Architectural Support for Software-Based Protection

Mihai Budiu
Microsoft Research

Úlfar Erlingsson
Microsoft Research

Martín Abadi
Microsoft Research and UC Santa Cruz

ABSTRACT

Control-Flow Integrity (CFI) is a property that guarantees program control flow cannot be subverted by a malicious adversary, even if the adversary has complete control of data memory. We have shown in prior work how CFI can be enforced by using inlined software guards that perform safety checks. The first part of this paper shows how modest Instruction Set Architecture (ISA) support can replace such guard code with single instructions.

On the foundation of CFI we have implemented XFI: a protection system that offers fine-grained memory access control and fundamental integrity guarantees for critical system state. XFI can be seen as a flexible, generalized form of software-based fault isolation (SFI). In the second part of this paper we present ISA support for XFI, in the form of simple bounds-check instructions.

CFI and XFI can significantly increase the security and integrity of software execution. Our results indicate that support for CFI and XFI is a straightforward, simple addition to hardware architectures. Compared to software guards, such hardware support increases the efficiency and simplicity of enforcement.

Categories and Subject Descriptors

B.8.1 Hardware [**Performance And Reliability**]: Reliability, Testing, and Fault-Tolerance; D.3.4 [**Programming Languages**]: Processors; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Performance

Keywords

Binary Rewriting, Control-Flow Graph, Control-Flow Integrity, Hardware Support, Software Fault Isolation, Security, Memory Protection

1. INTRODUCTION

Preventing software attacks is one of the most important challenges of computer security. A significant class of attacks exploits software faults to inject malicious code in the compromised system memory, using for example a buffer overflow. Once the attack code is in memory, it may attempt to hijack the execution of the resident executable, by subverting its control flow. In other cases, the attack proceeds by corrupting data that determines future activity.

In prior work [1, 2, 7] we have devised a pair of software techniques (CFI and XFI) for hardening programs against powerful malicious adversaries. We present a brief overview of CFI and XFI in the next section. Part of our prior work consists in giving precise formal statements and proofs for the kinds of guarantees that our techniques can offer, as well as experimental evidence and measurements to support our claims.

One important goal of our work is to show that these protection mechanisms can be implemented in software on legacy systems, without requiring fundamental changes to hardware or operating systems. However, there are disadvantages to software solutions, which include runtime overhead and additional complexity. The goal of this paper is to explore the ways in which architectural support can reduce this overhead and allow for simpler enforcement.

1.1 Contributions

The current paper is a companion to the paper on XFI [7], presenting in detail the architectural support which is only briefly described there. The following are new research contributions in this paper:

- We have designed a microarchitecture that offers support for CFI, described in Section 4. Our semantics for CFI instructions allows more precise static control-flow graph encodings than were possible with our original software CFI implementation.

- We have implemented support for CFI instrumentation in a binary-rewriting tool based on the Squeeze++ binary-instrumentation tool [15], and applied it to binary programs from the SPEC2k benchmark suite [13].

- We have designed microarchitectural support for XFI, described in Section 5.

Source		Destination	
Opcode bytes	**Instructions**	**Opcode bytes**	**Instructions**
FF E1	jmp ecx ; computed jump	8B 44 24 04	mov eax, [esp+4] ; dst

is instrumented as:

Source		Destination	
B8 77 56 34 12	mov eax, 12345677h ; load ID−1	78 56 34 12	.data 12345678h ; ID
40	inc eax ; compute ID	8B 44 24 04	mov eax, [esp+4] ; dst
39 41 04	cmp [ecx], eax ; comp ID & dst		
75 13	jne error_label ; if != fail		
8D 49 04	lea ecx, [ecx+4] ; skip ID at dst		
FF E1	jmp ecx ; jump to dst		

Figure 1: Example CFI software instrumentation of the x86 instruction `jmp ecx`. At the left is the branch (source), while at the right is the destination of the branch.

- We have implemented the proposed microarchitectural changes for CFI and XFI in Sim-alpha, a cycle-accurate simulation framework [6].

- We have hand-instrumented Mediabench kernels [10] for XFI support, to evaluate the static overheads and the complexity of the compiler analysis required to perform the instrumentation.

- We have performed detailed dynamic measurements of SPEC and Mediabench programs. We have quantified the overhead and dynamic behavior of the programs instrumented with the proposed ISA support for CFI and XFI.

We show that architecture support for software-based protection can substantially increase the efficiency and simplicity of the CFI and XFI implementations. At the same time, our proposed microarchitectural changes are truly minimal and require neither cryptographic capabilities nor increase of critical resources, such as stretching the clock cycle or increasing register file bandwidth.

2. BACKGROUND: CFI AND XFI
In this section we summarize our main results on CFI and XFI, which form the foundation of the current work.

2.1 CFI
Control-Flow Integrity (CFI) was introduced in a couple of papers. The first one [1] studies CFI enforcement, presents an implementation for Windows on the x86 architecture, gives results from experiments, and suggests applications. The second paper [2] presents the basic theory that underlies CFI enforcement, and includes statements and proofs of theorems about the executions of programs instrumented with CFI.

