# Monitor Integrity Protection with Space Efficiency and Separate Compilation

Ben Niu
Lehigh University
19 Memorial Drive West
Bethlehem, PA, 18015
ben210@lehigh.edu

Gang Tan
Lehigh University
19 Memorial Drive West
Bethlehem, PA, 18015
gtan@cse.lehigh.edu

## ABSTRACT

Low-level inlined reference monitors weave monitor code into a program for security. To ensure that monitor code cannot be bypassed by branching instructions, some form of control-flow integrity must be guaranteed. Past approaches to protecting monitor code either have high space overhead or do not support separate compilation. We present *Monitor Integrity Protection* (MIP), a form of coarse-grained control-flow integrity. The key idea of MIP is to arrange instructions in variable-sized chunks and dynamically restrict indirect branches to target only chunk beginnings. We show that this simple idea is effective in protecting monitor code integrity, enjoys low space and execution-time overhead, supports separate compilation, and is largely compatible with an existing compiler toolchain. We also show that MIP enables a separate verifier that completely disassembles a binary and verifies its security.

MIP is designed to support inlined reference monitors. As a case study, we have implemented MIP-based Software-based Fault Isolation (SFI) on both x86-32 and x86-64. The evaluation shows that MIP-based SFI has competitive performance with other SFI implementations, while enjoying low space overhead.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: [Software/Program Verification]; D.4.6 [**Operating Systems**]: [Security and Protection]

## Keywords

Control-Flow Integrity; Inlined Reference Monitors; Separate Compilation

## 1. INTRODUCTION

Inlined Reference Monitors (IRM [10, 9, 11]) are effective in enhancing software security. Low-level IRMs inline monitor code into low-level code such as assembly or binary code so that unsafe operations are checked before their execution. They have become popular in recent years thanks to their efficiency and broad applicability. A number of low-level IRMs have been described in the literature, including software-based fault isolation (SFI [20]), XFI [8], data flow integrity [5], fine-grained memory access control [2], and many others.

A basic requirement of a reference monitor is that the monitor mechanism must be tamper-proof. In the case of IRMs, we must have *monitor code integrity*: monitor code before unsafe operations cannot be bypassed by direct or indirect branches. In the presence of attacks that modify the control flow of a program, some form of Control-Flow Integrity (CFI) must be guaranteed to avoid the bypassing of checks including via jumping into the middle of an instruction, exploited in Return-Oriented Programming (ROP [18]) attacks on x86. Monitor code integrity is easy to enforce on direct branches, whose targets can be statically determined. The difficulty is how to restrict indirect branches, including indirect calls (e.g., function calls via registers), indirect jumps, and return instructions.

One way for monitor code integrity is through *aligned-chunk CFI* [14], which provides coarse yet sufficient granularity. In this approach, the code region is divided into chunks of the same size such as 16 bytes. Branch targets are aligned at chunk beginnings. All branches are restricted to target beginnings of chunks. For indirect branches, this is achieved by dynamic checks. Thanks to the restriction on branches, monitor code before unsafe operations cannot be bypassed as long as they stay in the same chunk. This form of CFI is coarse-grained in that indirect branches are allowed to target all chunk beginnings. It is easy to implement, efficient, and sufficient to guarantee that monitor code is always invoked. It has been adopted by PittSFIeld [14] and NaCl [22, 17].

However, for alignment of branch targets, aligned-chunk CFI requires the insertion of no-op instructions. For instance, the address after a function call must be an aligned address since it is the target of the return instruction in the callee function. If the address is not aligned, no-ops are inserted before the function call so that it becomes aligned after instrumentation. The extra no-ops slow down program execution. NaCl-JIT [4], a NaCl version that supports just-in-time compilation, reports no-ops account for half of the sandboxing cost (about 37% slowdown because of no-ops on x86-64). More importantly, the no-ops cause significant code-size blow up. For example, the code size after instrumentation in NaCl-x86-64 [17] is roughly 60% larger than the original program. Since NaCl applications are downloaded into a client browser on the fly, large code size implies long delay in starting up the application. Serious concerns were raised by developers about the large code-size increase on NaCl's mailing list [12].

Another way for monitor code integrity is through *fine-grained CFI*, in which the program is instrumented to enforce a given control flow graph (CFG). Once the CFG is enforced, static analysis can be performed based on the CFG to check whether monitor code can be bypassed. Various instrumentation techniques for fine-grained CFI have been proposed in the literature [1, 21, 2]. The

*CCS'13,* November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516649.

classic CFI instrumentation [1] inserts IDs before branch targets and inserts ID checks before indirect branches to ensure that they jump to targets specified in the CFG.

The fine-grained CFI provides stronger security, but also comes with higher runtime overhead. Furthermore, unlike aligned-chunk CFI, none of the instrumentation techniques for fine-grained CFI supports *separate compilation*. The term separate compilation traditionally refers to the ability of a compiler to compile modules separately and link the compiled modules. In the context of IRMs, it refers to the ability to perform instrumentation of modules separately, without considering other modules, and to link instrumented modules into a working program. The instrumentation of fine-grained CFI techniques requires all modules of an application, including libraries, to be available during instrumentation time. For instance, the IDs in the classic CFI instrumentation cannot overlap with numbers already in the program. This property cannot be guaranteed without inspecting the whole program. The loss of separate compilation means it is impossible to instrument libraries once and reuse those instrumented libraries for all programs. This is especially cumbersome for dynamically linked libraries (DLLs), which are built for reuse across programs. It implies that each program has to come with its own instrumented version of libraries. We believe this is the major obstacle that prevents fine-grained CFI from being adopted in practice.

Therefore, the challenge is to design some form of CFI that is strong enough to ensure monitor code integrity, is space efficient, and supports separate compilation so that libraries can be reused across programs. We propose a simple yet effective approach we call *Monitor Integrity Protection* (MIP). The basic idea is to divide the code region of a program into variable-sized chunks and use a chunk table to record boundaries of chunks. Inter-chunk control flow with direct or indirect branches is restricted to target only chunk beginnings. The restriction is enforced on indirect branches by dynamic checks that look up information from a runtime representation of the chunk table. Compared to aligned-chunk CFI, MIP is more flexible in that chunks have variable sizes; as a result, there is no need to insert no-ops for alignment. Compared to fine-grained CFI, MIP supports separate compilation. We highlight our main contributions below:

- We show how the chunk-table approach can be made efficient and compatible with separate compilation. In particular, MIP encodes each entry in the chunk table as a single bit and uses a bitmap to store all bits. The bitmap representation enables efficient checks before indirect branches and yields a small memory footprint for the table representation during runtime. To support separate compilation, each module in a program is instrumented separately and a module-specific chunk table is produced. When modules are combined statically or dynamically, their chunk tables are merged.
- We have implemented a complete toolchain including a MIP rewriter, verifier, and dynamic linker. Using the toolchain and SPECCPU2006 programs, we have performed a careful evaluation of MIP's performance and code-size increase, as well as its support for separate compilation.
- On top of MIP, we have implemented SFI for both x86-32 and x86-64 platforms. We show in experiments that our MIP-based SFI has competitive performance relative to an industrial-strength SFI system [17], but with a much lower code-size increase.

The remainder of the paper is organized as follows. Sec. 2 discusses related work. Sec. 3 gives an overview of MIP. Sec. 4 presents MIP's major design choices and Sec. 5 elaborates on the implementation details. In Sec. 6, we present a case study in which MIP is used as the basis of an SFI implementation. Evaluation results are discussed in Sec. 7 and we conclude in Sec. 8.

## 2. RELATED WORK

In essence, MIP is a special form of CFI so that IRMs can build on top of it for monitor code integrity. It is inspired by many previous systems, including a large body of work on CFI and IRMs.

