jump to more targets than allowed. The global uniqueness requirement is the reason why the classic CFI does not support separate compilation because the rewriting cannot be performed without all modules.

3. Related Work

In addition to the classic CFI, a number of other CFI instrumentation techniques have been introduced in the literature. According to the CFG precision, they can be broadly put into two categories: fine-grained and coarse-grained. The classic CFI we have discussed is a fine-grained CFI technique. In general, fine-grained CFI techniques rewrite a program so that its control flow follows a given CFG. Each indirect branch can have its own set of destinations according to the CFG. By contrast, in a typical coarse-grained CFI, indirect branches share a set or a few sets of targets.

Fine-grained CFI. Fine-grained CFI significantly enhances security and we believe it is the best defense against Return-Oriented Programming attacks. Various techniques for fine-grained CFI have been proposed for x86 [3, 4, 9, 12, 21] and ARM [7, 17]. We have explained the classic CFI and we next discuss other techniques.

For each indirect branch, HyperSafe [21] statically constructs a table, containing all target addresses that the indirect branch can jump to. The program is changed so that table indexes, not target addresses, are used in the program. Before an indirect branch, an index is converted into a target address using information in the table. HyperSafe's CFI precision is higher than the classic CFI's, which uses the notion of equivalence classes for efficiency at the expense of precision. However, HyperSafe does not support separate compilation because it requires a whole-program analysis to build the tables and those tables cannot be changed at runtime.

WIT's CFI enforcement [4] uses a color table. The color table is built by static analysis on the compiler IR code. Each indirect call is statically assigned a color, and functions that the call can target are assigned the same color. The color table is represented during runtime and dynamic checks consult the table before an indirect

call. WIT's color table is similar to MCFI's tables. But WIT relies on a whole-program analysis to construct its table and does not support dynamically linked libraries.

MoCFI [7] enforces CFI on the ARM architecture. An indirect branch is instrumented to consult an external CFG validation module. CFR [17] is another CFI instrumentation technique for ARM. For each indirect branch, it inserts a series of checks iterating through a white list of targets embedded in the code. Neither MoCFI nor CFR has addressed the issue of updating the CFG policy during dynamic linking.

Coarse-grained CFI. An imprecise CFG is enforced on indirect branches in coarse-grained CFI. In the simplest case, all indirect branches share a common set of possible targets. In a refinement, targets are classified into several categories and each class of indirect branches is allowed to jump to a particular category; for instance, all indirect calls can jump to function entries (but not other targets). The precision of coarse-grained CFI is much lower and consequently the search space for an attack is much larger than fine-grained CFI. Indeed, a recent work showed that ROP attacks can still be mounted on systems hardened with coarse-grained CFI [10]. The benefits of coarse-grained CFI are that the performance overhead is lower. Separate compilation is also supported. For instance, NaCl-JIT [5] demonstrates how code can be dynamically loaded and modified in the presence of multithreading.

We next enumerate several coarse-grained CFI techniques. PittSFIeld [13] and NaCl [5, 18, 22] implement aligned-chunk CFI: the code is divided into fixed-size chunks (16 or 32 bytes) and indirect branches are restricted by address masking to target chunk beginnings. Our previous work MIP [16] generalizes the idea of fixed-size chunks and arranges instructions in variable-size chunks. Each chunk contains one instruction or a sequence of instructions that does not have indirect-branch targets in the middle of the sequence. Indirect branches are dynamically restricted to never target the middle of a chunk through the help of a bitmap that remembers the chunk boundaries. The role of the bitmap in MIP is similar to the role of ID tables in MCFI. However, a MIP module's bitmap is never updated during runtime and is made read-only. Therefore, MIP does not need to address one key challenge in MCFI: thread safety of the ID tables when there are concurrent updates and reads. Another coarse-grained CFI example is CCFIR [25], which rewrites code to redirect all indirect branches into a new springboard code region. In the springboard region, each direct or indirect branch target has an aligned entry that contains instructions to transfer the control flow. Indirect branches are checked to ensure they target springboard entries at runtime. binCFI [26] replaces indirect branches with direct jumps to an external routine. The external routine checks an address table, which records the allowed addresses of the original indirect branch. Only if the target address is in the address table can the actual control-transfer occur.

4. MCFI Overview

MCFI enforces fine-grained CFI. We first present an overview, including its threat model and its main techniques.

Threat model. MCFI adopts CFI's concurrent attacker model [3]. The model allows a strong adversary, which is treated as a separate thread running in parallel with user threads. The attacker thread can read and write any memory (subject to memory page protection). Consequently, the attacker can corrupt writable memory between any two instructions in the user program. However, it is assumed that machine registers of a thread cannot be directly modified by the attacker thread. However, the attacker can still affect registers indirectly by corrupting memory. As an example, if the program reads from a region of writable memory to a register, then the register's value is under the attacker's control because the attacker can write any value to that region of memory.

In addition, to prevent arbitrary code execution, a trusted MCFI runtime enforces the invariant that no memory regions are both writable and executable at the same time. The invariant is enforced when an application is initially loaded by the runtime. The runtime sets up a separate code and data region. Code is loaded into the code region, which is executable and readable but not writable. Note that the code region can include some read-only data such as jump tables. The data region is readable and certain parts are writable, but not executable. The invariant also holds when dynamically linking libraries. New libraries are loaded into unoccupied parts in the code and data region.

ID tables. Similar to CFI, MCFI partitions indirect branch targets into equivalence classes and labels each with an ECN. To remove the global uniqueness requirement in the classic CFI, ECNs are pulled out of the code section and stored in a runtime data structure consisting of two separate tables. These tables are conceptually maps from addresses to IDs. An ID is a unique identifier associated with an address. It contains an ECN and other components (the format of an ID is detailed later). The branch ID table, called the *Bary table*, maps from an indirect-branch location to the location's branch ID, which is the identifier of the equivalence class of addresses the branch is allowed to jump to. The target ID table, called the *Tary table*, maps from an address to the identifier of the equivalence class to which the address belongs.

With the ID tables, instrumenting an indirect branch is straightforward. Take the example of a return instruction located at address l. The instrumentation can first use the Bary table to look up the branch ID for address l, use the Tary table to look up the target ID for the actual return address, and check whether the branch ID is the same as the target ID.

Separating IDs from code has several benefits. First, IDs in the tables can overlap with the numbers in the code section, eliminating the global ID uniqueness assumption in the classic CFI. Second, the instrumentation code before indirect branches is parameterized over the ID tables and remains the same once loaded. Therefore, code pages for applications and libraries can be shared among processes, saving memory and application launch time. Third, centralized ID tables enable favorable memory cache effect and fast table updates using parallel memory-copy mechanisms of the CPU.