Control-flow integrity means that the execution of a program dynamically follows only certain paths, in accordance with a static policy that comprises a Control-Flow Graph (CFG) of the machine code of the program. CFI can prevent attacks which, by exploiting buffer overflows and other vulnerabilities, attempt to control program behavior. Statically (i.e., before the program is run) the complete set of legal branch targets is established (i.e., the set of legal CFG edges is fixed, including those for indirect branches such as computed jumps). The set of CFG edges is encoded within the program executable. Code is inserted to ensure that the program follows at run-time the prescribed CFG in all cases, even when the data memory of the program is arbitrarily corrupted. (CFI assumes that the code segment is immutable, so direct branches need not be guarded at runtime: a static analysis can ascertain that their target lies within the code segment.) Contrast the rigidity of the control-flow enforced by CFI with the flexibility of control-flow in traditional machine code: in the latter an indirect branch (e.g., a return instruction after a buffer overflow on the stack) can target *any* byte of the program, including code, data, stack, even jumping in the middle of legal instructions on the x86 architecture.

Conceptually, CFI enforcement is achieved by subjecting executables to the following two transformations: (a) inserting identifying binary "labels" at each branch destination, and (b) preceding each branch with an inline code fragment which checks that the branch destination contains the correct expected label. These code transformations can be performed either by a binary rewriter or by a compiler. In practice CFI enforcement is easily applied to executables which are derived from high-level language programs: for such executables it is quite simple statically to compute a CFG to be followed during execution.

Figure 1 shows how an x86 indirect branch `jmp ecx` is instrumented using the software version of CFI. The instrumentation inserts a four-byte label `12345678h` at the destination. At run-time, prior to the execution of the branch, a software CFI guard compares the value in memory at address `ecx` to these label bytes. If the value is equal to `12345678h`, the branch is executed; on inequality, an error handler is invoked. (There are other possibilities for implementing CFI guards in software [1].)

Figure 1 highlights a number of important features of the software implementation of CFI guards:

- The labels must not have byte encodings that are part of other program instructions.

- The CFI guard code cannot contain the label bytes,

unless the guard code is considered a valid destination in the CFG. (This is why the guard encodes a value that is incremented to obtain the actual label.)

- The guard code may overwrite other registers and flags (eax in this example). If these registers are live, they have to be properly saved and restored by the guard code.

- The execution of cmp [ecx], eax fetches data in the D-cache from the code segment, causing additional memory traffic and cache pressure.

- The guard code contains two branches, including an additional conditional branch. The conditional branch is highly predictable, but it still pollutes the branch predictor structures.

Similar issues arise in the software implementation of XFI guards. Our architectural support for CFI and XFI addresses all these concerns, thereby enabling simpler, more efficient enforcement.

We should note that, even when CFI enforces a conservative, coarse-grained approximation of the CFG (i.e., allows a branch to target more destinations than strictly necessary), it still provides substantial security benefits. These benefits result from the CFI enforcement of important, simple runtime invariants, such as: "functions are entered only at the beginning", and "returns transfer control only to a point after a valid call site". In practice these properties alone are enough to thwart a wide class of malicious attacks, including code injection and jump-to-libc attacks [1, 12].

The extra checks and labels inserted by CFI do incur some storage and run-time overheads, which we have quantified for x86 [1]. We have found the dynamic run-time overheads to average 16%, with a maximum of 45%. In Section 4 we devise architectural solutions for tackling these overheads.

2.2 XFI

XFI is an efficient, comprehensive software protection system that supports fine-grained memory access control and fundamental integrity guarantees for system state [7]. XFI offers a flexible, generalized form of software-based fault isolation (SFI) [17] by building on CFI at the machine-code level.

XFI allows several software modules to execute safely side by side within a single (even fully privileged, e.g., ring 0) address space—without use of hardware support mechanisms such as page tables, segments, or instruction virtualization. A host system can grant XFI modules access to an arbitrary number of memory regions, at byte granularity. XFI also tightly controls the entry and exit points of a module (its interfaces). XFI uses a second stack to protect control-flow information, such as return addresses. XFI provides strong integrity guarantees for the second stack, and for other critical system state, such as the flags register. XFI also makes use of a number of software guards, including CFI guards, whose use is subject to static verification.

For the present purposes the most important guards used by XFI are memory-range guards, which bound memory

```
void f(char *p) {
    p[2] = 2;       /* movb [eax+2], 2 */
    p[-1] = 3;      /* movb [eax-1], 3 */
}
```

Figure 2: C source code (and x86 assembly code in comments) for a program with several memory accesses made through the same pointer.

accesses, checking their validity against the set of accessible regions. A bounds check must precede the execution of any memory-access instruction. For example, the guard for a four-byte write at the address pointed by EAX must ensure that all bytes in the range [EAX, EAX+4) are writable. Memory-range guards form the bulk of the XFI guards. The frequent use of memory-range guards introduces significant execution overheads. Therefore, this paper considers architectural support for memory-range guards, and assumes that other XFI guards remain implemented in software.