### 2.1 Control Flow Integrity (CFI)

In the introduction, we have motivated MIP by comparing briefly between MIP and some CFI systems. Next we present a more detailed discussion of CFI techniques and their comparison with MIP. In general, past CFI instrumentation techniques can be put into two broad groups: coarse-grained CFI and fine-grained CFI.

**Coarse-grained CFI.** In this group, an imprecise CFG is enforced on indirect branches. In the simplest case, all indirect branches share a common set of possible targets. In a slight 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). An example coarse-grained CFI is the aligned-chunk CFI, discussed in the introduction. Another example is CCFIR [25], which rewrites code to redirect all indirect branches into a new springboard code region. In the springboard region, roughly each direct and indirect branch target has an aligned entry that contains instructions that transfer the control flow. Indirect branches are checked to ensure they target springboard entries at runtime. binCFI [26] also enforces coarse-grained CFI on a binary program by replacing indirect branches with direct jumps that target an external routine, which consults an address translation map. The map's keys record all possible indirect branch targets in the binary and its values are direct jumps whose execution transfers the control to the actual target. Only if a target is within the map can the corresponding jump be triggered to perform the control transfer. The advantages of coarse-grained CFI are threefold: (1) its execution-time overhead is low; (2) its implementation is straightforward, because the building of its CFG does not require a sophisticated analysis; (3) it supports separate compilation. Aligned-chunk CFI has been extended to support just-in-time compilation [4], which generates code on the fly. Similarly, CCFIR allows each module to have its own springboard region and springboard regions of multiple modules can be combined during runtime. Moreover, binCFI supports separate compilation, since the address translation maps can be dynamically glued together. Our MIP is also a form of coarse-grained CFI and therefore enjoys the above benefits.

In contrast to past coarse-grained CFI systems, MIP is space efficient. Aligned-chunk CFI has a 60% space overhead [17] due to the extra no-op instructions inserted for chunk alignment. CCFIR does not require chunk alignment, but entries in its springboard region must start at aligned addresses (8-byte or 16-byte aligned). Its space overhead is about 33%, which is also on the high side. binCFI instruments the binary code and preserves a copy of the original code for referencing embedded data, thus its space overhead is nearly 139%. MIP's space overhead is around 15% thanks to its space-efficient encoding of the chunk table.

In addition to space overhead, MIP differs from previous coarse-grained CFI in its assumptions and instrumentation techniques. MIP requires compiler cooperation to produce a chunk table for every module. The chunk table ensures that a module can be completely disassembled. By contrast, CCFIR and binCFI work directly on binaries by first disassembling the code. Specifically, CCFIR disassembles Windows binaries aided by the attached relocation information. However, such information is not generally available on programs running on other OSes such as Linux; therefore CCFIR's disassembly would probably fail on such systems. binCFI adopts heuristics for disassembly and conservatively preserves the original code copy for unexpected data references. Although the CFI policy is enforced, the program's semantics might be changed. In addition, CCFIR's instrumentation is also much more involved than MIP, which requires instrumentation before only indirect branches. CCFIR further requires replacing all function addresses with corresponding addresses in the springboard region and requires instrumenting even direct function calls. Runtime performance comparison among MIP, CCFIR and binCFI will be discussed in the evaluation section.

**Fine-grained CFI.** In fine-grained CFI, each indirect branch has its own allowed jump targets, specified in a CFG. Fine-grained CFI provides stronger security and can prevent most attacks that induce illegal control-flow transfers. In addition, fine-grained CFI enables advanced static analysis for optimization and verification [24].

Different instrumentation techniques have been proposed for fine-grained CFI on x86 [1, 21, 2] and ARM [7]. As discussed before, the classic CFI [1] inserts a check instruction with an embedded ID before an indirect branch and inserts the same ID before all allowed targets of that instruction. HyperSafe [21] constructs a target table for every indirect branch. The table is statically constructed and contains all the entries that the indirect branch can jump to. The program is changed so that table indexes are passed in the program. Before an indirect branch, the index is converted into a target address based on entries in the table. WIT's CFI enforcement [2] uses a color table to record colors of control-flow targets. 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.

The fine-grained CFI provides strong security and the space overhead is small. On the other hand, they are more difficult to implement because precise CFG generation requires program analysis. More importantly, none of the techniques support separate compilation. In the classic CFI [1], IDs have to be unique so that they do not appear in the rest of the code. This requires the availability of all code at the instrumentation time. Similarly, HyperSafe and WIT use a whole-program analysis to construct their CFI tables. As a result, it is impossible to instrument libraries once and reuse them, which severely limits fine-grained CFI's adoption in practice.

## 2.2 Inlined Reference Monitors (IRMs)

CFI is a special IRM. There are many other kinds of IRMs. SFI has been studied extensively [20, 14, 22]. It rewrites untrusted code to prevent it from accessing memory outside of a designated data region and executing instructions outside of a designated code region. It isolates untrusted code in a separate fault domain, but does not provide fine-grained memory-access control. XFI [8] and a series of papers [2, 6, 3] by Microsoft Research at Cambridge study how to enforce fine-grained memory access control, which allows different access permissions for small buffers in the data region. In addition, Data Flow Integrity (DFI [5]) can prevent some non-control data attacks.

As a case study, we demonstrate how SFI can build upon MIP for monitor code integrity. It is clear that MIP is a general approach: all IRMs can build upon MIP, which enforces sufficient CFI for protecting the monitor.

## 2.3 Dynamic instrumentation

IRMs statically rewrite binaries to embed checks that enforce a security policy. In contrast, program shepherding [13] and software dynamic translation [16] dynamically modify and monitor untrusted applications to enforce a policy such as CFI. These systems can be viewed as interpreters with embedded reference-monitor checks (together with a set of techniques for fast dynamic translation and for removing unnecessary checks). Compared to IRMs, the downside is that the whole dynamic optimization and monitoring framework is in the TCB. The performance overhead of program shepherding is also higher.

## 3. OVERVIEW

Before a detailed discussion of MIP's design and implementation, we present the main ideas and benefits of MIP. At a high level, MIP moves from equal-sized chunks in aligned-chunk CFI to variable-sized chunks. As a result, the beginnings of chunks have to be remembered in a chunk table. For reliable disassembly and separate compilation, each *MIP module* (e.g., a DLL library) contains a chunk table as well as its code and data. Multiple such modules compose an application. In addition, control flow has to respect chunks in the following way:

1. Inter-chunk control flow edges must target chunk beginnings or a known safe exit. This property is enforced on direct branches statically and on indirect branches through instrumentation code that validates the target by consulting the chunk table before the control transfer.
2. Within each chunk, intra-chunk control flow because of direct branches is allowed, but indirect branches are not allowed to produce intra-chunk control flow.

This approach yields several benefits. First, it supports reliable whole-module disassembly, because a module's chunk table tells exactly all possible indirect-branch targets. Intra-chunk control flow can also be completely determined statically as it can only be the result of direct branches. Consequently, a standard recursive-traversal disassembly algorithm [15] can be performed on each chunk to extract the intra-chunk CFG. Monitor code before an unsafe operation cannot be bypassed as long as the monitor code and the unsafe operation stay in the same chunk and the monitor code dominates the unsafe operation according to the chunk's intra-chunk CFG. Thanks to reliable disassembly, a MIP module can be statically verified to determine whether it is safe according to an IRM policy. Further, reliable disassembly allows MIP to support mixed code and data. On x86, read-only data, such as jump tables and alignment bytes, are sometimes embedded in the code region, and the data must not be recognized as code to avoid instrumentation that might change the program semantics.