Table access transactions. The ID tables may be accessed concurrently by multiple threads. One thread may dynamically load a module, which triggers the generation of a new CFG. Consequently, a new set of IDs based on the new CFG needs to be put into the ID tables. At the same time, another thread may execute an indirect branch, which requires reading IDs from the tables. Since concurrent reads and writes are possible, a synchronization mechanism must be designed for maintaining the consistency of tables. Otherwise, the tables may reach some intermediate state that allows for illegal control-flow transfers. A simple lock-based scheme for accessing tables could be adopted, but it would incur a large performance penalty due to MCFI's table-read-dominant workloads: dynamic linking is a rare event compared to the use of an indirect branch (especially return instructions); even in Just-In-Time (JIT) compilation environments such as the Google V8 JavaScript engine, which optimizes code on-the-fly, the number of indirect branch execution is roughly 10^8 times of CFG updates triggered by dynamic code installation.

Our solution is to wrap table operations into transactions and use a custom form of Software Transactional Memory (STM) to achieve safety and efficiency. We use two kinds of transactions:

(1) *Check transaction (TxCheck).* This transaction is executed before an indirect branch. Given the address where the indirect branch is located and the address which the indirect branch targets, the transaction reads the branch ID and the target ID from

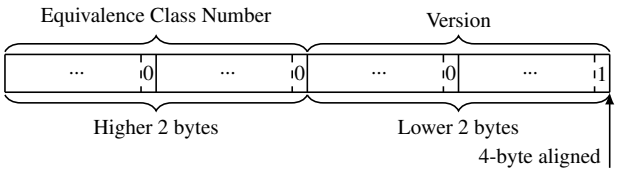

Figure 2. MCFI's ID Encoding.

the tables, compares the two IDs, and takes actions if the IDs do not match. This transaction performs only table reads.

(2) *Update transaction (TxUpdate).* This transaction is executed during dynamic linking. Given the new IDs generated from the new CFG after linking a library, this transaction updates the Bary and Tary tables.

The reason why a transaction-based approach is more efficient is that the check transaction performs speculative table reads, assuming there are no other threads performing concurrent writes; if the assumption is wrong, it aborts and retries. This technique matches our context well and provides needed efficiency. More details about ID tables and table transactions are in Sec. 5.

Module linking. An MCFI module not only contains code and data, but also auxiliary information. There is a range of options about what auxiliary information to attach to a module for CFG generation. In general, the more information that a module carries, the better precision the generated CFG has. MCFI adopts type information, which tells the types of functions and function pointers. The type information is used to generate a CFG for modules. An indirect call via a function pointer can invoke a function as long as the types of the function pointer and the function match. We find this approach can generate relatively precise CFGs, while requiring modest effort to adapt the source code.

5. ID Tables and Transactions

We next describe the detailed design of ID tables and transactions that access the tables. We present the design for the x86 architecture, including x86-32 and x86-64. Other architectures such as ARM can also be supported with small adjustments.

5.1 ID Tables

As discussed, MCFI maintains two tables. Both map from addresses to IDs. The Bary table holds branch IDs and the Tary table holds target IDs. An ID is four-byte long, visualized in Fig. 2. An ID is stored in a four-byte aligned memory address so that a single memory access instruction can atomically access it.

An MCFI ID contains several components. The first component is composed of the least significant bits in the four bytes. They are reserved and have the special bit values 0, 0, 0, and 1, from high to low bytes. These reserved bits are to prevent the use of an address that points to the middle of an ID to look up the tables; more on this will be discussed shortly. We define a *valid ID* to be an ID that has the special bit values at the reserved-bit positions.

An MCFI ID also contains a fourteen-bit target-address ECN in the higher two bytes and a fourteen-bit version number in the lower two bytes. The ECN is the same as the one in the classic CFI and tells what equivalence class the address associated with the ID belongs to. Our ID-encoding scheme allows $2^{14} = 16384$ different equivalence classes in programs. This is sufficient for even large programs, as shown in our experiments. The version number in an ID is to support transactions and is used to detect whether a check transaction should be aborted and retried. The ID-encoding also allows 2^{14} different version numbers. We will discuss how transactions are implemented and why we believe 2^{14} version numbers are sufficient in Sec. 5.2.

We next discuss how MCFI represents Bary and Tary tables during runtime. Since they are queried frequently, MCFI should choose an appropriate data structure to minimize the ID-access time. There is a range of data structures MCFI could use. A simple approach is to use a hash map that maps from addresses to IDs. This is space efficient, but the downside is that a table access involves many instructions for computing the hash function and even more when there is a hash collision.

MCFI adopts a simple representation of the ID tables. Both Bary and Tary tables are represented using arrays. The Tary table is an array of IDs indexed by code addresses. If a code address is not a possible indirect-branch target, then the corresponding array entry contains all zeros; otherwise, it contains the ID of the code address. This design clearly enables efficient look-ups and updates, but one worry is its space efficiency.

In the case that there is an entry in the table for every code address, the size of the table is four times of the code size since each ID is four-byte long. To have a smaller Tary table, MCFI uses a space-optimization technique. It inserts extra no-op instructions into the program to force indirect-branch targets to be four-byte aligned. As a result, the table needs entries only for four-byte aligned code addresses, and the size of the Tary table is the same as the code size. During runtime, since the majority of memory consumed by a program holds runtime data in the heap, the Tary table causes only a small increase on the runtime memory footprint.

Moreover, MCFI has to prevent programs from using indirect-branch targets that are not four-byte aligned. This is where those reserved bits in an ID help[1]. In particular, if an indirect branch uses an address that is not four-byte aligned, then the four-byte target ID loaded from the Tary table will not be valid (i.e., it will not have the special bit values 0, 0, 0, and 1 in the least-significant bits). Then the comparison with the branch ID will fail because the branch ID loaded from the Bary table is always valid, as discussed next.

The Bary table could use the same design as the Tary table, but MCFI uses an optimization to increase its space and time efficiency. Recall that the Bary table conceptually maps indirect-branch locations to branch IDs. One observation is that instruction addresses are known once they are loaded in memory. Therefore, when a module is loaded into the code region, MCFI's loader patches the code to embed constant Bary table indexes that correspond to correct branch IDs in branch-ID read instructions. In this design, the Bary table does not need entries for code addresses that do not hold indirect branches (in contrast, the Tary table has all-zero entries even for addresses that are illegal indirect-branch targets). Furthermore, all branch IDs loaded from the Bary table are valid IDs as long as the loader embeds the correct table indexes in branch-ID read instructions.