XFI may take advantage of control-flow integrity to relocate a memory-range guard to a program point that dominates the actual instruction subject to a bounds check. CFI allows XFI to hoist guards even out of loops: because control flow cannot be subverted, the guards are always executed. As a consequence of hoisting guards, XFI can merge several checks of memory accesses made through the same pointer into a single bounds check. Consider the code shown in Figure 2. Instead of performing two separate write checks for EAX-1 and EAX+2, XFI can merge them in a single write check for the interval [EAX-1,EAX+2], preceding both writes. (The assumption is that all intermediate bytes are also writable.) Therefore, XFI memory-range guards check that a memory range is accessible around an address held in a register, within constant offsets L, below, and H, above.

In general a module may have access to a large number of disjoint, contiguous memory regions. One of these regions [A,B) may be more frequently accessed during execution and its bounds are of particular interest. (These could be, for example, the bounds of the heap of the current module.) An XFI memory-range guard should be fast when it succeeds within the range [A,B), and should fallback on a slower solution for checking other ranges. The range [A,B) is called the "fastpath" range.

We have shown [7] that XFI memory-range guards can be implemented using a fastpath check, as shown in Figure 3. However, even this fastpath software implementation requires several instructions and branches, and has significant overheads for memory-access-intensive benchmarks (50%–90%). In Section 5 we provide hardware support for XFI memory-range guards that determine whether [$r-L, $r+H) ⊆ [A,B).

3. EVALUATION INFRASTRUCTURE

We evaluate ISA extensions for CFI and XFI using cycle-accurate simulations by executing programs to completion. We present data for all programs which worked with both our compiler and simulation infrastructure. We compile only

Parameter	Value
Issue width	4 int + 2 FP
Reorder buffer	80
DTLB	fully associative, 128 entries
ITLB	fully associative, 128 entries
Integer physical register file	80
FP physical register file	72
Integer units	4
FP units	2
L1I cache	64KB virtually indexed, 2-way, 32-byte blocks, 1 cycle hit
L1D cache	64KB virtually indexed, 2-way, 32-byte blocks, 3 cycles hit
L2 cache	16MB unified, physically indexed, direct mapped, 12 cycles hit
Victim cache	8 entries
LSQ	32 load entries, 32 store entries
Local branch prediction	1024 first level, 1024 3-bit saturating counters
Global branch predictor	4096 2-bit saturating counters
Tournament branch predictor	4096 2-bit saturating counters
Memory	SDRAM 131 MHz
Clock cycle	500 MHz
Fetch/map/issue width	4
Commit width	11

Table 1: Essential parameters of the simulated architecture.

```
if ($r < A + L) then goto SlowpathGuard
if (B - H < $r) then goto SlowpathGuard
/* perform access */ ...
```

Figure 3: The fastpath code for a software XFI memory-range guard. This code validates that the range [$r-L, $r+H) is contained within the "fastpath" range [A,B).

C programs, omitting C++ and Fortran. We do not omit programs because they perform poorly, but because they fail to work with our tool-chain.

CFI and XFI were designed to provide protection for the x86 variable-width ISA. However, because we do not have access to a cycle-accurate simulation framework for x86 processors, we resort to using an Alpha simulator, Sim-alpha [6]. Sim-alpha has been validated to be within 7% accurate on the SpecInt2K benchmarks, when compared with a real hardware machine, a Compaq DS-10L server. While the Alpha ISA is quite different from x86, we believe that: (a) the qualitative results we present can be extrapolated to any type of architecture, because we quantify relative overheads; and (b) the ISA extension proposals we describe are simple, and should be easily adaptable to other microarchitecture implementations.

The simulator parameters used for this study are given in Table 1.

4. HARDWARE SUPPORT FOR CFI

CFI works by constraining the possible destinations of indirect branches (including procedure returns) to a set of legal targets within the program CFG. Each target of an indirect branch is statically annotated with a *label*, embedded within
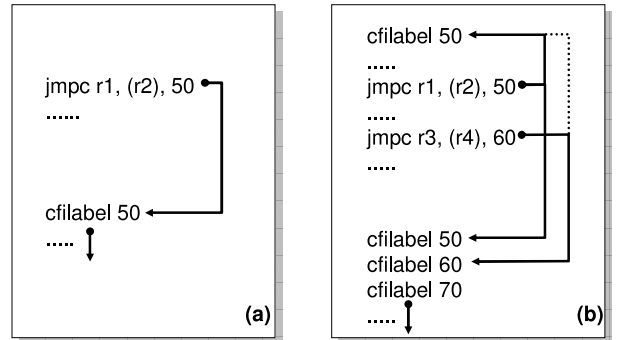

Figure 4: (a) Use of the CFI instructions. (In the Alpha ISA the second register argument of an indirect branch is used to save the Program Counter value.) (b) Implementing join points in the CFG. The dotted line is an extra CFG edge which is introduced in the prior software version of CFI by the constraint on CFG equivalence classes. The hardware version of CFI does not require this CFG edge to be present.

the binary code. For each indirect branch a check will be performed to establish whether the destination code contains the expected label; if no correct label occurs, the processor triggers an exception. In this way, indirect branches cannot target but the places that contain the correct embedded labels.

4.1 ISA Support for CFI

The ISA support for CFI is quite straightforward, as shown in Figure 4(a). It consists of 4 instructions, as shown in Table 2. There is a cfilabel instruction, and three "checked jump" instructions.