Second, MIP is space efficient. Because chunks can have variable lengths, MIP does not need to insert extra no-op instructions for chunk alignment, the main reason for the large space overhead in aligned-chunk CFI. MIP needs space to represent the chunk table and checks are inserted before indirect branches to enforce chunk boundaries. MIP's implementation chooses a space-efficient data structure (a bitmap) to represent the chunk table. As we will see in the evaluation section, MIP's space overhead and execution-time overhead are low.

Finally, MIP supports separate compilation. Each module is separately compiled and instrumented according to a specific IRM. The instrumentation process produces the module's chunk table, which must be respected for the IRM's monitor code integrity. When modules are statically or dynamically linked, the MIP toolchain merges their chunk tables in addition to their code and data.

On the flip side, MIP asks for the cooperation of code producers to attach chunk tables to modules. Consequently, MIP is not designed to work on off-the-shelf stripped binaries, but provides support to code producers who intend to harden their applications using IRMs. The requirement of chunk tables is not an onerous task for a code producer. As we will show, chunk tables can be produced by information in standard assembly files generated by an unmodified compiler such as GCC. By contrast, fine-grained CFI requires much more cooperation to produce a CFG.

## 4. MIP DESIGN

MIP's design will be discussed in several steps. We will first have a brief discussion about its threat model. We then present MIP's instrumentation, including how the chunk table is implemented and the instrumentation before indirect branches. Discussion about the support for separate compilation is presented next, followed by description about MIP's verification process, which statically verifies code before running it.

Before proceeding, we stress that MIP's goal is to support other IRMs: preventing IRM instrumentation from being bypassed. Therefore, some MIP components are parametrized by an IRM. For instance, MIP's verification should invoke an IRM-specific verifier, in addition to the verification about MIP's own instrumentation.

### 4.1 Threat model

MIP adopts a concurrent-attacker model originated in CFI [1]. It is realistic and conceptually simple. Specifically, it models the attacker as a separate thread, which runs in parallel with the user program. The attacker thread can read and write any memory (subject to memory page protection). Therefore, the attacker can corrupt data memory between any two instructions in the user program. However, it is assumed that machine registers cannot be directly modified by the attacker thread and they provide a safe place for storing values.

In addition, MIP's runtime enforces the invariant that no memory regions are both writable and executable at the same time and all code in executable regions has been statically verified to obey the IRM policy. Without the invariant, MIP cannot prevent arbitrary code execution. The invariant is enforced when an application is initially loaded by the runtime. The runtime sets up a separate code and data region. After verification, 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 is also enforced when loading DLLs. Libraries, after verification, are loaded into unoccupied parts in the code region. Sec. 4.3 will discuss more about the support for libraries.

### 4.2 Instrumentation

MIP uses a chunk table to remember boundaries of chunks and queries the chunk table to ensure that branches respect chunks. Next, we discuss a few design choices such as how the chunk table is represented.

**Chunk table representation.** The representation should meet two needs: (1) it is space efficient; (2) it supports time-efficient queries about whether an address is the beginning of a chunk. One candidate would be a balanced binary tree where tree nodes are chunk beginnings. This data structure would have low space overhead. However, the query time is logarithmically dependent on the number of chunks. Another representation choice would be a hash table that remembers the set of chunk beginnings. For instance, hash tables were used by an early SFI implementation for checking jump targets [19]. Although the query time of a hash table is usually constant, calculating the hash function may involve dozens of instructions and the collision cost is even higher.

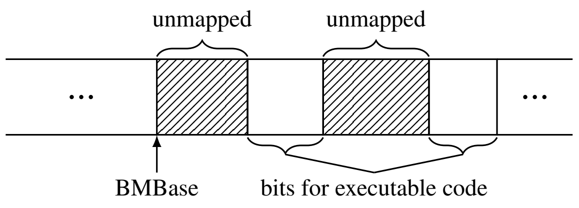

**Figure 1: Bitmap memory layout.**

MIP instead uses a bitmap to represent the chunk table. Conceptually, for every possible address, there is a corresponding bit in the bitmap. The bit is one if and only if the address is the beginning of a chunk. This representation clearly supports efficient queries. To check whether an address is the beginning of a chunk, just extract and test the bit based on the address and the start address of the bitmap (which can be achieved by a few machine instructions).

One worry about the bitmap representation is its space overhead. If every address has a bit in the bitmap, then its size is one eighth of the address-space size. However, notice that only those addresses belonging to executable code need to be represented by the bitmap. So one optimization is to represent bits for those addresses only in the code. Since the size of code is small compared to the address-space size, this optimization makes the bitmap compact. On the other hand, the code-address-only bitmap seems to require bounds checks. That is, given an address, we have to check that the address falls into the code region because otherwise we would retrieve a bit outside the code-address-only bitmap.

To avoid bounds checks, our solution is to reserve one eighth of the virtual address space for the complete bitmap, but leave those memory pages that do not correspond to the code region unmapped. This is depicted in Fig. 1. The bitmap has a start address `BMBase` (BitMap Base), which is the start address of the complete bitmap. Only those memory pages in the bitmap corresponding to code are mapped. Therefore, when an address is used to query the bitmap, we can omit the bounds checks because an address outside of the code region will be safely trapped by OS's page protection.

This solution results in low impact on the memory footprint because the physical memory for the bitmap is one eighth of the code-region size (not one eighth of the address space size). Furthermore, it supports separate compilation (discussed later). Note the code region is not required to be contiguous. When a new DLL library is loaded, MIP's runtime just needs to populate the bits corresponding its part of the bitmap. Finally, we note the bitmap is made read only for the user program so that untrusted code cannot tamper with the bitmap. On the other hand, MIP's trusted runtime can extend the bitmap when loading DLLs.

**Checks before indirect branches.** Dynamic checks to query the bitmap are inlined before indirect branches. For now, we use an abstract instruction "`CheckDest Reg`", which checks if the target in `Reg` is a valid chunk beginning according to the bitmap; if not, it jumps to the exit. The implementation section will discuss how the abstract instruction is lowered into machine instructions. The check is inlined before indirect branches straightforwardly. For instance, an indirect jump through register `eax` in "`jmp %eax`" becomes "`CheckDest %eax; jmp %eax`". An indirect branch through a memory location is first translated into an indirect branch through a register by reading the destination in a register and then instrumented. Return instructions are translated into an equivalent pop and indirect jump-through-register sequence and instrumented.

**Chunk granularity.** We have not discussed what constitutes a chunk. There is actually a spectrum of choices. In the simplest case, we can have *instruction chunks*. Many IRMs classify instructions into safe and unsafe instructions and insert checks immediately before unsafe instructions. We define a pseudo instruction to be either a safe instruction, or an unsafe instruction preceded by its check. In instruction chunks, a chunk is a single pseudo instruction. Although coarse-grained, enforcing instruction chunks prevents jumps into the middle of an instruction, and prevents checks before unsafe instruction from being bypassed.

Another choice is to have *basic-block chunks*, in which a chunk is a basic block with sequential intra-chunk control flow. A basic block can have multiple instructions, creating opportunities for intra-chunk check optimizations (e.g., redundant check elimination). One step further is to have *leaf-function chunks*, in which a chunk can be a leaf function or a basic block. A leaf function is one that does not call other functions; it can have complex intra-chunk control flow. Non-leaf function chunks are not allowed because function return addresses are targets of return instructions and so new chunks have to be formed at function return addresses.

Chunk granularity does not affect monitor code integrity as long as checks dominate unsafe instructions in chunks. On the other hand, bigger chunks have better security because it limits the space of indirect jump targets. We have implemented all three kinds of chunks and will discuss them more in Sec. 7.1.

