XFI: Software Guards for System Address Spaces

Úlfar Erlingsson
Microsoft Research
Silicon Valley

Martín Abadi
Microsoft Research
Silicon Valley
& UC Santa Cruz

Michael Vrable
UC San Diego

Mihai Budiu
Microsoft Research
Silicon Valley

George C. Necula
UC Berkeley

Abstract

XFI is a comprehensive protection system that offers both flexible access control and fundamental integrity guarantees, at any privilege level and even for legacy code in commodity systems. For this purpose, XFI combines static analysis with inline software guards and a two-stack execution model. We have implemented XFI for Windows on the x86 architecture using binary rewriting and a simple, stand-alone verifier; the implementation's correctness depends on the verifier, but not on the rewriter. We have applied XFI to software such as device drivers and multimedia codecs. The resulting modules function safely within both kernel and user-mode address spaces, with only modest enforcement overheads.

1 Introduction

XFI is a comprehensive software protection system that supports fine-grained memory access control and fundamental integrity guarantees for system state. XFI offers a flexible, generalized form of software-based fault isolation (SFI) [25, 36, 41] by building on control-flow integrity (CFI) [1, 2] at the machine-code level. This CFI foundation enforces external and internal interfaces, enables efficient XFI mechanisms, and helps protect the integrity of critical state, such as the x86 control registers.

In comparison with other protection alternatives, XFI requires neither hardware support [39, 44] nor type-safe programming languages [5, 19, 24]. XFI does not restrict memory layout and is compatible with system aspects such as signals and multi-threading. Furthermore, XFI applies at any privilege level, and even to legacy code that is run natively in the most privileged ring of x86 systems; in this respect, we regard XFI as achieving an important practical goal.

XFI has a clear architecture, whose basic implementation can be relatively straightforward and trustworthy. XFI protection is established through a combination of static analysis with inline software *guards* that (much as in SFI) perform checks at runtime. The *XFI verifier* performs the static analysis as a linear inspection of the structure of machine-code binaries; it ensures that all execution paths contain sufficient guards before any possible protection violation. Verification is simple and, in principle, amenable to formal analysis and other means of assuring correctness. An *XFI module* is an executable binary that passes verification; such modules can be created by hand, by compile-time code generation, or by binary rewriting. However, software that hosts XFI modules need trust only the verifier, not the means of module creation. Thus, XFI modules can be seen as an example of proof-carrying code (PCC) [29], even though they do not include logical proofs.

XFI protection relies on several distinct runtime mechanisms, whose correct use is established by the XFI verifier. Guards ensure that control flows only as expected, even on computed transfers, and similarly that memory is accessed only as expected. Multiple memory accesses can be checked by a single memory-range guard, optimized for fast access to the most-frequently-used memory. XFI also employs two stacks. The regular execution stack provides a *scoped stack*, which holds data accessible only in the static scope of each function, including return addresses and most local variables. The scoped stack cannot be accessed via computed memory references, such as pointers; therefore, it serves as isolated storage for function-local *virtual registers*. A separate *allocation stack* holds other stack data which may be shared within an XFI module. Like heap memory, the allocation stack may be corrupted by buffer overflows and other pointer errors.

XFI protection can be of benefit to any *host system* that loads binary modules into its address space to make use of their functionality. Operating systems are example host systems, as are web browsers. Conversely, those modules may rely on their host system, by invoking its

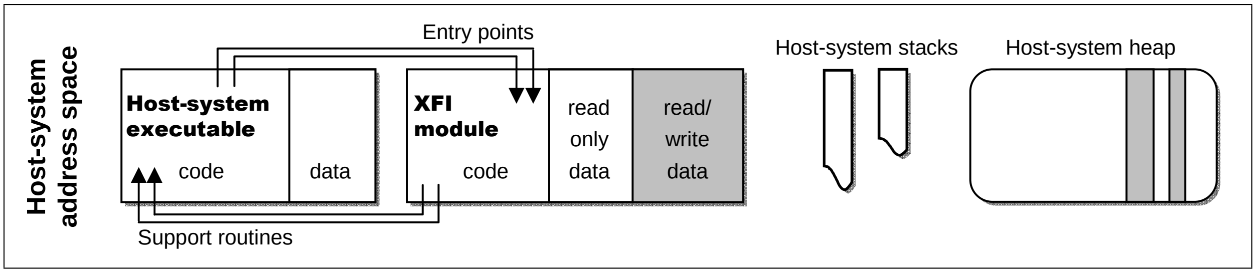

Figure 1: The address space of a host system with an XFI module. The module's external interfaces are restricted, as shown by the arrows. When the module uses a host-system stack, it is as a protected, scoped stack of virtual registers. Shaded areas are subject to arbitrary writes by the XFI module. Optionally, the module may read all memory.

support routines. For example, the XFI module in Figure 1 can use two support routines, and can be called through two *entry points* by its host system. The module can read its own code and constants (or, optionally, all of memory). It may read and write a section of its loaded module binary, and two contiguous heap regions to which the host system has granted it write access. It cannot modify its own code at runtime, so it cannot invalidate the properties established by the verifier statically.

XFI requires no complex infrastructure from host systems: XFI modules are normal executable binaries (e.g., DLLs), and can be loaded and used as such. As a result, host systems may sometimes be unaware of whether or not they are using an XFI module. In general, however, XFI protection does rely on support components, for instance for thread-specific management and for granting and revoking memory-access permissions.

We have implemented XFI for Windows on the x86 architecture as two main components: first, a relatively complex *XFI rewriter*, based on Vulcan [37], that instruments and structures executable binaries, and, second, a smaller, self-contained verifier. We have also designed and built host-system support components. Further, we have studied optional, lightweight hardware support for XFI, through cycle-accurate simulations.

We have used our implementation for creating independently verifiable dynamic libraries, device drivers, and multimedia codecs. The resulting XFI modules function safely within the address spaces of the Windows kernel and the Internet Explorer web browser, and provide services to those host systems. Thus, while still only a prototype, our implementation is already practical.