Operation	Semantics
`cfilabel L`	`if (cfi_register == L)` `    then cfi_register := 0`
`jmpc ra,(rb), L`	`cfi_register := L; jmp ra,(rb)`
`retc ra,(rb), L`	`cfi_register := L; ret ra,(rb)`
`jsrc ra,(rb), L`	`cfi_register := L; jsr ra,(rb)`

Table 2: ISA support for CFI.

- The new `cfilabel` instruction is used to embed a label bit pattern within the code segment, at the destination of a branch. (In the software method of Figure 1 the label was encoded using a `.data` directive.)

- Each checked jump instruction also embeds a label bit pattern and has the effect of a CFI guard, and thus eliminates the five instructions preceding the `jmp` in the left-bottom of Figure 1. After a checked jump, until a `cfilabel` with a matching pattern is committed, retiring any instruction other than a `cfilabel` will trigger an exception. These errors are dispatched using the processor vectored exception mechanism. We believe this heavyweight handling of errors is appropriate, since a failure of control flow is a rare and significant event that indicates important corruption.

To implement these instructions, we have added a new integer register to the microarchitecture, `cfi_register`. This register can be used only implicitly by the new instructions, as described below. `cfi_register` is renamed like any other integer register, allowing multiple CFI instructions to be in-flight at the same time.

- The `cfilabel` instruction is intended to be the destination of the checked branch instructions. It contains a 16-bit immediate label value. (Only 16 bits of encoding space were available in the branch instruction for the Alpha ISA; for an ISA such as x86 one can use longer labels.) The effect of the instruction is to compare the immediate with the contents of the `cfi_register`, and to reset the `cfi_register` if the two are equal.

  This particular choice of a semantics for the `cfilabel` allows multiple `cfilabel` instructions with different immediates to be used in sequence to implement join points in the CFG, as shown in Figure 4(b). Such a join point can be the target of multiple branches with different labels.

  The join points allow embedding precise static CFGs within the program. More concretely, the software inlined guards in [1] require that the program labels are partitioned into equivalence classes. The partitioning may force branches with different but overlapping sets of destinations to have identical labels, resulting in a coarser approximation of the CFG. Our ISA implementation removes this restriction.

  The example in Figure 4(b) shows two branches, with labels 50 and 60, that can target a common basic block in the CFG. In our prior work on CFI software guards, the dotted edge from Figure 4 would have to be present in the CFG (since labels 50 and 60 are in the same equivalence class).
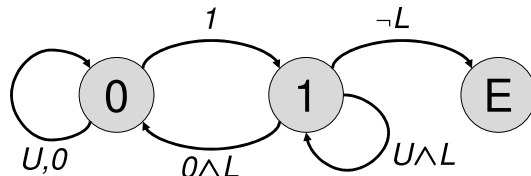

Figure 5: FSM used by the commit stage to retire instructions. The state labels indicates the current value of the predicate `cfi_register` $\neq$ 0. When the FSM enters the state labeled E a CFI exception is triggered. The arc events are labeled from instruction that retire: *0* if the instruction has set `cfi_register` to 0 in the EX stage, *1* if the instruction has set `cfi_register` to a non-zero value, and *U* if the instruction has left the register unchanged. An *L* on an arc indicates that a `cfilabel` instruction is retiring.

The key of the hardware solution leveraged to achieve join points is modifying the semantics of *all* the instructions, to check for CFI exceptions in the commit stage, as described below. It seems hard to perform the same feat inexpensively in software.

- The semantics of all three new checked branch instructions is essentially the same. They differ only with respect to branch prediction behavior. The three branch instructions correspond very closely to the Alpha ISA instructions JMP (indirect branch), RET (return from subroutine), and JSR (jump to subroutine), respectively. The instructions use the low 16 bits of the opcode to encode the CFI label. (The Alpha ISA uses these bits for branch destination hints.)

  As shown in Table 2, the first step of the execution of a checked branch sets `cfi_register` to the value of the immediate label; next the execution continues like the corresponding unchecked branch.

- In the commit stage of the processor, all Alpha instructions—except `cfilabel`—check and trigger pending CFI exceptions. An instruction will trigger a CFI exception if it commits while `cfi_register` is not zero. The only instruction that can legally follow a checked branch is a `cfilabel`: thus the processor needs to perform the exception check at all other instructions. (This implementation clearly preserves precise exception handling.)

  It is important to notice that the commit-stage check can be implemented without accessing the register file (we don't need to add extra ports or bypass paths), by just monitoring dynamically the operations which change the value of `cfi_register` (the ones in Table 2). Each instruction is tagged in the execution stage with the value of the predicate (`cfi_register` $\neq$ 0) after its execution; the value is one of $\{0, 1, U\}$. (*U* is from "unchanged".)

  The simple three-state FSM shown in Figure 5 is executed in the commit stage. It processes the stream of the in-order committed instructions, and uses the tag of each instruction to trigger the CFI exceptions.