## 4.3 Separate compilation support

In MIP, each module can be separately compiled and instrumented. It is important to note that the instrumentation code for one module does not need to be adjusted when it is combined with other modules. A module's instrumentation code is independent from the contents in its bitmap; its only assumption is that there is a bitmap that starts at `BMBase` and that bitmap is conceptually a complete bitmap that contains bits for all addresses (even though some parts are unmapped). When modules are combined, only their bitmaps need to be combined. We next discuss this step for both static linking and dynamic linking.

Support for static linking is straightforward. MIP's static linker first uses the standard static linker to link code and data of modules. Then it merges modules' bitmaps according to the relative positions of their code sections.

Support for dynamic linking is more complicated. MIP's dynamic loader augments a standard dynamic loader and performs dynamic linking in several steps. In the first step, memory pages for a new module are allocated and code is copied into memory. In this step, the newly allocated code pages are not executable so that the new code cannot be executed before verification. This step also allocates and populates memory pages in the bitmap memory area for the new module; the new bitmap memory pages are made read only. The second step is the standard dynamic-linking step, in which modules' data such as the global offset table (GOT) are adjusted for linking. In the final step, MIP verifies that the new code satisfies the IRM policy and, if verification succeeds, changes the new code pages to be executable. More about MIP's support for dynamic linking will be in Sec. 5.2.

One might worry about the bitmap's thread safety as one thread may query the bitmap and another thread may concurrently modify the bitmap when it loads a DLL library. However, since loading a DLL library only modifies part of the bitmap that is disjoint from the old parts of the bitmap, the bitmap is thread safe even without synchronization between threads.

## 4.4 Verification

MIP's verification takes a MIP module's code and bitmap as input and checks whether chunk boundaries are respected by the code. In addition, for each chunk the verification builds an intra-chunk CFG and invokes a custom $\mathrm{IRMCheck}$ procedure on the CFG. That is, the verification is parametrized by an IRM's check procedure, which validates that there is sufficient instrumentation in the chunk according to the IRM's policy. At a high level, MIP's verification checks the following two properties on a module:

$P_1$*: An inter-chunk control flow edge targets a chunk beginning or a safe exit.*

$P_2$*: Within a chunk, there is sufficient instrumentation before unsafe instructions according to a custom IRM, and the instrumentation cannot be bypassed because of intra-chunk control flow.*

With the bitmap, both $P_1$ and $P_2$ are local properties that can be checked on a per-chunk basis. If all chunks satisfy $P_1$ and $P_2$, the whole module obeys the IRM policy based on the following inductive argument. We assume that the module is initially entered at a chunk beginning. According to $P_2$, within the initial chunk, any unsafe instruction will be checked by sufficient instrumentation at runtime. When the control transfers outside the chunk, the destination can be only another chunk's beginning or a known safe exit according to $P_1$. In the former case, the execution in the next chunk obeys the IRM policy because of $P_2$; in the latter case, the module terminates safely.

Algorithm 1 presents the MIPVerify algorithm, which shows how the two properties are checked. The algorithm takes a module's $code$ and $bitmap$ as input and raises error if the verification fails. $code$ is an array of bytes and $bitmap$ is an array of bits. For ease of presentation, we assume that the code starts at address zero. We use notation $|code|$ for the size of the code and similarly for $|bitmap|$.

The MIPVerify procedure scans the bitmap and identifies chunk boundaries. It then disassembles a chunk to produce an intra-chunk CFG. The intra-chunk CFG is then examined by $\mathrm{MIPCheck}$.

The $\mathrm{Disassemble}$ procedure performs two tasks on the chunk with beginning address $i$ and end address $j$: (1) build an intra-chunk CFG, whose nodes are instructions and edges represent control flow between instructions; (2) check that chunk boundaries are respected by instructions in the chunk. It is a worklist algorithm that performs a recursive-traversal disassembly. It assumes an external procedure $\mathrm{decodeOneInstr}(code, n)$, which performs the decoding of one instruction at address $n$ and raises error if the decoding fails. A few data structures are used in $\mathrm{Disassemble}$: $cfg$ stores a partial CFG, whose nodes are pairs of addresses and instructions that have been decoded and whose edges represent control flow between instructions; $worklist$ contains a list of addresses to be decoded; and $occupied$ is an array for remembering what addresses have been occupied by decoded instructions. The $occupied$ array is used to prevent a direct branch from jumping into the middle of an instruction. At line 19, if the address range of a new instruction overlaps with a previous instruction, an error is raised. Variable $succ$ is used to hold the direct successor addresses of the current instruction. If the instruction is not a control-flow instruction, $succ$ is a singleton set with the fall-through address (that is, $n'$ in the code); if the instruction is a direct branch, $succ$ contains all possible target addresses of the branch (two addresses if it is a conditional jump); if the instruction is an indirect branch, $succ$ is the empty set. For each address in $succ$, at line 23 the code checks whether it is an address in the chunk (intra-chunk control flow) or outside (inter-chunk control flow). For intra-chunk control flow, the edge is added to the CFG; if the destination address has not been processed (checked at line 25) and not in the middle of a previous instruction (checked at line 26), then it is added to the worklist. For

**Algorithm 1:** MIP's Verification Algorithm

```
1  procedure MIPVerify ($code, bitmap$)
2    if $|code| \neq |bitmap|$ then raise Error;
3    foreach $0 \leq i < |bitmap|$ do
4      if $bitmap[i] = 1$ then
5        $j$ := the first index after $i$ so that
6            $bitmap[j] = 1$ or $j = |bitmap|$;
7        $cfg$ := Disassemble($code, bitmap, i, j$);
8        MIPCheck($cfg$)
9  procedure Disassemble($code, bitmap, i, j$)
10   $cfg$.nodes := $cfg$.edges := $\emptyset$;
11   $worklist := \{i\}$;
12   foreach $k \in [i..(j-1)]$ do $occupied[k] := 0$;
13   while $worklist \neq \emptyset$ do
14     Remove an address $n$ from $worklist$;
15     $r$ := decodeOneInstr($code, n$);
16     $cfg$.nodes := $cfg$.nodes $\cup \{(n, r)\}$;
17     $n' := n + \text{size}(r)$
18     if $n' > j$ then raise Error;
19     if $occupied[n..(n'-1)]$ is not all 0 then raise Error;
20     foreach $k \in [n..(n'-1)]$ do $occupied[k] := 1$;
21     $succ$ := the set of direct successor addresses for $r$;
22     foreach $s \in succ$ do
23       if $i \leq s < j$ then
24         $cfg$.edges := $cfg$.edges $\cup \{(n, s)\}$;
25         if $s \notin cfg$.nodes then
26           if $occupied[s] = 1$ then raise Error
27           else $worklist := worklist \cup \{s\}$
28       else if $bitmap[s] \neq 1$ and $s \notin$ SafeExit then
29         raise Error
30 procedure MIPCheck($cfg$)
31   foreach instr $r \in cfg$.nodes do
32     if $r \in$ ForbiddenInstr then raise Error
33     else if $r \in$ IndirectBranch then
34       if $r$ is not dominated by CheckDest on $r$'s
         destination then raise Error
35   IRMCheck($cfg$)
```

inter-chunk control flow, the code checks whether the destination is a chunk beginning or a safe exit (at line 28).

The MIPCheck performs several checks. First, it rules out forbidden instructions. For example, a `ret` instruction should have been translated into a pop followed by an indirect jump, so it is forbidden. Direct system calls through `int` or `syscall` are also forbidden. Second, indirect branch instructions must be preceded by a `CheckDest` instruction so that targets are chunk beginnings. Finally, IRMCheck is invoked to enforce the IRM policy on the chunk.