Although we have not explored many optimizations, XFI enforcement overhead is modest: in our experiments it ranged from 5% to a factor of two, on current x86 hardware. This overhead is acceptable for many systems; sometimes it can be lower than that of traditional hardware-supported address spaces (because context switching across XFI modules is fast [27, 41]).

Sections 2 and 3 describe XFI policies and mechanisms, respectively. Section 4 presents our implementation of these mechanisms. Section 5 discusses the application of XFI protection to device drivers and multimedia codecs. Section 6 gives measurements and other results of our experiments. Section 7 puts XFI in the context of related work, and Section 8 concludes.

2 Policies

XFI guarantees that certain properties hold during the execution of XFI modules. Some of these properties are fundamental ground rules about the interaction between XFI modules and their system environment. Others are convenient, auxiliary rules for this interaction. Yet others deal with the internal behavior of XFI modules, and contribute to security.

In discussing the implications of these properties, we rely on a running example: the Independent JPEG Group's image-decoding reference implementation [20]. This JPEG codec is an important, ubiquitous, legacy program. It is intended to behave as a pure function that reads compressed image data and writes an output bitmap. We have turned this codec into an XFI module, and used that module in the Internet Explorer web browser; we can equally well use it in the Windows kernel. XFI protection implies that image decoding does behave as a side-effect-free procedure.

2.1 External Properties

An XFI module complies with the following policy in its interactions with its system environment:

P1 *Memory-access constraints*: Memory accesses are either into (a) the memory of the XFI module, such as its global variables, or (b) into contiguous memory regions to which the host system has explicitly granted access. In both cases, read, write, and execute operations are handled separately, and may access different regions. In particular, no writes are permitted into the code of the XFI module.

P2 *Interface restrictions*: Control can never flow outside the module's code, except via calls to a set of prescribed support routines, and via returns to external call-sites.

P3 *Scoped-stack integrity*: The scoped stack is always well formed. In particular: (a) the stack register points to at least a fixed amount of writable stack memory; (b) the stack accurately reflects function calls, returns, and exceptions; (c) the Windows stack exception frames are well formed, and linked to each other.

P1 and P3 aim to ensure that state has appropriate values. In general, for memory, we do not know what those values are, so we trust particular components to maintain those values properly—hence we need to authenticate and authorize accesses to memory, as indicated in P1. For the scoped stack, on the other hand, we can formulate integrity properties, and preserve them as invariants, as indicated in P3. As for P2, it addresses external interface protection. Much as in other protection systems [35], P2 does not, by itself, rule out the improper use of legitimate interfaces, for example calling functions in an unsafe order. However, XFI facilitates checking compliance with safety specifications by other means (see Section 2.3).

With XFI, memory and interfaces can be protected at the granularity of individual bytes and calls, respectively. Moreover, different parts of an XFI module may enjoy different access rights, and—via special guards—XFI can support write-once, call-once, and other elaborate policies. For example, our JPEG module can be given access to the exact regions of an image data and an output bitmap (respectively, read-only and write-only), no matter where this memory resides. Moreover, although the JPEG module currently needs no external support, it could be safely allowed to invoke support routines, for instance to output status or error messages.

P1, P2, and P3 are more restrictive than strictly necessary. For example, an XFI module might safely be allowed to overwrite certain constants within its code section or to call other XFI modules instead of just host-system support routines. However, in what follows, we do not relax these properties but rather strengthen them.

Our additional restrictions do not, in practice, hinder the use of XFI protection for existing software; they do, however, generally help our implementation. In particular, the following restrictions P4 and P5 imply that, on the x86, we need not consider instructions with esoteric external effects on segments, descriptors, or page tables. Thus, P4 and P5 help ensure that an XFI implementation is complete and correct, despite complexities of the hardware architecture.

P4 *Simplified instruction semantics*: Certain machine-code instructions can never be executed. These include dangerous, privileged instructions (e.g., modifying x86 task descriptors), as well as potentially harmless but deprecated instructions (e.g., far jumps between segments). Certain other machine-code instructions may be executed only in a context that constrains their effects; this context may be established by static verification, or by a specific dynamic check.

P5 *System-environment integrity*: Certain aspects of the system environment, such as the machine model, are subject to invariants. For instance, the x86 segment registers cannot be modified, nor can the x86 flags register—except for condition flags, which contain results of comparisons.

In particular, these properties allow the x86 flags register to be safely saved on the scoped stack. They also allow the safe use of x86 programmed-I/O instructions, via a guard that restricts dynamically the I/O ports used. The stack-register integrity of P3(a) is a special case of P5.

System-environment integrity is a key advantage and differentiator of XFI protection. It is especially crucial to highly privileged host systems, such as operating system kernels. Without this protection, any computation, when run at high privilege, can fatally corrupt system state in a number of ways—not only through an incorrect, stray write to memory. In particular, XFI can restrict dangerous operations, such as modification of the x86 flags register, so that they are safe. For example, even if our JPEG module allowed buffer overflows, it could never tamper with the flag that controls the direction of REP memory operations. Thus, XFI can eliminate the possibility of unexpected ring changes, x86 "double faults", and other fatal conditions.

2.2 Internal Properties

In addition to restricting interactions between a module and its host, XFI places constraints on the execution of the module:

P6 *Control-flow integrity*: Execution must follow a static, expected control-flow graph, even on computed calls and jumps. In particular, function calls must target the start of functions, and those functions must return to their callers.

P7 *Program-data integrity*: Certain module-global and function-local variables can be accessed only via static references from the proper instructions in the XFI module. These variables are not subject to computed memory access.

P6 and P7 enable a simple, practical XFI verifier. We would find it much harder to have P1–P5 without P6 and P7. In particular, P6 prevents jumps that could circumvent guards, and P7 serves for securing temporary state. On the x86 architecture, P6 also supports P4 by preventing jumps into the middle of an instruction, where another, forbidden instruction may be encoded.