4.2 Evaluation Through Software Instrumentation

To instrument executables with CFI instructions we have modified the Squeeze++ binary-instrumentation program from Ghent University [15]. We compile programs using the standard Tru64 Alpha tool-chain, using the Compaq C Compiler V6.5-303, with the `-fast -arch ev67` options (optimizing for speed). We use Squeeze++ to insert `cfilabel` instructions in front of all indirectly accessed basic blocks (which are discovered using relocation information). We also rewrite all JMP, JSR, and RET instructions to their corresponding checked variants. Since we are just interested in evaluating the overhead of CFI, we have used a very coarse-grain CFG policy, by setting the label immediates to 1 everywhere, causing all CFI checks to succeed[1]. A bug in Squeeze++ prevents it from correctly handling some common code idioms generated by the version of the CC compiler we had available, and thus we are unable to report results for many SPEC2k programs.

The process through which Squeeze++ does binary rewriting is quite intrusive, since it involves disassembling the binary, normalizing the instructions, optimizing the code, and then reassembling. In order to eliminate the effects of Squeeze++, we also evaluate the baseline numbers after processing by Squeeze++. We report numbers for several types of binaries:

CFI With control flow restricted to the CFG.

CFI nop With each CFI instruction replaced with a nop.

protected stack With CFI only for indirect calls and indirect branches, but not for returns. The latter are assumed to be handled by a protected control stack (see [1, 7] for details).

protected stack nop Like protected stack, but with nops instead of CFI instructions.

Figure 6 shows the size overhead introduced by CFI. The average executable size increase is 3.25%, with a maximum of 4.08%. Increased size leads us to expect increased I-cache pressure. The processed binaries are run on Sim-alpha to completion on the "test" input set. The shortest benchmark runs for about 600 million clock cycles, the longest for about 22 billion clock cycles. No CFI exceptions are ever encountered.

The overwhelming majority of CFI instructions executed are `retc` and `cfilabel`, almost equally. There are two orders of magnitude fewer `jsrc` and even fewer `jmpc`. The run-time overhead results are presented in Figure 7. The baseline is the binary processed with Squeeze++ with no instrumentation inserted.

The run-time overhead of CFI instrumentation is on average 3.75%, with a maximum of 7.12%. Clearly, the overhead is

---

[1]Note that the checks would fail, triggering CFI exceptions, if an indirect branch would attempt to target the middle of a basic block, because of a software bug or a malicious attack.
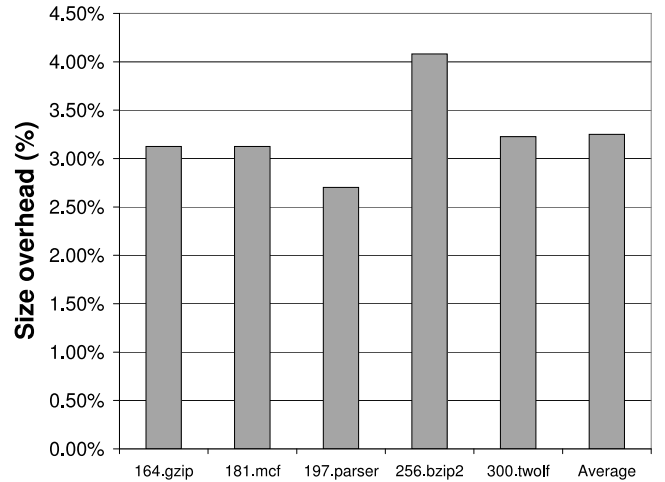

Figure 6: Size overhead of the CFI instrumentation. The baseline is the binary processed with Squeeze++ with no instrumentation.

never due to execution resources, since the NOP bar is practically identical to the previous bar. We have attributed the additional overhead to two main costs: additional L2 I-cache misses (at most 3% extra overhead), and load replay traps (another at most 3%). The traps are due to failed speculative executions of memory operations, and are not many, but are extremely costly. The traps occur only when the code misses in the cache, because some speculative operations execute in the wrong order, and this causes them to reside in the pipeline much longer. The extra traps are thus a consequence of the additional L2 I-cache misses. These traps could in principle be avoided by improved scheduling.

5. HARDWARE SUPPORT FOR XFI

XFI offers comprehensive software-based protection that includes a generalized form of software-based fault isolation. XFI allows several software modules to execute in the same, privileged address space, through the enforcement of memory access constraints and restrictions on both hardware and software interfaces. In particular, XFI requires all computed memory accesses to be checked at runtime by a memory-range guard.

5.1 ISA Support for XFI

We add support for XFI memory-range guards to the hardware architecture, since these guards cause the bulk of XFI runtime overhead, and contribute to the complexity of XFI enforcement. This support consists of three new ISA instructions, as shown in Table 3. In this section we evaluate the effectiveness of these instructions, with CFI enforcement disabled.