A few points are worth discussion about the verification. First, it performs modular verification. Each module is verified separately. The modular verification enables MIP to verify DLL libraries during dynamic linking. Second, the intra-chunk CFG for a chunk may not contain all bytes in the chunk. If some bytes are not reachable during recursive traversal, then they must be data and are not reachable during runtime. This is how MIP supports mixed code and read-only data. Finally, the verification trusts neither the code nor the bitmap. If a module's bitmap is modified by an attacker to change the chunk boundaries in a way so that the IRM's policy is violated, then the verification fails. This is because the verification checks that code follows the chunks specified in the bitmap and each chunk is validated in a modular way. In other words, if the attacker-provided code and bitmap pass the verification, then the IRM's policy is guaranteed to hold.

## 5. IMPLEMENTATION

We next describe MIP's implementation on Linux for x86-32 and x86-64 platforms. We first discuss how `CheckDest` is lowered into machine instructions. We next present MIP's toolchain including the rewriting tool, the verifier, the runtime, and the DLL support.

### 5.1 Check lowering

Conceptually "`CheckDest Reg`" extracts the bit corresponding to address `Reg` from the bitmap, tests if the bit is one, and jumps to an exit label when the bit is zero. This task can be accomplished by two simple machine instructions:

```
bt Reg, BMBase
jnc exit
```

"`bt Reg, BMBase`" is an x86 instruction called bit test. It selects the bit in a bit string in memory with base `BMBase` at the bit position `Reg` and stores the value of the bit in the CF flag. Recall that `BMBase` is the base address of the in-memory representation of the bitmap. Since this bit-test instruction has a base operand that specifies a memory location, the offset operand in `Reg` can be a 32/64-bit value. Therefore, it extracts the `Reg`-th bit from the bitmap to the CF flag. After that, a simple "`jnc exit`" instruction is used to test the bit and jump to an exit label when the bit is zero. We call this sequence `btm` (bit test on memory).

The `btm` sequence is simple and space efficient, but its execution time varies with different x86 micro-architectures. We tested it on x86 micro-architectures from Intel Core 2 to Nehalem to Sandy-Bridge, the cost ratio of "`bt Reg, BMBase`" to a single-clock-cycle instruction varies between 3 and 8 (after isolating the memory cache effect). Therefore, its execution-time overhead may be a concern for applications with heavy use of indirect branches.

An alternative but longer sequence is presented as follow (with comments after #).

```
mov Reg, SR1    # SR1 = Reg
mov Reg, SR2    # SR2 = Reg
shr $3, SR1     # SR1 = Reg / 8
mov BMBase(SR1), SR1 # SR1 = BMBase[SR1]
and $7, Reg     # Reg = Reg % 8
bt  Reg, SR1    # test whether the bit is one
mov SR2, Reg    # Reg = SR2; restore Reg
jnc exit
```

In the sequence, `SR1` and `SR2` are scratch registers. It moves the byte where the bit is into `SR1` and performs a bit test on `SR1`. Different from the previous sequence, the bit-test instruction operates on a bit string in a register. Therefore we call this sequence `btr` (bit test on register). The `btr` sequence is longer, but its execution time is shorter than `btm` based on our testing. The reason is that the `bt` instruction in `btm` causes the instruction decoder to stall several cycles. In contrast, although `btr` contains multiple instructions, each of them almost never stalls the instruction decoder.

The `btr` sequence requires two scratch registers `SR1` and `SR2`. Our implementation tries to find dead general purpose registers for them. If no general registers are available, we use dead SSE registers instead.[1]

[1]SSE registers are usually used only in specialized vector and multimedia routines, but not elsewhere. In our experiments, we found

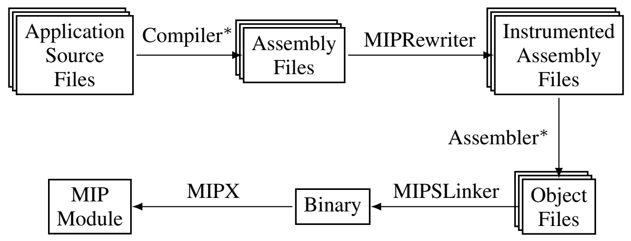

**Figure 2: The MIP toolchain for creating a MIP module. Standard Linux tools are marked with *; the rest are MIP added tools.**

## 5.2 MIP toolchain

We have built a MIP toolchain for rewriting and running x86 Linux applications. It operates at assembly level and takes advantage of symbolic information in assembly code for rewriting and for building the chunk table. As a result, it is compatible with any compiler such as GCC.

Fig. 2 visualizes the work flow of MIP's toolchain. Application source code is compiled into assembly code by a compiler such as GCC. The compiler produces meta information including labels and assembly directives that are embedded in assembly files. Afterwards, MIPRewriter performs assembly-level rewriting by extending the assembly code streamer of LLVM-MC, LLVM's assembling and disassembling component. MIPRewriter first transforms the assembly file's content into a stream of instructions, assembly labels and strings. On the stream, multiple rounds of rewriting according to different IRM instrumentation rules can be performed, with MIP's instrumentation being the last round. MIPRewriter provides a general interface for easily adding new IRM instrumentation. After instrumentation, MIPRewriter analyzes the stream for identifying the chunks of different kinds: instruction chunks, basic-block chunks, and leaf-function chunks. After MIPRewriter identifies a chunk, it inserts a unique label (called a MIP label) containing a special name before the first instruction of the chunk. In total, MIPRewriter's implementation includes around 2,800 lines of new C++ code on top of LLVM-MC.

Instrumented assembly files are then assembled by a native Linux assembler such as GNU AS into object files. The MIP labels inserted by MIPRewriter are transformed and stored in the symbol table and string table of the resulting ELF file, which effectively remembers the chunk beginnings.

The assembled object files are then fed to MIPSLinker for object linking. Modified from GNU LD, MIPSLinker emits an instrumented Procedure Linkage Table (PLT) for the symbol name resolution. The PLT contains indirect jumps whose targets depend on the runtime adjustable GOT entries. However, such indirect jumps may potentially violate chunk boundaries and need to be instrumented as well. Therefore, we changed around 60 lines of the LD's PLT entry template to emit instrumented indirect jumps. Note that each PLT entry is also a chunk with a fixed size. Except for this change, MIPSLinker performs exactly the same job as LD.

The binary created in the previous linking step essentially contains the module's chunk table—encoded as MIP labels and stored in the symbol table, string table and PLT section. Such information is extracted by a 120-line Python script MIPX to build the bitmap. Then a MIP module is assembled by composing the binary with the bitmap. We note that the process of creating an application's executable MIP module is the same as the process of creating a library MIP module.

Given an application's executable MIP module, MIPRunner loads the module into memory, invokes MIPVerify for verification, and executes the application. Thanks to the verification step, all the previous steps are outside of the TCB, including compilation, rewriting, assembling, and bitmap extraction. MIPVerify implements the algorithm presented in Section 4.4 and uses LLVM-MC's instruction decoding engine. MIPVerify's implementation has around 1,000 lines of C++ code.

MIPRunner is implemented on top of the NaCl runtime. We changed its application loading procedure so that it not only loads code, but also populates the bitmap. On x86-32, the application's code, data and bitmap stay in different segments from MIPRunner for isolation. On x86-64, MIPRunner allocates the lower 6GB with [4GB, 6GB) unmapped (described in detail in Section 6). In addition, since MIPRunner and application code run in the same address space on x86-64, for isolation MIPRunner unmaps the bitmap pages for its own code. Therefore, on both platforms, the application code is sandboxed and cannot directly jump to the MIPRunner code. In total, around 800 lines of C code were changed or added to the NaCl runtime code base.