Furthermore, XFI protection constrains software internals in ways that make many attacks impossible. In combination with P1–P5, P6 implies that attackers cannot subvert control flow, and P7 implies that attackers cannot overwrite certain program state, such as global variables that hold configuration data or function-local variables that indicate authentication status. These properties offer a strong defense against actual attacks (see [2, 12, 32]). Although an XFI module may still allow certain buffer overflows and other vulnerabilities, XFI protection can prevent exploits of those vulnerabilities. In particular, XFI guarantees correct function returns (cf. [13]), and prevents introduction of new machine code as well as unexpected uses of existing code. Thereby, XFI foils two of the most popular exploit techniques: code injection and jump-to-`libc` attacks [32]. For example, our JPEG module cannot be exploited by any of the several published JPEG attacks [2], despite originating in the same source code as vulnerable versions.

2.3 Additional Properties

P6 and P7 can also serve as a foundation for the dynamic enforcement of additional properties, such as the validity of system-call arguments and (more generally) compliance with specifications for the use of interfaces. P6 implies that dynamic checks cannot be circumvented, while P7 provides secure storage for state used by those checks.

Going further, P1–P7 support the enforcement of many other useful properties. We have implemented the following one, because it provides a guarantee that is important in our application of XFI to Windows device drivers, described in Section 5.

> *Assured self-authentication*: An XFI module authenticates itself to the host system, e.g., by passing an immutable value chosen by the host system as an extra argument to support routines.

We have considered several other properties (without yet finding the need to realize them). For instance, when it is difficult or awkward to implement timeouts (e.g., in a kernel interrupt handler), XFI could bound the length of execution before a support routine is invoked. Alternatively, XFI could allow interrupts to be disabled when they are re-enabled within a fixed number of instructions.

3 Mechanisms

Several mechanisms can enforce the policies of Section 2. We have chosen mechanisms that allow for an x86 software implementation with good runtime performance. However, our mechanisms should apply to any architecture, and could be combined with hardware-based means of protection.

In our choice of mechanisms, we have considered not only efficiency but also trustworthiness. Because of restrictions such as P4, our mechanisms safely exclude problematic aspects of the x86 architecture. Also, complex rewriters and compilers are not in our trusted computing base. Crucially, correctness depends on an independent static verifier. In principle, correctness could be established formally by analysis of this verifier. We have created proofs for CFI for a small machine-code language [1]; in further work, those proofs could be extended to the full XFI system given a partial formal model of the x86 instruction set (e.g., as in [3, 11, 25]).

This section describes the XFI mechanisms and outlines their implementation. The next section provides further details.

Static Code Inspection The XFI verifier checks statically that each XFI module has the appropriate structure and the necessary guards. For instance, the verifier checks that the XFI module includes only certain instructions. The guards should ensure that, when executed on the intended host system, the module will satisfy sufficient conditions for properties P1–P7 and, optionally, other requirements specific to the host system.

In particular, the verifier establishes constraints on control flow and memory accesses. For direct branches and jumps that statically meet those constraints, no CFI guard is required. Similarly, for direct access to global variables in the XFI module and to virtual registers on the scoped stack, no memory-range guard is required. In other cases, the verifier ascertains the presence of proper guards; these guards must precede the instruction in question along all execution paths.

XFI verification can be done by the users of extensible systems, in order to establish the safety of an XFI module before it is installed or executed. Optionally, verification could be repeated at runtime, for example to counter the corruption of read-only memory by an external device. Verification may be useful in other scenarios as well. For instance, for systems structured as distinct modules with clearly defined interactions (e.g., with COM [7]), verification may be a quality-assurance step during software production.

Section 4.3 gives further details on the verifier.

CFI Guards on Computed Control-Flow Transfers As indicated above, computed control-flow instructions may be preceded by CFI guards that ensure control transfers are within the allowed control-flow graph.

In one implementation [2], a fresh, constant identifier is assigned to sources and targets of computed control-flow transfers. An identifier is embedded within the code of a guard at each source and immediately before the code of each target. At runtime, the guard compares the identifier of the source and target, and ensures that they match. The verifier knows the identifiers and ensures that they appear in the code segment only at valid targets.

Figure 2 shows machine code for a guard. (For clarity, this example, and most other machine-code in this paper, is written in a generic, 32-bit assembly notation with x86

```
    EAX := 0x12345677     # Identifier - 1
    EAX := EAX + 1
    if Mem[EBX - 4] ≠ EAX, goto CFIERR
    call EBX
...
    0x12345678         # Target identifier
L:  push EBP                   # Callee code
```

Figure 2: A computed call instruction, with a CFI guard and one valid callee destination.

register names.) This guard checks that right before the target destination EBX is the four-byte constant identifier 0x12345678. The guard embeds that constant minus one as a literal, not the constant itself, so that the guard's code bytes do not become a valid target. Not shown is the guard that ensures that EBX points to XFI module code.

A guard is needed for computed jumps that may otherwise target unexpected addresses—because of error or attack—but not for function returns, since function-return addresses are kept safe in virtual registers (described below).

Two Stacks XFI makes use of two stacks: the regular execution stack for its scoped stack, as well as a separate allocation stack. This separation prevents the corruption of values, such as return addresses, that are subject to integrity guarantees. The scoped stack is used only in a stylized, structured manner, and is never the target of a computed memory access; therefore the integrity of values on the scoped stack can be established by the XFI verifier. The allocation stack is used for those stack values that an XFI module may access via a pointer. These include all local variables whose address is taken (e.g., arrays, or variables passed as call-by-reference arguments to functions).

At runtime, guards ensure that the two stacks do not overflow, and, thereby, that property P3(a) holds. Such guards must be placed at each module entry point, as well as in any cycle in the module's function-call graph. (Similar guards have been used for implementing non-contiguous stacks for lightweight threads [40].)

Statically, all explicit references to the scoped stack must be to SSP+K, where K is a positive, properly aligned constant, and SSP is the scoped-stack pointer register. (On the x86, SSP is the regular x86 stack register ESP, so that pushes, pops, calls, and returns use the scoped stack.) Furthermore, at each instruction of a module function, the scoped stack must be of a known, constant depth. Within each function, SSP can be lowered and raised only by constant amounts, in a controlled fashion. SSP is preserved by function calls.