Finally, the tables need to be protected at runtime so that application code cannot directly change them. We adopt MIP's design [16] to restrict the memory region where application code can write so that it cannot directly modify the tables. A brief summary of this code sandboxing technique is as follows. On x86-32, memory segmentation is used, as in NaCl [22]. A 1GB segment is reserved for running the application code and another 1GB segment is reserved for the table region. x86-64, however, does not support memory segmentation. Instead, memory writes are instrumented so that they are restricted to the [0, 4GB) memory region. Another 4GB memory region is reserved for tables. To access table entries efficiently, a spare segment register is used to be the base address of the table region: `%fs` is used in x86-32 and `%gs` in x86-64.

[1] Alternatively, we can insert an `and` instruction to align the indirect-branch targets by clearing the least two bits, but it incurs more overhead.

```
1  void TxUpdate () {
2    acquire(updLock);
3    globalVersion = globalVersion + 1;
4    updTaryTable();
5    sfence;
6    updBaryTable();
7    release(updLock);
8  }
9  void updTaryTable() {
10   // allocate a table and init to zero
11   allocateAndInit(newTbl);
12   for (addr=CodeBase;addr<CodeLimit;addr+=4) {
13     ecn=getTaryECN(addr);
14     if (ecn >= 0) {
15       entry=(addr - CodeBase) / 4;
16       newTbl[entry]=0x1; // init reserved bits
17       setECNAndVer(newTbl, entry,
18                    ecn, globalVersion);
19     }
20   }
21   copyTaryTable(newTbl, TaryTableBase);
22   free(newTbl);
23 }
```

Figure 3. Pseudo code for implementing update transactions.

5.2 Table Transactions

MCFI could adopt standard STM algorithms to implement the transactions. However, those algorithms are generic and separate meta-data (e.g., the version numbers) from real data (the ECNs). As a result, they require multiple instructions for retrieving meta-data and real data, and multiple instructions for comparing meta-data and real data to check for transaction failure and CFI violation. We micro-benchmarked the TML [6] algorithm, a state-of-the-art STM algorithm particularly optimized for read-dominant workloads, and found its slowdown is around 100% more than MCFI's custom transaction algorithm, which puts meta-data and real data in a single word. The compact representation enables MCFI to use a single instruction to retrieve both meta- and real data and a single instruction to check for transaction failure.

Update transactions. When a library is dynamically linked, MCFI produces a new CFG for the program after linking. Sec. 6 presents MCFI's method for generating the new CFG; in this section we assume such a method is there. Based on the new CFG, a new set of Equivalence Class Numbers (ECNs) is assigned to equivalence classes induced by the new CFG. In the rest of this section, we assume the existence of two functions that return the new ECNs: (1) `getBaryECN` takes a code address as input and, if there is an indirect branch at that address, returns the branch ECN of the indirect branch; it returns a negative number if there is no indirect branch at the address; (2) `getTaryECN` takes a code address as input and, if the address is a possible indirect-branch target, returns the address's ECN (i.e., the ECN of the equivalence class that the address belongs to); it returns a negative number if the code is not a possible indirect-branch target.

Fig. 3 presents the pseudocode that implements update transactions. It is implemented inside MCFI's runtime and is used by MCFI's dynamic linker to update the ID tables. An update transaction starts by acquiring a global update lock and incrementing a global version number. The lock is to serialize update transactions among threads. This simple design takes advantage of the fact that update transactions are rare in practice and allowing concurrency among update transactions does not gain much efficiency. We note that the global update lock does not prevent concurrency between update transactions and check transactions.

```
1  TxCheck {
2    popq  %rcx
3    movl  %ecx, %ecx
4  Try:
5    movl  %gs:ConstBaryIndex, %edi
6    movl  %gs:(%rcx), %esi
7    cmpl  %edi, %esi
8    jne   Check
9    jmpq  *%rcx
10 Check:
11   testb $1, %sil
12   jz    Halt
13   cmpw  %di, %si
14   jne   Try
15 Halt:
16   hlt
17 }
```

Figure 4. Implementation of check transactions for x86-64 return instructions.

The update transaction performs table updates in two steps: first update the Tary table, and then the Bary table. The separation of the two steps is achieved by a memory write barrier at line 5, which guarantees that all memory writes to Tary finish before any memory write to Bary. Bary and Tary table updates cannot be interleaved; otherwise, at some intermediate state in an update transaction, some IDs in the Tary and Bary tables would have the old version and some IDs would have the new version. Consequently, check transactions would use different versions of CFGs for different indirect branches. By updating one table first before updating the other, check transactions either use the old CFG or the new CFG for all indirect branches at all times.

Function `updTaryTable` first constructs a new Tary table (line 11). Constants `CodeBase` and `CodeLimit` are the code region base and limit, respectively. The table construction process iterates each four-byte aligned code address, invokes `getTaryECN`, and updates the appropriate entry in the table. The auxiliary function `setECNAndVer` updates the table entry with the ECN and the global version number; its code is omitted for brevity. After construction, the new Tary table is copied to the Tary table region with the base address in `TaryTableBase` (line 21). The `copyTaryTable` implementation is critical to the performance of update transactions. An insight is that table entries can be updated in parallel; the only requirement is that each ID update should be atomic. Therefore, we use the weak order memory write instruction `movnti`, which directly writes data into memory without polluting the cache, to perform fast parallel copying.

Function `updBaryTable` performs similar updates on the Bary table with the help of `getBaryECN`; its pseudocode is omitted.

Check transactions. Check transactions run during the execution of indirect branches. For efficiency, MCFI implements a check transaction as a sequence of machine instructions and instruments an indirect branch to inline the sequence. The sequence is slightly different for each kind of indirect branches (i.e., returns, indirect jumps, and indirect calls). Further, it needs adaptation for different CPU architectures. We present the x86-64 sequence in this section. The implementation on x86-32 is similar and is omitted for brevity.

Fig. 4 presents how a check transaction is implemented in assembly for return instructions on x86-64. A return instruction is translated to a `popq`/`jmpq` sequence (lines 2 and 9); this is to prevent a concurrent attacker from modifying the return address on the stack after checking. Instruction at line 3 operates on lower four bytes of `%rcx` and has the side effect of clearing the upper 32 bits of `%rcx`. As discussed, the sandbox is in the region of [0, 4GB); so

the instruction at line 3 restricts the return address to be within the sandbox. Instruction at line 5 reads the branch ID from a constant index in the Bary table. Instruction at line 6 reads the target ID from the Tary table. As discussed before, the Tary table starts at `%gs`.