Each `mrguard` instruction encodes explicitly two 10-bit immediate values, `L` and `H`. (An instruction set like x86 could devote more bits to the encoding of these bounds.) The instruction implicitly refers to two of six fixed system registers, $\$low_{OP}$ and $\$high_{OP}$ (a pair for each $OP \in \{R, W, X\}$). These registers, respectively, hold the address bounds for the fastpath range $[A_{OP}, B_{OP})$ described in Section 2.2 and in [7]. (It is worthwhile to note that, by using registers,
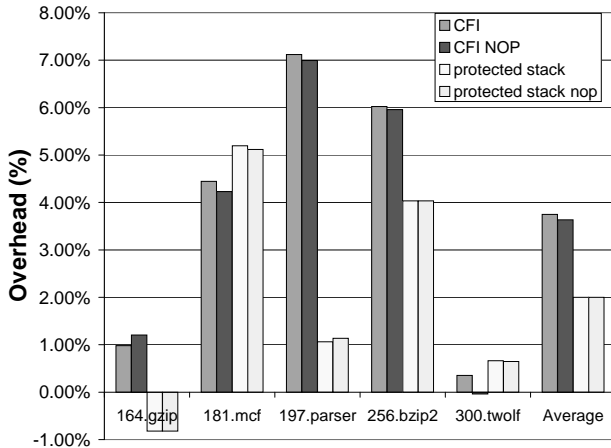
Figure 7: Dynamic overhead (slowdown) caused by the CFI instrumentation. The baseline is the same binary processed through Squeeze++, but with no instrumentation of any kind.

Operation	Semantics

mrguardOP $r, L, H	if ($r < low_{OP}$ + L)
	    then XFI_exception(OP)
	if ($high$_{OP}$ - H < $r)
	    then XFI_exception(OP)

Table 3: ISA support for XFI. OP is one of R (Read), W (Write), X (Execute). low_{OP} and $high_{OP}$ are two system registers that are preloaded with the bounds of the current fastpath region [A_{OP},B_{OP}). The mrguardOP operation checks whether [$r-L,$r+H) is contained within [A_{OP},B_{OP}).

the mrguard instructions are more flexible than our software guards, which encode [A_{OP},B_{OP}) using immediate constants.)

In our current implementation we assume that these registers are loaded by Alpha PAL code. Since PAL code execution is expensive, if domain crossings between modules are frequent enough, it is easy to devise an alternative architecture which relies on non-privileged instructions for altering the contents of these registers. In the benchmarks below we do not evaluate the cost of modifying these registers. The exception handler receives on the stack the address of the faulting instruction, so the OP argument is implicit in the opcode of the mrguardOP instruction passed to the handler. The XFI exception handler performs additional policy checks in software, and may finally allow or deny memory access, but it is never invoked in our benchmark runs.

The mrguard instruction performs hardware checks against only a single memory range [A_{OP}, B_{OP}); to permit other memory ranges, the XFI software handler must be invoked. In an alternative implementation, a hardware check could be performed against multiple memory ranges by using a parallel search of a small, associative lookaside structure. In this generalization, lookup failure would still invoke a software handler, but might happen less frequently. Software management of the associative structure could make it reflect the most common memory ranges. Such extended mrguard hardware support can be both simple and inexpensive, e.g., by with an implementation similar to that of protection-lookaside buffers [18]. The simpler register-based mrguard instructions that we have chosen to implement can be seen as a special case that uses a single-entry lookaside structure. In our simulation-based experiments we have evaluated this case only.

As we discuss in [7], the mrguard instructions are suitable only for bounding memory accesses with constant known compile-time offsets L and H from the base register. These are sufficient for the Alpha ISA, as long as we do not lift guard instructions out of loops, since Alpha has no complex memory addressing modes. The x86-like ISA supports complex memory addressing modes, and also the REP prefix on instructions, which causes them to be iterated. If those cases were to be handled in hardware, a different type of mrguard instruction would be required, where one of the two bounds L or H is no longer an immediate, but a (loop-index-dependent) register value.

5.2 Evaluation Through Software Instrumentation

Since we do not have a compiler for performing XFI instrumentation for Alpha binaries[2], we have resorted to hand-instrumentation. We have chosen to evaluate Mediabench applications, because (a) they are CPU intensive, (b) they have small code bases and thus can be instrumented manually, and (c) they have clearly defined kernels. The procedure we have followed is:

1. We use profiling tools to locate the kernel of each benchmark (the procedures where most of the time is spent). Because of inlining by the compiler, for some of these programs the kernel consists of multiple procedures. On average the programs spend 50% of the time in the kernels, with a maximum of 100%, and a mimumum of 10%.

2. We compile the kernel and generate assembly code.

3. We instrument the kernel by hand, inserting XFI guard instructions in assembly. Since these instructions do not exist in the unmodified compiler tool chain, they are assembled using .byte sequences, handled using preprocessor macros.

4. We compile and link the whole application.

5. We execute the application on the modified Sim-alpha simulator, extended with support for the new instructions. We run Sim-alpha to completion on the default input sets, measuring the overhead with respect to the non-instrumented application. Mediabench running times are much shorter than for SPEC. They range between 10 million and 680 million clock cycles.

---

[2]An instrumentation tool based on a binary rewriter for x86 binaries is described in [7], but we could not port it to the Alpha due to the absence of a flexible enough binary-rewriting tool for Alpha. Unfortunately the bugs in Squeeze++ prevented us from using it for this purpose.