The memory of the scoped stack is not accessible to an XFI module, except in the manner described above. Therefore, locations on the scoped stack can function as virtual registers: like registers, they are thread-local state that is only accessed by name. In realizing property P7, these virtual registers can hold local function variables and other XFI module state to be protected from memory corruption within the module.

On the other hand, the allocation stack is used for thread-local module data that is accessible via pointers (and, thus, potentially vulnerable to pointer errors). The allocation stack implements an efficient alternative to the heap allocation of this data; it is indexed by the allocation-stack pointer (ASP) register. (ASP can be a virtual register or, on the x86, the register EBP.) Within each function, ASP can be modified only in a controlled fashion, by either a constant amount or one bounded by a runtime check. ASP is preserved by function calls.

XFI requires only the scoped stack for certain simple modules, including those where arrays and pointer-accessible variables reside only on the heap. A host system that loads only such modules may omit all support for allocation stacks. In general, however, host systems must support this two-stack execution model, in particular by managing allocation stacks and by properly passing arguments and results between stacks. Section 4.5 describes the components of this support.

Guards on Computed Memory Accesses Whether it uses an allocation stack or not, a module must be able to access the memory that contains its code, read-only constants, and writable global variables, in order to execute, read, and read or write that memory, respectively. In addition, a host system may wish to give selective access to other memory regions. XFI uses guards for enabling the use of those additional regions—in any number, at any granularity, and for any type of access.

An XFI module has access only to a certain set of contiguous memory regions; for each type of access, a certain range $[\,A\,,B)$ may be special, as explained below. For accesses to constant addresses and to the scoped stack, this property is established by static verification. Guards check other accesses at runtime.

XFI memory-range guards ensure that a register holds an accessible address; moreover, they ensure that a range around this address is also accessible (within non-negative constant offsets L, below, and H, above). Control-flow integrity implies that a single such guard can protect multiple memory accesses—namely, those that it dominates, use the same register value, and remain within the range determined by L and H. Similar (but more complex) memory-range checks ensure that only accessible addresses are used in simple loops such as the x86 REP instructions. In this case, L and H may be variable and reside in registers.

The comparisons in memory-range guards must take into account the sizes of memory accesses, and H must be chosen accordingly. For example, the guard for a

```
   # mrguard(EAX, L, H) ::=
     if EAX < A + L, goto S
     if B - H < EAX, goto S
M:   Mem[EAX] := 42      # Two writes
     Mem[EAX - L] := 7   # both allowed
...
S:   push EAX            # Arguments for
     push L, H           # slower guard
     call SlowpathGuard
     jump M              # Allow writes
```

Figure 3: Two memory writes, and a memory-range guard for the range [EAX-L, EAX+H). The guard executes faster if this range lies within [A, B). The constant H should be at least 4.

four-byte write through EAX must ensure that all bytes in the range [EAX, EAX+4) are writable. The guards must also handle corner cases, such as when A is near the end of addressable memory. In the common situation where A+L and B−H are constants established by the loader (as described in Section 4.1), the corner cases can be treated statically. For variable memory ranges, like those of x86 REP loops, guards must perform additional runtime checks. The guards must also consider the possibility of arithmetic overflows.

Figure 3 shows an example memory-range guard that establishes that access is allowed to all addresses in the range [EAX-L, EAX+H). The figure includes two write instructions that each accesses four bytes of memory through EAX. For the guard to check the writes correctly, the offset H must therefore be at least 4. The figure also introduces a shorthand mrguard. Below, mrguard(EAX, L, H) represents the contents of the figure minus the two writes. As shown in the figure, memory-range guards can be implemented as two paths: one faster, the other slower.

- The fastpath directly compares an address in a register with the values A+L and B−H. The guard permits access if the address lies within these bounds.
- If the fastpath comparisons fail, then the guard calls a host-system slowpath with appropriate arguments. The slowpath searches to see if the address range lies within any other accessible memory regions. The address, and the values L and H, are parameters to this search. The search itself may be arbitrary code, and may for example involve direct comparisons or data structures similar to page tables. The search may be invoked by a direct jump, a trap or fault (e.g., the x86 bounds exception), or by other explicit or implicit control flow.

Analogously, CFI guards could also use a slowpath when embedded identifiers do not match, thus supporting a rich, dynamic notion of validity for function pointers. We have not yet needed this extension.

4 Implementation

We have designed and built an XFI implementation. This section describes it, and also considers optional, simple, specialized hardware support for XFI. Although still a prototype, the implementation is already complete enough to be practical, as demonstrated by the applications and measurements described in later sections.

4.1 XFI Modules

XFI modules are dynamically loadable executables in an appropriate object format. In our implementation, they are Windows "Portable Executable" binaries, often named EXEs or DLLs [34].

Modules consist of multiple sections, which may have different access permissions. Machine code is in one executable section, and program data, such as read-only constants and writable global variables, is in others. Data in import and export sections allow the determination of module entry points and use of host-system support routines. Other sections, and the module header, provide host systems with auxiliary data (for example, cryptographic signatures and offsets for load-time relocation). Auxiliary sections are used only at load time.

XFI protection thus relies on several module sections. Import sections, as well as host-system policy, limit an XFI module's use of support routines; similarly, memory access is constrained by section access permissions. The relocation section may give values to XFI constants, such as the constants A+L used in memory-range guards. Finally, a new, auxiliary section holds untrusted verification hints (discussed in Section 4.3).

The efficiency of memory protection can benefit from choices in the structure of XFI modules. Specifically, writable fastpath memory may lie completely within a read/write section of each XFI module: once a module is loaded, this section can have any amount of memory. (A section-header value gives its size.) In this case, the module, or its host system, may provide a heap implementation that allocates memory within that section. Such a fastpath region can be made large without wasting physical memory: host systems with virtual memory support can allocate physical pages as requested by the XFI module (e.g., by the heap implementation calling a support routine). This strategy is especially attractive for 64-bit systems, where ample virtual size can be given to a fastpath region.