Based on the values of the branch and target IDs, the following four cases may occur:

(1) If the branch ID in `%edi` equals the target ID in `%esi`, then instructions at lines 7, 8 and 9 get executed, performing the control transfer. In this case, the target-ID-validity check, the version check, and the ECN check are completed by a single comparison instruction, making this common case efficient.

(2) If the target address is not four-byte aligned or its corresponding Tary ID contains all zeros, then the target ID in `%esi` is invalid. Since the branch ID is always valid, the ID comparison fails. As a result, instructions at lines 7, 8, 11, 12, and 16 get executed and the program is terminated. In "`testb $1, %sil`", `%sil` is the lowest byte in `%esi` and the instruction tests whether the lowest bit in `%sil` is one. If it is not one, then we have a violation of the CFI policy because it uses a return address that cannot be a possible target.

(3) If the target ID is valid, but the branch ID in `%edi` has a different version from the target ID in `%esi`, instructions at lines 7, 8, 11, 12, 13, and 14 get executed, causing a retry of the transaction. This case happens when an update transaction is running in parallel. The check transaction has to wait for the update transaction to finish updating the relevant IDs.

(4) If the target ID is valid, and the versions of the two IDs are the same, but they have different ECNs, then instructions at lines 7, 8, 11, 12, 13, 14, and 16 get executed and the program is terminated. This case violates the CFI policy.

Indirect calls and jumps can be instrumented similarly with minor adjustments.

Linearizabiliy. The two ID tables can be viewed as a concurrent data structure with two operations (check and update operations). One widely adopted correctness criterion in the literature of concurrent data structures is *linearizability* [11], meaning that a concurrent history of operations should be equivalent to a sequential history that preserves the partial order of operations induced by the concurrent history. Our ID tables are linearizable. In `TxUpdate`, the linearization point is right after the memory barrier at line 5. Before the point, `TxCheck`s respect the old CFG; after the point, `TxCheck`s respect the new CFG. In `TxCheck`, the linearization point is the target ID read instruction at line 6 when the valid target ID has the same version as the branch ID or the target ID is invalid.

The ABA Problem. MCFI's ID-encoding scheme supports 2^{14} versions and it might encounter the ABA problem [8]. For example, an attacker may load over 2^{14} modules and exhaust the MCFI's version number space. This is unlikely in practice, even for just-in-time compiled code. Security is violated only if the program has at least 2^{14} code updates during a check transaction. If this were a concern, MCFI could maintain a counter of executed update transactions and make sure it does not hit 2^{14}. After completion of an update transaction, if every thread is observed to finish using old-version IDs (e.g., when each thread invokes a system call), the counter is reset to zero. Further, MCFI could use a larger space for version numbers such as 8-byte IDs on x86-64.

Procedure Linkage Table (PLT). PLT entries are used for dynamic linking. A PLT entry contains an indirect jump whose target depends on the runtime adjustable Global Offset Table (GOT). To accommodate PLT and GOT, MCFI needs to make two adjustments. First, the GOT entries are dynamically adjusted from the address of the linker to the addresses of corresponding library functions; such GOT entry updates are inserted between line 5 and 6 in Fig. 3 and serialized by another memory write barrier. Second, indirect jumps in the PLT may potentially violate the CFI policy and need to be instrumented with a check transaction as well. The complication is that those indirect jump targets are in the GOT and dynamically adjusted by an update transaction. Therefore, the instrumentation for indirect jumps in the PLT needs to reload the target address from GOT when a transaction is retried.

6. Module Linking

In addition to code and data, an MCFI module also contains auxiliary information for CFG generation. When MCFI modules are statically or dynamically linked, not only are their code and data linked, but their auxiliary information is also merged into the combined module.

There is a range of choices for what kind of auxiliary information MCFI can attach to a module. The richer the auxiliary information is, the better it can enable the generation of a precise CFG. On the other hand, richer auxiliary information implies more analysis time is needed for generating and merging the information and for producing a CFG from the information. Since the dynamic linker cannot afford long analysis time, there is a tradeoff between the CFG precision and efficiency.

MCFI attaches type information to modules and uses type matching for fast online CFG generation. Specifically, an MCFI module comes with the types of its functions and its function pointers. The benefit of this design is that it can efficiently generate a relatively precise CFG compared to coarse-grained CFI. Moreover, the type information for an individual module can be generated by an augmented compilation toolchain. For this purpose, we modified LLVM, which propagates types from the source level to low level. Finally, combining type information of multiple modules during linking is a simple union operation.

Type-matching CFG generation. We next describe how MCFI generates a CFG for a module from its type information. Note that a module may be the result of linking several smaller modules (e.g., the result of linking the main module with a DLL module). Another note is that our current CFG generation assumes that the module is produced from C code. CFG generation for C++ modules would require handling additional control-flow mechanisms such as exceptions and dynamic dispatch.

The control-flow edges out of non-indirect-branch instructions can be computed from the code itself. We therefore discuss only the cases of indirect branches. For an indirect call through a function pointer whose type is $\tau*$, we allow it to call any function as long as (1) the function's address is taken in the code, and (2) the function's type is some τ' that is structurally equivalent to type τ. In structural equivalence, named types are replaced by their definitions. The structural equivalence rule may break some C code, meaning that the produced CFG may not include all control-flow edges necessary for the C code to run. We will later discuss sufficient conditions for C code not to break.

Indirect jumps can be classified into two categories: intraprocedural and interprocedural control transfers. The former is used by LLVM to compile `switch` or indirect `goto` statements. Their targets are organized in read-only jump tables, which are hard-coded into the program. Such indirect jumps are statically analyzed to determine their control-flow targets using information in the jump tables, suggested by [23]. The interprocedural indirect jumps implement indirect tail calls. We handle them using the same type-matching approach; that is, an interprocedural indirect jump is allowed to jump to any function whose type is τ, assuming the type of the function pointer used in the indirect jump is $\tau*$.

To compute control-flow edges out of return instructions, we construct a call graph, which tells how functions get called by direct or indirect calls. Tail calls are handled in the following way: if in

function f there is a call node calling g, and g calls h through a series of tail calls, then an edge from the call node in f to h is added to the call graph. Using the call graph, control-flow edges out of return instructions can be computed: if there is an edge from a call node to a function, then return instructions in the function can return to the return address following the call node.