The application's system call invocations must be intercepted. For example, if `mprotect` could be called without restriction, the application's code could arbitrarily change its memory protection. For monitoring system calls, we can either adopt an IRM or an external reference monitor. Since the NaCl runtime already provides a set of wrappers for security-sensitive system calls, we reuse that part for monitoring system calls. Therefore, application code is forbidden to invoke any system calls directly through `int` or `syscall` instructions. Instead, the relevant system calls are replaced by MIPRunner's system-call wrappers. For porting existing programs (especially the SPECCPU2006 benchmarks) to MIPRunner, we use a modified version of MUSL[2] standard C library, whose native system call invocation is replaced by MIPRunner's wrapper invocation. The MUSL library is also instrumented using the same IRM instrumentation as the application. Currently, C++ programs are not supported, since we have not ported any C++ libraries yet.

When the application is running, it may dynamically link and load DLLs. For dynamic linking, MIP provides a MIPDLinker, which is implemented by modifying MUSL's dynamic linker (for less than 100 lines of code). Our system could put MIPDLinker into MIPRunner and dynamically intercept `dlopen` and `dlsym` requests from the application. Instead, we put MIPDLinker into the application sandbox so that it is untrusted. The next three steps explain how MIPDLinker itself is loaded and how MIPDLinker loads a DLL library:

1. **Loading**. Before MIPRunner loads any application or library modules, it verifies and loads the dynamic linker MIPDLinker, which is also instrumented by the same IRMs as the application and libraries. Then, MIPRunner transfers control to MIPDLinker to further load other modules. When a module is loaded in memory, its memory pages are not executable.
2. **Linking**. When MIPDLinker finishes loading a module, it links that module with already installed modules that might be running. During the linking process, the already installed modules' code pages remain unchanged because MIPRunner restricts their protection to be not executable but writeable. However, all modules' GOT entries are subject to changes.
3. **Installation**. After linking, MIPDLinker invokes a MIPRunner provided API `InstallCode` for module installation.

that spilling general purpose registers into SSE registers imposed less overhead than spilling them into the stack.

[2]www.musl-libc.org

InstallCode first sets the new module code and bitmap to be read-only, forbidding MIPDLinker to further change the content. It then invokes MIPVerify for verifying the code and the bitmap. Once the verification succeeds, MIPRunner changes the protection of the new module's code pages to be executable.

## 6. MIP-BASED SFI

MIP provides control-flow support for an IRM to guarantee monitor code integrity. We next present a case study in which we implement Software-based Fault Isolation (SFI) on top of MIP. The SFI policy is to restrict control flow within a code region and restrict memory access within a data region. The control-flow part of the SFI policy is already enforced by MIP. Therefore, only memory accesses need to be restricted for the SFI policy. Direct memory accesses can be statically verified; indirect memory accesses should be restricted by dynamic checks. On x86-32, since the CPU supports memory segmentation that isolates memory access at the hardware level, our implementation allocates a data segment for holding application data (similar to NaCl-x86-32). On x86-64, since segmentation is not supported, we apply the following data-sandboxing scheme to control memory writes[3]:

- The sandbox range is [0, 4GB) with an unmapped guard zone [4GB, 6GB). The code and data region both reside in the sandbox with different memory protection.
- The stack pointer, `rsp`, is only used as an address reference register. Instructions changing `rsp` are replaced with equivalent ones changing `esp`. For example, `add $3, %rsp` is replaced by `add $3, %esp`. Because modifying `esp` automatically clears the upper 32 bits in `rsp`, the value of `rsp` is guaranteed to be within [0, 4GB).
- Memory writes to addresses `disp(%rsp)`/`disp(%rip)`, where `disp` is an immediate displacement and `rip` is the program counter, are not checked. The CPU accepts only a 32-bit signed integer displacement with a range of [-2GB, 2GB). Since `rsp` is dynamically restricted to be within [0, 4GB) and `rip` always points to the code region within [0, 4GB), memory references based on such forms could address [0, 6GB)∪[16EB-2GB, 16EB)[4]. This is safe because accesses to [16EB-2GB, 16EB) are trapped as it is used by the Linux kernel; accesses to [4GB, 6GB) are also trapped since it is the unmapped guard zone.
- Other memory writes are instrumented by dynamic checks to ensure the addresses are within [0, 4GB). Take instruction "`mov $3, (%rax)`" as an example. It writes constant 3 to the memory address stored in `rax`. This instruction is changed to "`lea (%rax), %ebx`", followed by "`mov $3, (%rbx)`". It uses `rbx` as a scratch register. The `lea` instruction automatically clearly the upper 32 bits of `rbx`, ensuring the target address is within [0, 4GB). Other memory writes are instrumented in the same way.

With the help of MIP, the SFI implementation is minimal. The implementation adds an SFI rewriting pass (about 200 lines of code) to MIPRewriter and adds an SFI check procedure (also about 200 lines of code) to MIPVerify.

[3]Similar to NaCl-x86-64, our implementation does not confine memory reads since they are considered less harmful than writes; this also enables a straightforward comparison between our SFI implementation and NaCl-x86-64.

[4]16EB = $2^{64}$.

## 7. EVALUATION

We evaluated the space and time efficiency, and the support for separate compilation of MIP and MIP-based SFI. The experiments were conducted on both x86-32 and x86-64 platforms. In SPEC-CPU2006, only C benchmarks were included[5], with nine integer performance testing programs and three floating-point performance testing programs. All programs were compiled with the O3 optimization level. Experiments were performed on a system with 64-bit Ubuntu 12.10, an Intel Core i7-3770 CPU at 3.8 GHz, and 8GB physical memory.

### 7.1 Chunk granularity

We have discussed three kinds of chunks: instruction chunks, basic-block chunks and leaf-function chunks. We experimented with the three kinds of chunks on benchmark programs and measured the number of chunks in each case. Table 1 lists the numbers.

As seen in the table, the number of basic-block chunks is about an order of magnitude less than the number of instruction chunks, because a basic block contains multiple instructions. The number in the case of leaf-function chunks is less than the case of basic-block chunks, but the difference is small. As discussed before, a non-leaf function still needs to be split into basic blocks because of return targets in the middle of the function. Therefore, the number of basic blocks dominates the number of chunks even in the leaf-function chunk case. Nevertheless, there are still many leaf functions in a typical program. Table 1 includes columns that present the percentage of leaf functions among functions.

The coarse-grained CFI enforced by MIP is good for security because it disallows ROP gadgets that start in the middle of a chunk. We experimented with a ROP gadget finding tool called ROPgadget[6] to search for gadgets in benchmark programs. Table 2 lists the unique gadgets found in original and MIP-instrumented benchmarks (with the `btm` instrumentation) for the case of instruction chunks. In MIP instrumented programs, although ROPgadget can still find gadgets, none of them can be exploited, because they all start in the middle of instruction chunks. This rather surprising finding suggests even extremely coarse-grained CFI can significantly limit ROP attacks. Nevertheless, we believe basic-block and leaf-function chunks provide better security because any single gadget-finding tool cannot claim completeness and other kinds of attacks may be possible within the limit of a CFG.

Basic block and leaf-function chunks also lead to sparser bitmaps. Although our implementation does not take advantage of this, a MIP module could have a compressed bitmap, which could decrease the binary size. Experiments showed that basic-block/leaf-function chunk bitmaps have a compression ratio (by `gzip`) of about 80% while the instruction-chunk bitmaps have around only 40%.

Since our implementation uses uncompressed bitmaps, the space overhead and execution-time overhead of a program running under MIP are independent of which kind of chunks is used. For a given program and a given data set, the same execution path will be executed no matter which bitmap is used (assuming no control-flow violation happens during runtime). Consequently, the same instru-

[5]MIP's basic idea of using variable-sized chunks to restrict indirect branches clearly applies to C++ code. However, a code-sandboxing framework needs a system-call interposition mechanism to prevent/restrict dangerous system calls such as `mprotect`. Our system implements user-space system-call interposition, which replaces libraries' system calls by their wrappers. Therefore, the only part in our implementation that does not support C++ is the library support, as discussed in Sec 5.2.