4.2 The Rewriter

We produce XFI binary modules from Windows x86 executables with a rewriter based on the Vulcan library [37]; XFI rewriters could as easily be created using similar libraries for other architectures [37, Section 7]. Although our x86 rewriter requires neither recompilation nor source-code access, it makes use of debug information (PDB files), for instance to distinguish code from data. Such PDB files are publicly available for all Windows components. Our rewriter is relatively complex and not very fast. In particular, it must do intra-procedural analyses for control flow, stack use, and register use; these analyses are not always linear.

Our rewriter does not currently handle certain hand-written modules; it also stumbles on frame-pointer-optimized code, and code with certain variable-size stack allocation. (Both can be supported using a virtual register.) Similarly, our verifier does not handle MMX, SSE, and other x86 instruction-set extensions. However, since our verifier is conservative, neither these limitations nor rewriter bugs should ever result in the execution of an XFI module without XFI protection.

Alternatively, XFI modules could be output by the code-generation phase of a compiler. A compiler that would produce XFI modules would generate mostly standard machine code, but add structured inline guards and verification hints, and use the allocation stack for stack data subject to computed memory access. Such a compiler could make more thorough use of XFI mechanisms, help reduce overhead, and remove some limitations of our prototype.

4.3 Verification

As discussed in Section 3, the correctness of XFI protection depends on the load-time verification of XFI modules. Our verifier was written from scratch and is self-contained. In particular, it is independent from our rewriter, and from any specific strategy for creating XFI modules. It is 3000 lines of straightforward, commented C++ code, most of which are tables for x86 opcode decoding. The verifier needs only a basic understanding of x86 behavior, modeling nothing more complex than integer comparisons and how instructions copy registers. Its simplicity contributes to our confidence in the verifier.

The verifier is much simpler than the rewriter because the verifier does only local reasoning about individual basic blocks. This reasoning takes advantage of a set of untrusted verification hints that must be present in the XFI module. A similar strategy is used by Java bytecode verifiers and in PCC systems.

Therefore, the verifier is not only simple but also fast. It makes a linear pass over the bytes of an XFI module, doing mostly instruction decoding and comparisons.

In order to establish the correct use of other XFI machinery, the verifier checks several specific conditions. These conditions refine and strengthen the requirements outlined in Section 3. In particular, the verifier ensures that both the allocation stack and the scoped stack are managed properly: the allocation-stack and scoped-stack pointers are updated only by bounded, constant amounts, the code contains stack overflow guards, the return address and other virtual registers are saved and retrieved from the scoped stack only, and function calls preserve the heights of both stacks.

Verification proceeds by considering the execution of machine-code instructions abstractly; it manipulates verification states which are predicates that describe concrete execution states. In particular, verification states model register contents, including the contents of the scoped stack which holds function-local virtual registers. The execution of each instruction `i` can be represented by a Hoare triple: $\{\, P \,\}\ \mathtt{i}\ \{\, Q \,\}$, where P and Q are verification states. Given P, the verifier ensures that P guarantees the safe execution of `i`; the verifier also computes Q and ensures that Q implies the verification states before each possible successor instruction.

To simplify verification, XFI modules must include a set of verification hints that guide the verifier. For the most basic version of the verifier, the hints must provide the verification state for the entry to each basic block. The verification is done one basic block at a time, and on basic-block exit the verifier checks that the final, computed verification state implies the verification state at entry to all possible successor blocks.

As mentioned above, the verifier also ensures that the proper CFI guards precede computed control-flow transfers. The verifier expects hints to specify the set of possible targets of computed jumps; these hints allow the verifier to collect the set of CFI target identifiers used in the module. The verifier scans the machine code of the module in order to ensure that these identifiers occur only at the beginning of basic blocks. Finally, the verifier allows computed control flow only when the verification state records the effects of an appropriate CFI guard. Similarly, the verifier allows a computed memory access only when the verification state records the effects of an appropriate memory-range guard.

Figure 4 shows an example program fragment. The memory-range guard in line 0 checks that the memory in the range $[$`EAX-0`$,$`EAX+8`$)$ lies within a single accessible memory region. (See Figure 3 for a definition of `mrguard`.) This fragment copies the word stored at address `EAX` to address `EAX+4`, loads the result value from the allocation stack into `EAX` (in line 3), then restores the allocation-stack pointer (`ASP`) by loading it from a virtual register on the scoped stack, and returns. As shown in the comments, the pop and return instructions use the scoped stack.

At the start of verification of the code in Figure 4, the verification state encodes that the original value of the scoped stack pointer is 8 more than the current value, the return address is in the virtual register at address `SSP+4`, the value on function entry of the allocation-stack pointer is stored in the virtual register at address `SSP`, and the allocation-stack frame range $[$`ASP-32`$,$`ASP`$)$ falls into a

```
   Code  Verification state
         {origSSP=SSP+8, valid[SSP, SSP+8) }
         {retaddr=Mem[SSP+4]}
         {origASP=Mem[SSP], valid[ASP-32, ASP)}
0. mrguard(EAX, 0, 8)
         {valid[EAX-0, EAX+8)}
1. EDX := Mem[EAX]
2. Mem[EAX + 4] := EDX
3. EAX := Mem[ASP - 4]
4. pop ASP # ASP := Mem[SSP]; SSP := SSP+4
         {origASP=ASP, valid[SSP, SSP+4)}
         {origSSP=SSP+4, retaddr=Mem[SSP]}
5. ret      # SSP := SSP+4; jump Mem[SSP-4]
```

Figure 4: Verification states for a small fragment of XFI module code. The items written in braces correspond to the verification state at the given program point.

contiguous accessible region of memory. For clarity, Figure 4 shows only modifications to the verification state. The verifier recognizes the meaning of the instructions of the memory-range guard in line 0, and adds the corresponding "valid" fact to the verification state for the following instruction. All of the previously known facts are preserved in this case, and the resulting "valid" facts suffice for checking all the memory accesses in the program fragment. In line 4, the verification state is changed to reflect the arithmetic operation implicit in the `pop` instruction. After line 4, the verification state considers valid only the range [`SSP`, `SSP+4`), since in systems code the contents of the stack below `SSP` can always be clobbered by interrupts. At line 5, the verifier checks that the stack pointers will be restored to their original values before the return, and that the proper return address is used. In order to enable this reasoning, the verifier includes support for manipulating linear equalities and inequalities.