There are also unconventional control flows to handle. We next discuss a few such issues. First, `longjmp` mostly returns (through an indirect jump instruction) to the address set up by a `setjmp` call. Our implementation simply connects the `longjmp`'s indirect jump to the return address of each `setjmp`. Second, C allows variable-argument functions. If a function pointer type $\tau*$ allows variable arguments, our implementation allows it to invoke any function whose address is taken, whose return type matches, and whose parameter types match the fixed parameter types in τ. For instance, suppose the function pointer has type "`int (*)(int, ...)`"; an indirect call through the pointer can invoke any function whose address is taken, whose return type is `int`, and whose first parameter type is `int`. Third, signal handlers do not return to the application's code; instead, it returns to a small code snippet that invokes the `sigreturn` system call. In our implementation, the code snippet is inlined into signal handlers and thus the return is eliminated. Fourth, inlined assembly code is sometimes used in C code. For instance, the `libc` library uses inlined assemblies to implement CPU-specific versions of `memcpy`. For such inlined assembly code, our system requires developers to add type annotation for function pointers and functions used in the assembly.

Conditions for type-matching CFG generation. We assume the input C program has been preprocessed to satisfy the following conditions before CFG generation.

C1 *No type cast to or from function pointer types.*

C2 *No assembly.*

Condition C1 disallows explicit or implicit type casts from or to function pointer types. For instance, it is not allowed to cast a function pointer of type "`void (*)(int)`" to type "`void (*)(char *)`" and make an indirect call through the resulting pointer. It implies only those functions whose addresses are taken can be indirect call targets. C1 includes implicit type casts involving function pointers, for example, when a union type includes a function pointer field or when a struct is cast to another struct which include a function pointer field whose type is incompatible with the one in the first struct. However, C1 does not include all type casts; only casts involving function pointer types are included. Violations of condition C2 requires adding type annotations to assembly code so that the same type-matching approach can be used.

We believe that for well-behaved C programs that satisfy the conditions the type-matching approach generates CFGs that do not break the code. The conditions essentially enforce a simple type system, ensuring that values of function pointer types cannot be forged. We leave the formal proof of this property to future work.

We investigated how much effort it takes to make SPECCPU2006 C programs to comply with the conditions. We implemented an analyzer on top of the StaticChecker framework of Clang, LLVM's front end. The analyzer over-approximates violations of the two conditions. Violations of condition C1 are caught easily because the LLVM's internal representation makes all type casts explicit. For C2, the analyzer just reports cases of inlined assemblies.

The analyzer found no violation of C2 in the twelve benchmarks. (However, the libc library we used includes inlined assemblies and we manually provided type annotations for them.) All violations found by the analyzer are in C1, and we summarize the results in Table 1. For a benchmark, column SLOC lists its lines of source code and column VBE (Violation Before false-positive Elimination) lists the number of C1 violations. While some benchmarks such as `gobmk` have no violations, two benchmarks, `perlbench` and `gcc`, have thousands of violations. We found many cases do not lead to actual violations of the CFG built by our system; that is, they are false positives.

SPECCPU2006	SLOC	VBE	UC	DC	MF	SU	NF	VAE
perlbench	126,345	2878	510	957	234	633	318	226
bzip2	5,731	27	0	0	6	4	0	17
gcc	235,884	822	0	0	15	737	27	43
mcf	1,574	0	0	0	0	0	0	0
gobmk	157,649	0	0	0	0	0	0	0
hmmer	20,658	20	0	0	20	0	0	0
sjeng	10,544	0	0	0	0	0	0	0
libquantum	2,606	1	0	0	0	0	0	1
h264ref	36,098	8	0	0	8	0	0	0
milc	9,575	8	0	0	3	0	0	5
lbm	904	0	0	0	0	0	0	0
sphinx3	13,128	12	0	0	11	1	0	0

Table 1. C1 violations in SPECCPU2006 benchmarks.

Some of the false positives have common patterns and can be easily ruled out by the analyzer. We next briefly discuss those cases: (1) *Upcast (UC).* C developers sometimes use type casts between structs to emulate features such as parametric polymorphism and inheritance. An abstract struct type is defined and it contains common fields for its subtypes. Then a few concrete struct types are physical subtypes of the abstract struct type (in the sense that they share the same prefix of fields). A function can be made polymorphic by accepting values of the abstract struct type. Callers of the function have to perform type casts. Those type casts are upcasts, which are false positives in our system because the extra fields in a concrete struct cannot be accessed after the cast. (2) *Safe downcast (DC).* Downcasts from an abstract struct type to a concrete struct type are in general not safe. However, a common pattern is to have a type tag field in the abstract struct; the runtime type tag encodes the type of a concrete struct when it is cast to the abstract struct. Clearly, if all casts involving the abstract struct type respect a fixed association between tag values and concrete struct types, then those casts can be considered false positives. Such association can be specified manually (or inferred from source code) and fed to the analyzer. (3) *Malloc and free (MF).* `malloc` always returns `void*`. If it is invoked to allocate a struct that contains function pointers, C1 is violated as it involves a type cast from `void*` to a struct with function pointers inside. We consider such violations false positives because if the function pointers inside the struct are used without proper initialization, the C program is not well behaved as it exhibits undefined behavior. Similarly, type casts in invocations of `free` are also considered false positives. (4) *Safe update (SU).* We consider updating function pointers with literals as false positives. For instance, function pointers may be initialized to be `NULL`, which involves a cast from integers to function pointers. This is a false positive as dereferencing a null value would crash the program. (5) *Non-function-pointer access (NF).* There are some type casts that involve function pointers but after casts the function pointers are not used. Take the following example from `perlbench`.

```
if (((XPVLV*)(sv->sv_any))->xlv_targlen) { ... }
```

Struct `XPVLV` has a function-pointer field, but after the cast only non-function-pointer fields are used. It is a false positive.

In Table 1, columns "UC", "DC", "MF", "SU" and "NF" list the numbers of false positives removed by our aforementioned elimination methods. Column "VAE" presents the number of cases after elimination. As can be seen from the table, the elimination methods are effective at eliminating a large number of false positives.

	perlbench	bzip2	gcc	libquantum	milc
K1	4	0	36	1	0
K1-fixed	4	0	22	1	0
K2	222	17	7	0	5

Table 2. Numbers of cases for the two kinds of violations.

After the process, seven benchmarks report no violations and need no code fixes. For the other five benchmarks, the remaining cases can be put into the following kinds:

K1 A function pointer is initialized with the address of a function whose type is incompatible with the function pointer's type.

K2 A function pointer is cast to another type and cast back to its original type at a later point.

Table 2 reports the number of K1 and K2 cases in the remaining five benchmarks. Row K1-fixed lists the number of cases in K1 that require changes to the source code to generate a working CFG using the type-matching method. None of the cases in K2 required us to change the source code.