6. We extrapolate the overhead to the whole application, based on the known kernel running time.

The kernels we have used are presented in Table 4. The kernels range in size from 73 lines of assembly (excluding comments and spaces) to almost 1900. The large size is due to loop unrolling performed by the compiler on the inner loops. On average we added one annotation for every 26 instructions. We have not attempted to lift guard instructions out of loops, but we have used a combined memory-range guard at the beginning of basic blocks when multiple references are made relative to the same base address. Since with XFI the stack pointer's usage is very disciplined, we check bounds for the current stack frame only in the procedure entry basic block; since none of our kernels dynamically extends its stack frame, this method is sound. We have used a "null" policy, setting low_{OP} to zero and $high_{OP}$ to OxFF...FF. The null policy ensures that no exceptions are triggered, but that overheads are accurately measured.

The results in this section are valid under the assumption that the density of `mrguard` instructions is relatively uniform for the kernel and the rest of the application, because we extrapolate the cost of memory-range guards for the kernel to parts of the application which are not instrumented.

Figure 8 displays the performance overhead of the XFI instructions extrapolated to the whole application. We present two measures for each applications: protecting all R/W accesses, and protecting only memory writes. When protecting R/W accesses, the maximum overhead is 14.8%, with an average of 7.1%. When protecting only the memory writes, the average overhead is 2.8%, with a maximum of 6.7%[3].

When replacing all `mrguard` instructions with just `nops`, the overhead is virtually identical. This fact shows that the bottleneck is the front-end of the processor, and not execution resources. The `mrguard` memory-range guard instruction introduces no additional dependences in the computation flow: all the extra arithmetic operations are off the critical path.

The processor executes on average 7% additional instructions, with a maximum of 17.3%. Of these, `mrguard` instructions guarding writes are 0.9%, and never more than 3.4%.

6. RELATED WORK

Prior proposals for using hardware protection for isolating software modules are substantially different from ours.

We do not review here classic approaches, such as capabilities [5], segmentation [9], and paging [8]. We focus instead on more recent proposals.

The closest work in goals is probably Mondrix [18], which

---

[3] jpeg_e has an an anomalous behaviour, since the program with R/W memory-range guards is faster than without (and also faster than just with write guards). The reason is a fortuitous alignment which causes a good interaction with the IL1 way predictor, making the program with R/W memory-range guards have 30% fewer prediction misses, and thus a better performance despite a higher number of instructions executed.
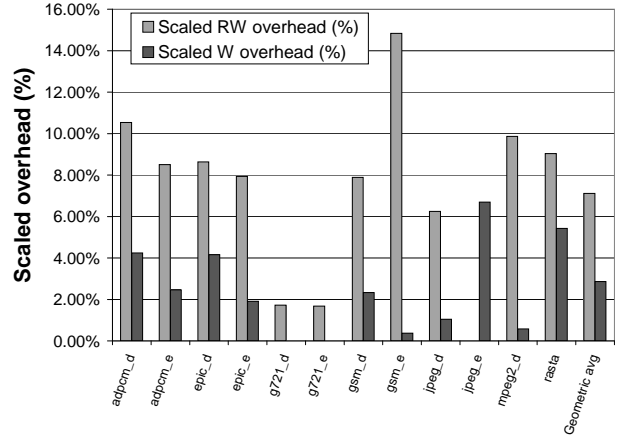

Figure 8: Overhead of the XFI memory-range guard instruction `mrguard`.

attempts to provide fine-grained memory protection. The key structure in Mondrix is the Protection Lookaside Buffer, which uses a contents-addressable memory to check k entries in parallel. (As mentioned in Section 5, we could also use such a hardware structure to provide a generalized form of our `mrguard` instruction.) However, Mondrix does not rely on a lightweight mechanism for enforcing control-flow integrity; instead it has to use Gate Lookaside Buffers for inter-module transfers of control. Control flow inside of a module is trusted; as a result (since an innocuous instruction may embed a dangerous, privileged instruction), Mondrix must include a protection supervisor at the highest privilege level.

In XOM [11] the hardware does not trust main memory, and memory is partitioned into isolated compartments. The key primitive is encryption and decryption of memory content at the level of cache lines.

An alternative proposal for the integrity of control-flow generalizes our CFI by enforcing a more complex policy: sequences of n consecutive CFG edges [20]. The result is increased security, but requires a trusted co-processor, which consumes substantial storage and computational resources. This solution may also generate false alarms, since the policy is learned on-line, and is not conservative.

Nooks is mostly a software architecture for isolating device drivers inside an operating system [16]. Nooks drivers are isolated within the same address space — in this respect Nooks is related to XFI. Nooks achieves isolation by switching page tables between protection domains. The non-tagged TLB of x86 requires the TLB to be flushed on page table changes, and makes domain crossings with hardware support quite expensive.

iWatcher is a way of generalizing watchpoints and relying on thread-level speculation hardware for storing checkpointing state [21]. iWatcher can be used for lightweight instrumentation for buffer overflows or other similar software weaknesses — in this respect it overlaps in goals with XFI. iWatcher re-