4.4 Inline Guards

Many considerations influence the choice of the exact machine-code sequences for XFI mechanisms such as the inline guards. Finding suitable, efficient guards can be particularly challenging for the x86, because of its complex, non-uniform instruction set, and dearth of registers. In our XFI implementation, we address these x86 kinks partly through a register-liveness analysis that the rewriter employs to discover or make available free registers or x86 condition flags. We also use the following specific code sequences for each type of guard.

CFI Guards Our x86 CFI guards are implemented in much the same manner as shown in Figure 2, and detailed in [2]. Instead of having identifiers precede valid destinations, we embed callee identifiers within an instruction, `prefetchnta`, that has few side effects.

```
    EAX := SSP - L - K
    if EAX < Mem[FS + 8], goto SSPERR
C:  SSP := SSP - L        # Lower SSP by L
```

Figure 5: Scoped-stack overflow guard on Windows.

Stack-Overflow Guards We employ memory-range guards in order to prevent allocation-stack overflow: if `ASP` is to be lowered by `L`, the range [`ASP-L`, `ASP`) must lie within an accessible region.

Our guard against scoped-stack overflow exploits a particular property of x86 Windows: the bottom of the current stack is held in directly accessible, thread-local storage pointed to by the `FS` segment register. (Future, Windows-specific implementations might usefully keep XFI-specific data in this thread-local storage.)

Figure 5 shows an x86 scoped-stack guard; it compares a proposed `SSP` value with the stack limit at `FS+8` in the thread control block. Our guards ensure that free space always remains at the bottom of the scoped stack (e.g., for interrupt state). Thus, in the figure, code at label `C` lowers `SSP` by `L`, but our overflow guard ensures that `K` more space is available, where `K` is a constant.

Allocation-stack overflow guards can also be implemented in a host-specific manner. For instance, for single-page allocation stacks (such as may be used in the Windows kernel), an invariant on the top bits of `ASP` could allow for a fast guard—even when allocation stacks are placed in slowpath memory.

Memory-Range Guards Our implementation uses a fastpath memory region whose endpoints `A` and `B` are embedded in the memory-range guards. This region lies within the XFI module, as suggested in Section 4.1; its endpoints are relocation constants set during loading. In theory, the endpoints could be fairly arbitrary, even values held in reserved registers—although our implementation does not spend x86 registers for this purpose.

Our x86 memory-range guard is like that shown in Figure 3 of Section 3. The slowpath call is not placed inline, both because it would waste code cache and also because the x86 predicts forward branches as not taken, by default. We have considered having our guards raise exceptions, caught by a slowpath guard provided by the host system; however, this approach is likely to be less efficient than guards that call slowpaths. In particular, our implementation cannot use the x86 `BOUND` instruction because hardware bounds exceptions are treated as fatal conditions in the Windows kernel.

4.5 Host-System Support

Most XFI modules need runtime support from their host system, as discussed above. This section describes the support components that host systems must implement in order to make use of XFI protection.

As an alternative, instead of the host system implementing them separately, these support components may be included with XFI modules, for instance in a distinct library-code section. Such library code may be trusted because of cryptographic signatures or for other reasons; therefore, this code may even include the XFI verifier.

Slowpath Permission Tables When executing their slowpath, memory-range guards must search permission tables: runtime data structures, maintained by the host system, that hold the set of accessible memory regions. There may be multiple such tables, one for each type of slowpath access. Each search checks whether a range $[$`R-L`, `R+H`$)$ lies within a contiguous memory region to which the host system has granted access.

Searching these tables must be fast, as some software will access memory in a manner that frequently calls the slowpath of memory-range guards. Fortunately, fast permission tables can be implemented in several ways. In particular, by using tables similar to page tables, searches can use known, efficient techniques for software-filled translation lookaside buffers, or TLBs [43].

In our implementation, permission tables are very simple: a null-terminated list of address pairs, of the start and end of regions. Even though there are faster alternatives, we chose this representation not only because it is simple to maintain and search, but also because our experiments indicated that more complex tables would be of limited benefit.

Allocation-Stack Manager A host system must properly associate an allocation stack with each thread that executes in an XFI module. Allocation-stack management must consider both performance and resource consumption. For instance, a thread can keep using the same allocation stack if it calls an XFI module reentrantly; alternatively, it may adopt a new allocation stack on each entry.

Our implementation uses a pool from which host-system threads draw allocation stacks when they call XFI modules. The size of the pool is adjusted on the basis of the concurrency in the XFI module. The pool's data structure is a simple array, guarded by a single lock; this array is consulted in order to acquire and release allocation stacks, as threads go through software call gates.

Software Call Gates An important component of XFI is the software that mediates calls to entry points and support routines, in particular to map to and from the two-stack model used by XFI module execution. These wrappers are a software form of call gates, as found in some hardware architectures [43]. They will typically be implemented by the host system, and may perform all the work necessary to manage stacks and to maintain permission tables. Our wrappers, in addition, resolve complications such as the multiple calling conventions of x86 Windows and variable-argument support routines.

Instead of copying arguments and results, as in component systems [7], our wrappers can edit the slowpath permission tables to marshal access rights on calls and returns. This alternative is important when it is not desirable to copy into and out of XFI module fastpath memory, such as for large, infrequently used data buffers. Device drivers may be good candidates for this optimization, as they often manage large buffers, while performing little processing of those buffers. (For example, a storage driver may never examine the data being read or written.)

Windows Exception Dispatcher The Windows software execution model includes a unified mechanism for handling software and hardware exceptions, both synchronous and asynchronous. This Structured Exception Handling (SEH) [31, 34] applies both in user mode and in the kernel; it generalizes Unix signals and C++ exceptions, and can be used to implement both. On the x86, SEH requires functions that catch exceptions to place relevant SEH metadata on the stack, including the address of a catch expression and a catch handler. (On the other hand, 64-bit Windows places the metadata for all functions in a static, read-only table.) Function prologues and epilogues link this metadata into a chain of handlers, whose head is at `FS+0` in the thread control block. Upon an exception, catch expressions along this chain are evaluated, determining which catch handler is used. Multiple, strong invariants apply to this metadata, as well as to the handler chain.