Most K1 cases required us to change the source code manually because the unmatched types of function pointers and functions may cause missing edges in the generated CFG. Consider a case in the `gcc` benchmark that is related to a generic splay tree implementation. Each node in the splay tree has a key typed `unsigned long`. There is a key-comparison function pointer typed `int (*)(unsigned long, unsigned long)`. In two places, the function pointer is set to be the address of `strcmp`, whose type is `int (*)(const char*, const char*)`. Since the function pointer's type is incompatible with `strcmp`'s, the CFG generation does not connect the function pointer to `strcmp`. To fix the problem, we added a `strcmp` wrapper function that has the equivalent type as the type of the comparison function and makes a direct call to `strcmp`. The key-comparison function pointer is then set to be the address of the wrapper function. All cases in the "K1-fixed" row can be fixed by wrappers or by directly changing the types of function pointers or functions. We fixed all those cases with 6 lines of code changes for `perlbench`, about 30 lines for `gcc`, and 1 line for `libquantum`. There are 14 K1 cases in `gcc` that did not require us to patch the code. The reason is that the involved function pointers are never used (dead code) even though they are initialized with functions of incompatible types.

Four benchmarks report K2 cases. Consider an example in the `perlbench` program. A function pointer is initially stored in a `void*` pointer and later the `void*` pointer is cast back to the original function pointer's type and dereferenced. In `perlbench` and `gcc`, there are also cases of downcast without performing dynamic checking on type tags. In these cases, developers decided those downcasts are safe (perhaps through code inspection) to avoid dynamic checks. None of the K2 cases required code changes to generate a working CFG. This was confirmed by running instrumented benchmarks successfully with the provided data sets.

Our experience on SPECCPU2006 shows that the task of making source code work with the type-matching approach is not onerous and can be achieved with zero or small changes to the code. Furthermore, our empirical investigation suggests that only K1 cases are the ones that need fix.

Static and dynamic linking. In MCFI, separately compiled and instrumented modules can be statically linked together. MCFI's static linker changes the standard static linker with the phase of combing modules' auxiliary information. It also modifies the standard linker's PLT entry templates to emit MCFI-instrumented PLT entries in lieu of the original unsafe ones.

MCFI also allows a multithreaded program to load new libraries dynamically and transfer the control to the libraries' code. The dynamic linking is jointly performed by MCFI's dynamic linker, CFG generator, and runtime. The dynamic linker itself is instrumented by MCFI and runs within the sandbox, same as other program modules. Before any program module is loaded, the dynamic linker is first loaded in memory. The program modules' GOT entries are set to the dynamic linker's entry point. In detail, dynamically linking a library is performed in the following steps:

(1) **Module preparation**. A running program invokes MCFI's dynamic linker (by jumping to a PLT entry or invoking `dlopen`) to load a new library. The dynamic linker loads the library in the sandbox and sets the library code to be writable but not executable. Then, the linker analyzes the library and generates new PLT target addresses.

(2) **New CFG generation**. The linker invokes the CFG generator to generate a new CFG for the original program with the new library. PLT entries are connected to functions with matching names. New IDs are generated for the Bary and Tary tables. Further, the runtime patches the in-sandbox library so that the library's code has the Bary table indexes embedded in instructions that read branch IDs. Next, the code pages are set to read-only and statically verified to obey the CFI policy (the verifier will be discussed in the toolchain section). Then, the code pages of the library are set to be executable but not writable.

(3) **ID table updates**. The linker passes the new PLT target addresses to the runtime and executes an update transaction, adjusting the IDs in the tables as well as modifying entries in the GOT to use the new PLT target addresses.

7. MCFI's Toolchain

We have implemented an MCFI toolchain on x86-32/64 Linux. The toolchain includes a rewriter that performs program instrumentation, a static linker that combines modules and emits instrumented PLT entries, a CFG generator that collects auxiliary module information and constructs CFGs, a verifier that checks whether an MCFI module is instrumented to respect its CFG, a runtime system that loads and executes instrumented programs, and a dynamic linker that is invoked by the runtime to load libraries dynamically.

MCFI's rewriter (≈4000 lines of C++ code) is implemented inside the LLVM compilation framework (version 3.3). Three passes are inserted into LLVM's backend to reserve scratch registers used in `TxCheck` transactions, dump type information, and perform instrumentation. These three passes operate on LLVM's machine-dependent representation and execute after LLVM IR-level passes. The augmented LLVM framework is used to generate instrumented x86 machine code and auxiliary type information.

MCFI's CFG generator is a 500-line C++ program, implementing the CFG generation process described in Sec. 6. It takes a module's auxiliary type information and generates Bary and Tary tables. The type-based approach enables fast construction of those tables: it takes about 150 milliseconds for `gcc`, whose code size is about 2.7MB.

We have also implemented an independent verifier (≈4000 lines of C++ code on top of the LLVM instruction decoder) that performs modular verification of MCFI modules. The verifier takes an MCFI module, disassembles the module, and checks whether indirect branches are instrumented as required, memory writes stay in the sandbox (so that the tables are protected), and no-ops are inserted to make indirect-branch targets aligned. The auxiliary type information in an MCFI module enables the complete disassembly of the module because it tells all possible targets of indirect

branches (as described in the CFG generation process). The verifier removes the rewriter outside of the trusted computing base. We plan to prove the correctness of the verifier formally, similar to previous efforts of verifying SFI verifiers [14, 27].

MCFI's runtime loads and runs MCFI modules. The runtime is based on the MIP runtime [16] (with ≈200 lines changed). The runtime invokes the CFG generator to generate tables in memory. The runtime does not allow modules to directly invoke native system calls. Instead, it wraps system calls as API functions and checks their arguments. For instance, when `mmap` is invoked, the runtime checks the newly mapped memory cannot be both writable and executable. A similar restriction is placed on the `mprotect` system call. In essence, the runtime implements a form of user-space system-call interposition, similar to NaCl [22].

From a developer's perspective, it is cumbersome to port applications to run in MCFI's runtime, because the runtime only provides syscall-like APIs. Therefore, same as MIP, we ported a standard C library called MUSL[2] by changing its system-call invocations to MCFI runtime API invocations. The MUSL libc, which is also created as a module using the MCFI framework, is instrumented in the same way as other program modules. MUSL has about 64K lines of C code. Our analysis tool reports 45 C1 violations, in which 5 violations are K1 and the other 40 violations are K2. All the five K1 violations break the code and need to be fixed by the same approach of function wrappers or type adjustments, as discussed before.

MCFI's static linker is modified from Linux's standard linker (`ld`). MCFI's dynamic linker, modified from MUSL's linker, loads libraries at runtime. We have previously discussed the steps involved in dynamic linking.