[6]http://shell-storm.org/project/ROPgadget/

| SPECCPU2006 | x86-32 Instruction chunks | x86-32 Basic-block chunks | x86-32 Leaf-func chunks | x86-32 Leaf function proportion | x86-64 Instruction chunks | x86-64 Basic-block chunks | x86-64 Leaf-func chunks | x86-64 Leaf function proportion |
|---|---|---|---|---|---|---|---|---|
| 400.perlbench | 251256 | 43732 | 43574 | 8.0% | 237917 | 43974 | 43787 | 10.8% |
| 401.bzip2 | 16936 | 1697 | 1684 | 14.5% | 15352 | 1670 | 1657 | 14.5% |
| 403.gcc | 726919 | 125363 | 123599 | 10.3% | 653678 | 122879 | 121248 | 11.9% |
| 429.mcf | 2898 | 435 | 353 | 41.7% | 2567 | 436 | 351 | 41.7% |
| 445.gobmk | 197921 | 29310 | 28987 | 6.6% | 188784 | 28828 | 28527 | 9.6% |
| 456.hmmer | 69328 | 10120 | 9808 | 15.9% | 63757 | 10055 | 9736 | 16.3% |
| 458.sjeng | 26454 | 3984 | 3728 | 34.3% | 24863 | 3964 | 3705 | 36.6% |
| 462.libquantum | 9359 | 1098 | 1082 | 18.9% | 7927 | 1075 | 1061 | 18.9% |
| 464.h264ref | 197092 | 14976 | 13135 | 24.3% | 177489 | 14992 | 12969 | 24.9% |
| 433.milc* | 38276 | 4998 | 4916 | 32.1% | 32748 | 5138 | 5060 | 31.7% |
| 470.lbm* | 2263 | 166 | 119 | 23.5% | 1901 | 165 | 119 | 23.5% |
| 482.sphinx3* | 41532 | 6333 | 6062 | 20.0% | 38055 | 6305 | 6042 | 21.3% |

**Table 1: Numbers of chunks in SPECCPU2006 benchmarks. Floating-point benchmarks are marked by *.**

| SPECCPU2006 | Original | After MIP | Exploitable |
|---|---|---|---|
| 400.perlbench | 242 | 148 | 0 |
| 401.bzip2 | 73 | 53 | 0 |
| 403.gcc | 415 | 285 | 0 |
| 429.mcf | 73 | 36 | 0 |
| 445.gobmk | 195 | 119 | 0 |
| 456.hmmer | 124 | 83 | 0 |
| 458.sjeng | 106 | 73 | 0 |
| 462.libquantum | 77 | 50 | 0 |
| 464.h264ref | 169 | 107 | 0 |
| 433.milc* | 103 | 67 | 0 |
| 470.lbm* | 68 | 38 | 0 |
| 482.sphinx3* | 123 | 80 | 0 |

**Table 2: ROP gadgets removal by MIP on x86-32 benchmarks.**

| SPECCPU2006 | MIP/SFI-32 | | MIP-64 | | SFI-64 | |
|---|---|---|---|---|---|---|
| | `btm` | `btr` | `btm` | `btr` | `btm` | `btr` |
| 400.perlbench | 14.2 | 18.9 | 13.7 | 16.5 | 15.5 | 18.4 |
| 401.bzip2 | 13.3 | 15.4 | 13.6 | 15.1 | 19.6 | 21.1 |
| 403.gcc | 13.0 | 16.9 | 12.0 | 14.5 | 13.1 | 15.6 |
| 429.mcf | 15.4 | 20.9 | 15.4 | 19.7 | 21.7 | 26.0 |
| 445.gobmk | 3.9 | 5.8 | 3.7 | 4.9 | 4.0 | 5.1 |
| 456.hmmer | 13.8 | 17.8 | 13.7 | 16.3 | 16.2 | 18.7 |
| 458.sjeng | 11.9 | 14.7 | 12.1 | 14.3 | 14.0 | 16.2 |
| 462.libquantum | 15.3 | 20.7 | 15.9 | 20.1 | 17.2 | 21.1 |
| 464.h264ref | 12.8 | 14.2 | 12.9 | 13.8 | 16.5 | 17.5 |
| 433.milc* | 13.6 | 17.0 | 13.7 | 16.2 | 15.5 | 18.0 |
| 470.lbm* | 12.8 | 15.8 | 13.3 | 14.6 | 15.9 | 18.0 |
| 482.sphinx3* | 13.1 | 17.1 | 13.0 | 16.4 | 14.8 | 17.4 |
| GM | 12.7 | 16.2 | 12.7 | 15.1 | 15.1 | 17.7 |

**Table 3: Static binary-size increase by MIP and MIP-based SFI in percentage (%).**

mentation code is run and the same bits are retrieved. We tested all three kinds of chunks and confirmed that the results were similar. Therefore, in the following sections, we report overhead numbers only for the case of instruction chunks.

## 7.2 Space overhead

In general, MIP's space overhead is caused by the bitmap and also the instrumentation code inserted before indirect branches and unsafe instructions. For each benchmark program, we measured its space overhead in terms of both the static binary-size increase and the impact on dynamic memory footprint.

Table 3 presents the static binary-size increase for MIP and MIP-based SFI. It presents the cases for both sequences of bit tests discussed in Sec. 5.1: `btm` and `btr`. On x86-32, since MIP-based SFI uses hardware segmentation to constrain memory access, the SFI instrumentation is the same as the MIP instrumentation. Therefore, we use a single column in the table for MIP and MIP-based SFI-32. The MIP-64 column shows the overhead of MIP on x86-64, and the SFI-64 column shows the overhead of MIP-based SFI on x86-64. The GM row shows the geometric mean of columns.

The table shows that on x86-32 and x86-64 MIP incurs an average of around 13% and 16% binary-size increase, using `btm` and `btr`, respectively. Most binary-size increase is caused by the size of the bitmap (about 11%). The `btr` sequence adds an extra 2-5% increase than `btm`. Furthermore, the SFI-64 instrumentation increases the size for about 3% because of memory-write sandboxing. Thanks to variable-sized chunks, MIP's binary-size increase is much smaller than previous SFI systems. For instance, the binary-size increase of NaCl-x86-64 [17] is roughly 60%. We measured NaCl-x86-32's binary-size increase on SPECCPU2006 programs and its increase is also around 60%. PittSFIeld [14], another SFI system, has a binary-size increase of around 75% on SPECCPU2000 programs. Compared to CCFIR [25] and binCFI [26], MIP's binary-size increase is less: CCFIR incurs around 33% due to the inserted code and the springboard section; binCFI imposes around 139% due to the instrumentation, its address translation map, and a copy of the original code.

We also measured the impact of MIP on the memory footprint. The memory-footprint increase for the benchmark programs was negligible. The majority of physical memory used by a running program is occupied by its runtime data. MIP adds a bitmap, whose physical-memory size is one eighth of the code, and causes slight increase to the code region. As a result, it has a negligible impact on the memory footprint.

## 7.3 Execution-time overhead

Table 4 lists the execution-time overhead for MIP and MIP-based SFI. The GM (INT) row computes the geometric mean of all integer benchmarks, and the GM (ALL) row computes the geometric mean of integer and floating-point benchmarks.

MIP's performance overhead is around 5% for the `btr` sequence. It is lower than a typical fine-grained CFI implementation: the classic CFI [1] reports 15% and an improved implementation reports around 6–8% [23]. CCFIR [25] also enforces coarse-grained CFI. Its performance overhead on SPECCPU2006 programs is around 4.2%; however, CCFIR optimizes its treatment of return instructions (using a stack-based "ret" rather than a register based jump) in a way that improves performance but is unsafe under MIP's attacker model, because return addresses suffer from a time-of-check-to-time-of-use race condition. As another coarse-grained CFI instrumentation technique, binCFI [26] imposes around 4.3% overhead on SPECCPU2000 C benchmarks, comparable to MIP.