XFI protection for Windows must consider SEH, both to enforce policies (e.g., P6), and to support correct execution of Windows software, which often makes use of SEH. In particular, XFI should allow both catch expressions and catch handlers to run on both XFI stacks, and (possibly) allow control to resume at the faulting instruction. Fortunately, XFI mechanisms such as virtual registers enable SEH integrity to be fully maintained. This integrity results in important security benefits: SEH corruption is a favorite means of attack on Windows [32].

We have designed support for SEH in our XFI implementation; we outline it next (omitting details because of space constraints). We have not yet fully built this support, but our implementation does handle Unix-style signals. The support includes both the necessary host-system components and what conditions are checked by the XFI verifier. All exceptions are directed to our exception-dispatcher implementation in the host system; it saves registers, and invokes the XFI module's SEH code in the proper order, and using the right context (e.g., for the ASP register). The XFI verifier ensures each function's SEH metadata is well formed, and used appropriately; it also restricts catch expressions so they properly access stacks via a base pointer.

4.6 Possible Architecture Support

The specifics of XFI mechanisms and their behavior depend on hardware characteristics. In particular, the code and the cost of software guards will vary across platforms. Therefore, we have designed and evaluated relatively straightforward hardware support for CFI guards and memory-range guards in an Alpha simulator. (No precise x86 simulator is available.) This support extends the Alpha instruction set with a few, new register-based comparisons instructions.

We implemented CFI guards with four new instructions: a `cfilabel` instruction and a variant of each of the Alpha computed transfer instructions (“indirect jump”, “return”, and “jump to subroutine”). The instructions contain 16-bit, immediate identifiers. After a CFI transfer instruction with identifier `ID`, a `cfilabel` with `ID` must be executed before any other type of instruction; otherwise, a hardware exception is triggered. Between these instructions, the value `ID` is stored in a normal, renamed register, so multiple CFI guards can be in flight simultaneously. To avoid reads of this CFI register at each instruction, it is monitored with a two-state automaton in the commit stage of the pipeline. An x86 implementation could be similar, except that it might use instructions up to six bytes long (e.g., in order to allow larger identifiers) and an explicit CFI register.

We implemented memory-range guards with three new instructions, one for each type of access (read, write, and execute). Each instruction takes the form `mrguard R, L, H`, naming a register `R`, and providing `L` and `H` as 10-bit immediate constants. The instructions compute values `R-L` and `R+H` and compare them with registers that hold `A` and `B`, using the regular, scheduled arithmetic units of the processor. (Thus, the instructions implement `mrguard` as defined in Figure 3.) On fastpath failure, our instructions (like the x86 `BOUND` instruction) throw a hardware exception that provides `R`, `L`, and `H` only as implicit arguments via the faulting instruction. An x86 `mrguard` instruction could perform a real call, and push arguments onto the stack, with accordingly higher performance; it could be implemented in microcode, with the address of the slowpath function to call in a machine-specific register.

We studied these new instructions for XFI guards in a validated, cycle-accurate Alpha EV6 simulator [14]; the details are in a companion report [9]. Our results indicate that hardware-supported XFI can be quite efficient: adding effect-free `NOP` instructions instead of XFI guards leads to virtually identical overhead as using the new instructions. In part, these results reflect that our XFI instructions introduce no new dependencies, and can leverage unused processor units; they also reflect cache pressure due to increased code size. These results are likely to carry over to the x86 architecture. Section 6.2 gives measurements of the overhead of using x86 `NOP`s in place of XFI guards.

5 Applications

XFI has a wide range of potential applications. To date, we have applied our XFI implementation to dynamic libraries, device drivers, and multimedia codecs. In this section we describe some of these applications; for brevity, we focus on device drivers.

Device drivers are a prime application domain for XFI protection, both because they should comply with XFI policy and also because their failure to do so has serious consequences.

We have implemented XFI for kernel-mode device drivers developed using the Windows Driver Foundation (WDF)—a new-generation framework, designed to simplify the development of correct drivers for all versions of Windows [26]. With WDF, a device driver no longer has direct access to kernel abstractions, such as work-item queues, request packets, or device objects; rather, these are opaque to the driver, and WDF performs all modification and synchronization on the driver’s behalf. In addition to holding abstract such critical state, WDF validates that drivers comply with contracts, even on production systems. Thus, WDF represents a significant advance in interface safety over previous driver support frameworks [30].

XFI protection can isolate device drivers in a WDF host system. Without this isolation, despite all its interface-safety checks, it is impossible for WDF to contain driver faults (or exploits) fully, or even to detect them in all cases. Since WDF mediates on all interactions with the kernel, device drivers interact with their host system frequently; therefore, XFI protection is more attractive than potential alternatives with higher context-switch latency.

We have added to WDF the necessary components for XFI protection. These include the machinery described in Section 4, as well as some WDF-specific machinery. In particular, this machinery enforces the assured self-authentication policy (described in Section 3), as follows. Since WDF can serve multiple drivers simultaneously, drivers pass along a pointer to their WDF control block when they call support routines. A driver may attempt to spoof this value in order to call WDF as another driver. We thwart such impersonations by requiring the first argument to each WDF support routine to be a specific, read-only variable in the XFI module of the calling driver. WDF has write access and sets this variable as it loads the driver.

It is simple to determine the set of memory regions that a WDF driver should be allowed to access. A driver gains access to memory either through explicit allocation—a call to a WDF support routine—or implicitly, such as when an I/O buffer is passed to the driver for processing. In all cases, WDF has the information necessary to grant (and later revoke) access to those memory regions, by modifying the slowpath permission tables.

We have applied XFI to some WDF drivers, including a RAM disk driver and a benchmark driver used by the WDF team to measure performance. Section 6 gives the results of these experiments.

Multimedia codecs are another attractive application domain for XFI protection. They are typically extensions (often downloaded, untrusted code) that operate on data at the request of their host system. Furthermore, they have been the subject of many successful attacks. As described in Section 2, we have turned the standard JPEG implementation of image decoding [20] into an XFI module. Section 6 gives the results of these experiments as well.