8. Evaluation

Using its toolchain, we have evaluated MCFI on its performance overhead, the CFG generation process, and security. The evaluation was performed on SPECCPU2006 C benchmarks (with slight modifications as discussed before), including nine integer benchmarks and three floating-point benchmarks. All benchmarks were compiled at optimization level three. The experiments were conducted on a system with x86-64 Ubuntu 13.10, an Intel Core i7-3770 CPU at 3.8 GHz, and 8GB physical memory.

8.1 Overhead

MCFI slows down a program's execution. The slowdown is caused by two factors. First, since the program is statically instrumented, the extra checks and no-ops (for alignment) inserted in the program increase its execution time. Second, when the program dynamically loads a library, MCFI's runtime generates a new CFG and updates the ID tables using an update transaction. During an update transaction, check transactions running in parallel cannot finish until the relevant IDs are updated to the new version. Both the CFG-generation process and the delay on check transactions increase the execution time.

Execution overhead due to code instrumentation. We measured how much execution-time overhead MCFI imposes on benchmarks due to code instrumentation. All benchmarks were instrumented and tested on reference data sets three times with a maximum variance of 1.5%. Fig. 5 presents the percentage of execution-time increase for the benchmark programs. In this experiment, libraries are statically linked. Therefore, ID tables are not updated during program execution. Consequently, check transactions are not running in parallel with update transactions.

The table shows that the average overhead is around 4-6% on x86-32 and x86-64. It includes the overhead of executing check transactions and sandboxing memory writes. The overhead is comparable to other CFI systems. A recent CFI system [24] reports about 5.9% on x86-32 and 8.0% on x86-64; however, no separate compilation is supported. Coarse-grained CFI systems such as CCFIR [25], binCFI [26], NaCl [18, 22] and MIP [16] have around 5% overhead; however, they do not support fine-grained CFGs.

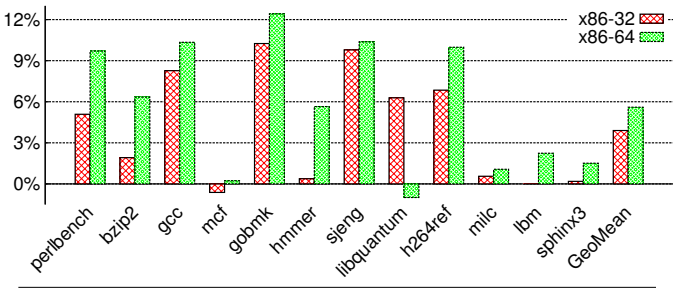

Figure 5. MCFI overhead on SPECCPU2006 C benchmarks. No update transaction is concurrently running.

The low overhead of MCFI's instrumentation seems surprising considering that a check transaction has two memory reads for reading branch and target IDs. The reason is that the two reads are executed in parallel by the CPU since there is no mutual dependency; this was confirmed by our micro-benchmarks.

Execution overhead due to code updates. As mentioned, in the previous experiment there is no parallel check and update transactions. It is rather difficult to design experiments with parallel check and update transactions for SPECCPU2006 benchmark programs because they load DLLs at the beginning of execution and do not use `dlopen` to load libraries. There are programs that dynamically load libraries in the middle of execution using `dlopen`. For example, a Java Virtual Machine (JVM) dynamically loads and unloads libraries according to what Java classes are running. However, we think the performance overhead imposed on those systems would be similar to Fig. 5 because library loading is a rare event. A rather extreme test for whether MCFI's transactions scale in a parallel environment is in a Just-In-Time (JIT) compilation environment, where code is generated and installed on-the-fly, and as a result, ID tables need to be updated frequently. However, our implementation has not covered a JIT environment yet.

In the absence of a JIT implementation, we designed a simulation experiment to test how MCFI's transactions scale. Specifically, we added a separate ID-table update thread when running SPECCPU2006 benchmarks. The thread simulates frequent ID-table updates. At a fixed interval, it performs an update transaction that updates the version numbers of all IDs in the ID tables (but preserving the ECNs). As a result, parallel check transactions are delayed. To determine the frequency of update transactions, we measured how frequently new code is installed in Google's V8, a JavaScript JIT execution engine. Based on that data, we set the frequency to be 50Hz. Fig. 6 presents the percentage of execution-time increase for the benchmarks. The average overhead is 6-7%, which demonstrates MCFI's transactions scale well with frequent code updates.

Space overhead. On average, MCFI increases the static code size by 17% on the benchmarks. During runtime it also requires extra memory as large as the code region to store the Bary and Tary tables. However, memory footprint increase is negligible because the majority of a program's runtime memory is occupied by its runtime data in the heap, which is much larger than the code region.

Evaluating MCFI's transaction algorithm. MCFI uses its custom transaction implementation. Using micro-benchmarks, we compared MCFI's transaction algorithm with other possibilities.

[2] http://www.musl.org. MUSL libc is a standard C library implementation.

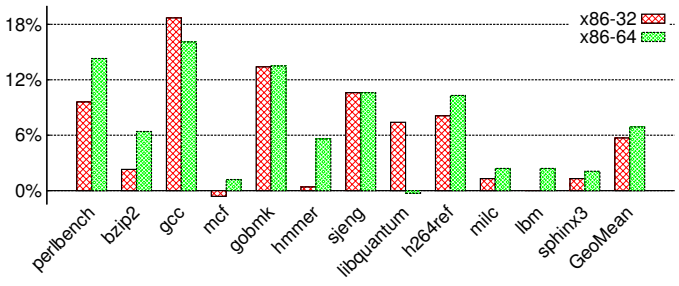

Figure 6. MCFI overhead on SPECCPU2006 C benchmarks. Update transactions are executed at the frequency of 50Hz.

The following table shows the micro-benchmark result of normalized execution time of check transactions implemented by TML [6], Readers-Writer-Lock (RWL) [2], or a mutex implemented by atomic Compare-And-Swap. TML, the most performant STM under read-dominated workload, doubles the execution time of MCFI, because it needs to read the global sequence lock before and after ID reads. RWL and Mutex, which use expensive LOCK-prefixed instructions, are much slower than MCFI's transactions.

	MCFI	TML	RWL	Mutex
Normalized Exec Time	1	2	29	22

8.2 CFG Generation

We measured the precision of the CFGs generated by MCFI. Table 3 lists the relevant statistics of benchmarks when they are statically linked with libc. For a benchmark, the "IBs" column lists the number of instrumented indirect branches. Since an indirect branch is allowed to target all addresses in an equivalence class, the number of indirect branches is also the upper bound of possible equivalence classes of addresses in a CFG. The "IBTs" column lists the number of possible indirect branch targets (i.e., functions whose addresses are taken and addresses following a call instruction). The "EQCs" column lists the number of equivalence classes of addresses in the benchmark's generated CFG using the type-matching approach. On x86-64, fewer equivalence classes are generated, mainly because more tail calls are replaced with jumps by LLVM's tail call optimization.