On top of MIP, SFI-64 data sandboxing adds only 1–2% more execution-time overhead. The reason is twofold: first, the memory-write sandboxing instruction `lea` is lightweight; second, many memory writes do not need to be instrumented because the targets are computed by adding a constant to `rip` or `rsp` (discussed in Section 6). In comparison, NaCl-x86-64 has an overhead of 14.7% on SPECCPU2000 programs. NaCl-x86-32 has an average overhead of 5%. Therefore, our SFI implementation achieves competitive performance, and a much less space overhead than previous SFI systems.

In general, the `btr` sequence incurs less execution-time overhead than `btm`. We did further micro-benchmark analysis, which showed that the `btr` sequence stalls the instruction decoder less, and therefore enables more instruction-level parallelism. When combining the results of space and execution-time increase, we can see that `btr` is more time efficient, at the expense of extra binary-size increase.

Some benchmarks in the table have much higher performance overheads than others. Further experiments showed that the performance overhead correlates with the frequency of indirect branches. For example, `perlbench`, a Perl script interpreter, has high overhead because its implementation first translates Perl scripts into its internally defined instructions and then interprets an instruction by executing its corresponding function through an indirect call. Therefore indirect calls and returns are executed frequently. In contrast, `bzip2`'s computation mostly executes arithmetic instructions for compression; its overhead is low. Some benchmarks report slight performance improvement, possibly because of alignment benefits after instrumentation.

We note that the performance numbers are for the case of statically linking benchmark programs with the MUSL library, which was rewritten with the same instrumentation. Further, we used MIPDLinker to dynamically load and link the benchmark programs and the MUSL library, and we found that the execution-time overhead was almost the same as the case of static linking.

We also measured the MIPVerify's SFI verification time of all SPEC programs. On average, MIPVerify can verify 10-megabyte code per second. The instruction decoding engine of LLVM-MC consumes about 60% of the verification time.

## 8. CONCLUSIONS AND FUTURE WORK

This paper proposes MIP, an efficient instrumentation technique for monitor code integrity in low-level IRMs. To achieve this, MIP partitions the code into variable-sized chunks, maintains a bitmap for recording chunk beginnings, and instruments indirect branches so that they target chunk beginnings. Compared to other forms of CFI, MIP's coarse-grained CFI is sufficient for monitor code integrity, causes low space overhead, and requires only small changes to a compiler toolchain to support separate compilation. We believe MIP provides an IRM designer a sweet spot that combines efficiency, security, and convenience of reusing instrumented libraries.

As future work, we plan to extend MIP to support dynamically generated code, common in modern language interpreters such as JavaScript interpreters. When code is dynamically generated, the bitmap needs to be extended, similar to the case of DLLs.

| SPECCPU2006 | MIP/SFI-32 | | MIP-64 | | SFI-64 | |
|---|---|---|---|---|---|---|
| | `btm` | `btr` | `btm` | `btr` | `btm` | `btr` |
| 400.perlbench | 27.4 | 17.4 | 26.3 | 15.0 | 31.3 | 14.9 |
| 401.bzip2 | -0.1 | -0.1 | 1.2 | 1.5 | 4.6 | 4.6 |
| 403.gcc | 6.1 | 4.8 | 11.7 | 7.5 | 14.5 | 8.9 |
| 429.mcf | 1.0 | 0.5 | -2.2 | -1.0 | 0.3 | -1.4 |
| 445.gobmk | 11.8 | 11.4 | 13.3 | 11.1 | 14.2 | 13.2 |
| 456.hmmer | 0.0 | -0.3 | 0.2 | 0.0 | 4.6 | 4.0 |
| 458.sjeng | 11.1 | 9.7 | 12.3 | 7.8 | 7.8 | 7.6 |
| 462.libquantum | 1.1 | -0.2 | -0.1 | -2.3 | -0.7 | -1.4 |
| 464.h264ref | 11.7 | 7.3 | 25.5 | 16.0 | 24.6 | 14.7 |
| 433.milc* | 2.5 | 1.4 | 1.5 | 0.4 | 1.7 | 0.4 |
| 470.lbm* | 0.6 | 0.6 | 0.0 | 0.2 | 1.0 | 1.0 |
| 482.sphinx3* | 2.3 | 0.4 | 1.9 | 0.2 | 2.4 | -0.2 |
| GM (ALL) | 6.0 | 4.2 | 7.2 | 4.5 | 8.9 | 5.3 |
| GM (INT) | 7.4 | 5.4 | 9.3 | 6.0 | 11.3 | 7.1 |

**Table 4: Execution-time overhead imposed by MIP/SFI in percentage (%).**

## Acknowledgments

We thank Stephen McCamant and anonymous reviewers for their valuable comments. This research is supported by US NSF grants CCF-0915157, CCF-1149211, CCF-1217710, and two research awards from Google.

## 9. REFERENCES

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

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

[3] Akritidis, P., Costa, M., Castro, M., and Hand, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th Usenix Security Symposium* (2009), pp. 51–66.

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

[5] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 147–160.

[6] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. Fast byte-granularity software fault isolation. In

*ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2009), pp. 45–58.

[7] DAVI, L., DMITRIENKO, R., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., NURNBERGER, S., AND SADEGHI, A.-R. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium(NDSS)* (2012).

[8] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 75–88.

[9] ERLINGSSON, Ú., AND SCHNEIDER, F. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)* (1999), ACM Press, pp. 87–95.

[10] ERLINGSSON, Ú., AND SCHNEIDER, F. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy (S&P)* (2000), pp. 246–255.

[11] EVANS, D., AND TWYMAN, A. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy (S&P)* (1999), pp. 32–45.

[12] JIM, KHALYAVIN, A., MCCUTCHAN, J., AND CHEN, B. Some thoughts on NaCl binary size. Sent to the Native Client mailing list. URL: `https://groups.google.com/forum/#!topic/native-client-discuss/M9Jv5uCS3BA`, Aug. 2012.

[13] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure execution via program shepherding. In *11th Usenix Security Symposium* (2002), pp. 191–206.

[14] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *15th Usenix Security Symposium* (2006).

[15] SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. Disassembly of executable code revisited. In *In Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE* (2002), IEEE Computer Society, pp. 45–54.

[16] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference* (2002), ACSAC '02, pp. 209–218.

[17] SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPF, K., YEE, B., AND CHEN, B. Adapting software fault isolation to contemporary CPU architectures. In *19th Usenix Security Symposium* (2010), pp. 1–12.

[18] SHACHAM, H. 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)* (2007), pp. 552–561.

[19] SMALL, C., AND SELTZER, M. A tool for constructing safe extensible C++ systems. In *Proceedings of the Third Usenix Conference on Object-Oriented Technologies* (1997), pp. 13–23.

[20] WAHBE, R., LUCCO, S., ANDERSON, T., AND GRAHAM, S. Efficient software-based fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (New York, 1993), ACM Press, pp. 203–216.

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

[22] YEE, B., SEHR, D., DARDYK, G., CHEN, B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (S&P)* (May 2009).

[23] ZENG, B., TAN, G., AND ERLINGSSON, Ú. Strato: A retargetable framework for low-level inlined-reference monitors. In *22nd Usenix Security Symposium* (2013).

[24] ZENG, B., TAN, G., AND MORRISETT, G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *18th ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 29–40.

[25] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. *2013 IEEE Symposium on Security and Privacy 0* (2013), 559–573.

[26] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *22nd Usenix Security Symposium* (2013).