Benchmark	Kind	Kernel Functions	Time (%)	LOC (asm)	Annotations Read	Annotations Write	LOC/ annot.
adpcm_d	Sound decoding	adpcm_decoder	100	84	5	2	12
adpcm_e	Sound encoding	adpcm_coder	100	111	5	3	14
epic_d	Image decompression	collapse_pyr	48	1670	43	133	9
epic_e	Image compression	internal_filter	85	1879	63	23	22
g721_d	Voice decompression	fmult, quan	60	73	1	0	73
g721_e	Voice compression	fmult, quan	60	73	1	0	73
gsm_d	Speech encoding	Short_term_synthesis_filtering	71	153	6	3	17
gsm_e	Speech decoding	Calculation_of_the_LTP_parameters	43	805	48	5	15
jpeg_d	Image decompression	jpeg_idct_islow	49	386	18	6	16
jpeg_e	Image compression	encode_mcu_AC_refine	31	335	29	8	9
mpeg2_d	Movie decoding	idctcol, idctrow, Fast_IDCT	31	354	11	5	22
rasta	Speech recognition	FR4TR	10	1618	16	39	29

Table 4: Mediabench kernels used for XFI evaluation. The "time" column shows an estimate of the time spent in the kernel by the whole program. "LOC" measures the number of lines of code in assembly of the kernel. The "annotations" columns show how many `mrguardR` and `mrguardW` annotations were inserted. The last column is computed as LOC/(`mrguardR`+`mrguardW`).

quires substantial support from the operating system when its cache-resident data structures overflow.

Both Minos [4] and Secure Information Flow [14] attempt to prevent control-flow transfers based on tainted data; both propose microarchitectures that focus on taint propagation, and require tracking tainted data through memory. Their guarantees are different from those of CFI and XFI.

DFI [3] is a new security mechanism that is related to both CFI and XFI. All three use inline guards to enforce at runtime a conservative, high-level model of permitted behavior that is derived from static analysis. Whereas CFI focuses on control flow, DFI focuses on memory access, maintains auxiliary state, and uses that state to ensure that a program instruction reads only memory values written by appropriate, other program instructions. Our CFI hardware support similarly validates modifications of the instruction pointer, using the `cfi_register` for auxiliary state.

SAFE-OPS [19] attempts to provide embedded software security by having an executable contain a pattern of executable instructions which is the cryptographic signature of another sequence of executable instructions of the same executable. An FPGA trusted coprocessor monitors whether the binary has been corrupted.

7. CONCLUSIONS

In a software implementation the job of the `jmpc` or `mrguard` instructions is handled by 4 to 6 machine code instructions, including a conditional branch for dispatching to the error label. The ISA support alternative that we have presented has many benefits:

- Smaller executable size.

- Reduced pressure on instruction-fetch related structures (I-cache, trace-cache, branch predictor).

- Decreased register pressure, since no intermediate results need to be computed.

- No pollution of the condition flag registers (for architectures with implicit side-effects, such as x86).

- No pollution of the data-cache for fetching of the label from the code segment for CFI.

This research advances the thesis that a modest amount microarchitectural support can greatly enhance the simplicity and efficiency of inlined enforcement of security policies. The execution overhead for enforcing CFI is below 7%, while the memory-range guard overhead (assuming the fast-path checks succeed), is below 14% for read-write protection and below 7% for write-only protection. While comparing numbers obtained on x86 and Alpha is difficult, these experiments suggest that support from `mrguard` instructions can reduce overhead by a factor of 5 on average. Building XFI support does not require stretching any of the critical stages of the microprocessor. Taking into account the great benefits for security and the minimal amount of hardware real-estate and performance impact, we posit that XFI architectural support is attractive.

Acknowledgments

8. REFERENCES
[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, pages 340–353, Alexandria, VA, November 7-11 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control-flow. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 111–124, Manchester, UK, November 1-4 2005.

[3] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In

Symposium on Operating System Design and Implementation (OSDI), Seattle, WA, November 6-8 2006.

[4] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966.

[6] R. Desikan, D.C. Burger, S.W. Keckler, and Todd Austin. Sim-alpha: a validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, University of Texas at Austin, Department of Computer Sciences, 2003.

[7] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, November 6-8 2006.

[8] J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM (CACM)*, 4(10):435–436, Oct 1961.

[9] Anatol W. Holt. Program organization and record keeping for dynamic storage allocation. *Communications of the ACM (CACM)*, 4(10):422–431, 1961.

[10] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–335, 1997.

[11] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168–177, New York, NY, USA, 2000. ACM Press.

[12] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 02(4):20–27, 2004.

[13] Standard Performance Evaluation Corp. *SPEC CPU 2000 Benchmark Suite*, 2000. http://www.specbench.org/osg/cpu2000.

[14] G. Edward Suh, Jae W. Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, October 2004.

[15] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, September 2005.

[16] Michael Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–222, Bolton Landing, NY, USA, October 2003.

[17] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216. ACM Press, 1993.

[18] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 304–316, New York, NY, USA, 2002. ACM Press.

[19] Joseph Zambreno, Alok Choudhary, Rahul Simha, Bhagi Narahari, and Nasir Memon. Safe-ops: An approach to embedded software security. *Trans. on Embedded Computing Sys.*, 4(1):189–210, 2005.

[20] Tao Zhang, Xiaotong Zhuang, Wenke Lee, and Santosh Pande. Anomalous path detection with hardware support. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 43–54, San Francisco, CA, 2005.

[21] Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu, and Josep Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM Trans. Archit. Code Optim.*, 2(1):3–33, 2005.