Compared to coarse-grained CFI techniques with several equivalence classes supported, MCFI's CFGs can generate two to three orders of magnitude more equivalence classes. For instance, CCFIR and binCFI allow an indirect call to target any function whose address is taken; therefore all such functions are included in one equivalence class. CCFIR and binCFI also allow any return instruction to target any instruction following a call, combining all return sites in one equivalence class. The classic CFI's instrumentation [3] can support a fine-grained CFG, but for implementation convenience its CFG generation also allows all indirect calls to target any function whose address is taken. NaCl and MIP enforce chunk-based CFI in which an indirect branch can target any chunk beginning; it enforces even less-precise CFGs.

8.3 Security

In an MCFI-hardened program, an indirect call targets only type-matched function entries and a return can jump to those return sites according to the generated call graph. Therefore, return-into-libc attacks are mitigated because attackers cannot redirect returns and function calls to arbitrary functions. In addition, since MCFI guarantees that only instructions appearing in the CFG are executed, a ROP (return-oriented programming) gadget starting in the middle of an instruction is eliminated. We measured gadget elimination by counting unique gadgets in the original benchmarks and MCFI-hardened ones using a ROP-gadget finding tool called `rp++`[3]. On average, MCFI can eliminate 96.93%/95.75% of ROP gadgets on x86-32/64.

SPEC 2006	x86-32			x86-64		
	IBs	IBTs	EQCs	IBs	IBTs	EQCs
perlbench	2250	15492	930	2081	15273	737
bzip2	220	515	110	217	544	93
gcc	5215	48634	2779	4796	46943	1991
mcf	170	468	119	174	445	106
gobmk	2734	11073	709	2487	10667	579
hmmer	726	4464	401	715	4369	353
sjeng	305	1457	207	337	1435	184
libquantum	246	754	161	258	702	121
h264ref	1099	3677	493	1096	3604	432
milc	441	2443	312	432	2356	264
lbm	161	455	112	161	426	96
sphinx3	585	2963	380	589	2895	321

Table 3. CFG statistics for SPECCPU2006 benchmarks.

To compare with other CFI techniques, we have also calculated the *Average Indirect-target Reduction* (AIR [26]) metric for the twelve SPECCPU2006 benchmarks. Intuitively, the AIR metric, which is a real number in [0, 1), measures how many indirect-branch targets are reduced on average by a CFI technique. A program without any CFI protection has an AIR value of 0 as an indirect branch can jump to any code address in the program. A CFI protection restricts indirect branches to a subset of code addresses. The more restriction a CFI technique places on indirect branches, the closer the AIR metric gets to 1. We summarize the AIR values of different CFI approaches in the following table.

	MCFI (32)	MCFI (64)	Classic CFI	binCFI	NaCl
AIR (%)	99.99	99.97	99.16	98.91	96.15

The AIR values of MCFI on both x86-32 and x86-64 are computed while other values are derived from the data reported in [26]. Since CCFIR and MIP are capable of enforcing the same protection as binCFI, but weaker than the classic CFI, their AIR values should be between binCFI and the classic CFI. In comparison, MCFI produces the best AIR values.

AIR values aside, there are more evidence that suggests fine-grained CFI provides better protection than coarse-grained CFI. First, ROP attacks can still be launched for systems hardened with coarse-grained CFI [10]. Second, some attacks hijack a function pointer and use that function pointer to jump to a dangerous library function such as `execve`. This kind of attacks may still be possible under coarse-grained CFI, but not fine-grained CFI. For instance, under coarse-grained CFI, the vulnerability, CVE-2006-6235, of GnuPG allows a remote attacker to control a function pointer and jump to `execve`, whose address is taken when GnuPG is linked with the MUSL `libc`. If protected by MCFI, the function pointer cannot be used to jump to `execve` because their types do not match.

9. Conclusions

We have described MCFI, the first system extending fine-grained CFI with the support of separate compilation. MCFI instruments indirect branches to consult two ID tables that are the runtime representation of the program's Control-Flow Graph (CFG). When li-

[3] https://github.com/0vercl0k/rp

braries are dynamically linked, MCFI's runtime updates the ID tables according to a new CFG generated by a type-matching approach, which takes advantage of modules' auxiliary type information. For thread safety, table operations are put in transactions, which are supported by techniques inspired by Software Transactional Memory. Experiments demonstrate MCFI is efficient and provides stronger security than other CFI techniques.

Acknowledgments

We thank anonymous reviewers for insightful suggestions and also thank Axel Souchet for explaining `rp++`. This research is supported by NSF grants CCF-0915157, CCF-1149211, and two research awards from Google.

References

[1] LLVM. `http://llvm.org`.

[2] Simple, non-scalable reader-preference lock. `http://www.cs.rochester.edu/research/synchronization/pseudocode/rw.html`.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy (S&P)*, pages 263–277, 2008.

[5] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. Schuff, D. Sehr, C. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 355–366, 2011.

[6] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional mutex locks. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par'10, pages 2–13, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15290-2, 978-3-642-15290-0.

[7] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A. reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[8] D. Dechev. The ABA problem in multicore data structures with collaborating operations. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 158–167, 2011.

[9] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

[10] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Security & Privacy (Oakland)*, San Jose, CA, USA, May 2014. IEEE.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[12] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, pages 195–208, 2010.

[13] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th Usenix Security Symposium*, 2006.

[14] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: Better, faster, stronger SFI for the x86. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 395–404, 2012.

[15] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine, Volume 11, Issue 0x58, File 4 of 14*, 2001.

[16] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 199–210, 2013. .

[17] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *ACSAC '13: Proceedings of the 2013 Annual Computer Security Applications Conference*, 2013.

[18] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th Usenix Security Symposium*, pages 1–12, 2010.

[19] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[20] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, 1995.

[21] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*, pages 380–395, 2010.

[22] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (S&P)*, May 2009.

[23] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *18th ACM Conference on Computer and Communications Security (CCS)*, pages 29–40, 2011.

[24] B. Zeng, G. Tan, and Ú. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *22nd Usenix Security Symposium*, pages 369–382, 2013.

[25] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)*, pages 559–573, 2013.

[26] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd Usenix Security Symposium*, pages 337–352, 2013.

[27] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. Armor: Fully verified software fault isolation. In *11th Intl. Conf. on Embedded Software*. ACM, 2011.