6 Evaluation

We have performed a number of experiments and analyses in order to evaluate the benefits and overheads of XFI protection. This section summarizes our results.

6.1 Protection Benefits

XFI protection against faults such as spurious writes to memory is similar to the protection of hardware-supported address spaces (e.g., processes). However, some aspects of XFI protection do not lend themselves to simple comparisons; for instance, XFI enforces integrity guarantees that protect against security exploits. We have validated these XFI benefits, in part by experimenting with actual exploits.

We have established that CFI guards alone are sufficient to prevent a wide variety of attacks that follow deviant machine-code execution paths and thereby violate XFI policy. In particular, CFI blocks the exploits used by Blaster and Slammer and those in a test suite of 18 other attack vectors; CFI also thwarts the published exploits of a heap overflow in the widespread software used in our JPEG module. More details can be found in [2].

XFI also helps defend against data-corruption exploits For example, in an XFI module, if a global configuration variable does not reside in a region that is subject to computed memory access, then that variable can be modified only by those instructions that refer to it by name; similarly, access to function-local variables is limited. In this manner, XFI program-data integrity can effectively thwart exploits described in a recent paper on data-only attacks [12].

Unfortunately, some attacks still succeed, despite XFI protection. Most notably, these include Nimda-like attacks that abuse over-permissive support routines. Defense against such attacks remains an open problem.

6.2 Enforcement Overhead

We applied our x86 XFI implementation to WDF device drivers, SFI benchmarks [36], the JPEG decoder [20], and Mediabench kernels [22], all compiled using Microsoft VC++ 8.0, with optimizations. Rewriting each module took a few seconds; verification is linear in the module size and typically takes a few milliseconds. We made elapsed-time measurements for a Pentium M 1.5GHz processor, on an idle system with daemon services disabled. The processor was fully utilized in all runs. (For drivers, 90% of the time was in the kernel.)

Tables 1, 2, and 3 summarize the results of our measurements; the tables report overheads relative to modules before XFI rewriting. Overheads are the average of five runs of more than ten seconds, with standard deviation less than 1.5%. Overheads are shown for *write protection*, which restricts only writes, and (between parentheses) for *read-write protection*, which also restricts reads. We emphasize write protection because integrity (rather than confidentiality) is usually the central goal of protection.

Our tables have three columns of results. The slowpath column shows overhead when memory-range guards have to consult permission tables for access to the data processed by the XFI module. The fastpath column shows results when data already resides within the module's fastpath region—for instance, because the host system passed a copy of the data when it called the module entry point. Both columns reflect the case where XFI modules have a private, fastpath memory. Memory-to-memory copies are fast on modern processors, and fastpath regions can be large (as discussed in Section 4.1), so host systems may find it most convenient to copy arguments to and from XFI modules; in this case, only fastpath overheads apply. Slowpath memory, and the techniques of Section 4.5, may be used only rarely, in particular when state truly needs to be shared.

The NOP column shows the measured overhead when each inline guard is replaced with a six-byte x86 NOP instruction. Therefore, it gives a baseline for the cost of structuring binaries as XFI modules, and of including guards in the appropriate places. In particular, the NOP column shows overheads that results from increased cache pressure and from additional instruction decoding. As discussed in Section 4.6, NOP overheads may also be indicative of the overheads that would be seen with hardware-supported XFI: on the x86, six bytes are a reasonable size for new XFI guard instructions.

Each table also shows the code size of each XFI module as a multiple of the code size of the original binary. The size increase results from inline guards, and is often substantial; however, the added code is often not executed, since our x86 guards are implemented with an out-of-band slowpath (as shown in Figure 3).

		NOP	fastpath	slowpath
hotlist	**Δ sz**	2.1x (2.6x)	2.5x (4.1x)	3.9x (8.3x)
	%	1% (5%)	4% (94%)	5% (798%)
lld	**Δ sz**	1.2x (1.3x)	1.5x (1.8x)	1.7x (2.3x)
	%	10% (28%)	27% (60%)	93% (346%)
MD5	**Δ sz**	1.1x (1.1x)	1.2x (1.3x)	1.3x (1.5x)
	%	−1% (2%)	3% (7%)	27% (101%)

Table 1: Code-size increase and slowdown of SFI benchmarks, with XFI. The % rows show slowdown.

	Kt/s	NOP	fastpath	slowpath
Δ sz		1.3x (1.3x)	1.3x (1.4x)	1.4x (1.6x)
1	193	5.0% (4.8%)	6.8% (6.1%)	5.9% (13.4%)
512	151	4.7% (3.9%)	5.3% (4.7%)	4.8% (10.6%)
4K	71	1.7% (1.7%)	2.7% (2.9%)	2.6% (5.0%)
64K	5	1.2% (1.9%)	1.4% (0.4%)	1.7% (1.8%)

Table 2: Code-size increase and slowdown for different kernel buffer sizes for a WDF benchmark, with XFI. The unprotected driver is 11KB of x86 machine code; its transactions per second (shown in thousands) form the baseline for the slowdown percentages.

Table 1 shows numbers for well-established benchmarks for SFI performance [16, 25, 36]; each is less than 3K of machine code. Unlike XFI, SFI supports only one accessible memory region, so only XFI fastpath overhead is directly comparable with published SFI measurements. This overhead is either similar to that of SFI or significantly lower. For instance, XFI fastpath overhead on MD5 is only 3%; for SFI, the corresponding numbers range from 23% [36] to 47% [25]. XFI can have lower overhead than SFI because, with XFI, a single memory-range guard can suffice for multiple memory-access instructions. The hotlist benchmark searches a linked list, in a tight loop; each pointer must be checked separately. Therefore, its read-write slowdown (up to nine-fold) reflects the use of a memory-range guard every few instructions.

Table 2 gives the results of running a WDF benchmark with XFI protection, using KMDF 1.0 [26]. In the benchmark, a user-mode program stores in a kernel-mode driver a data buffer of a certain size; this buffer is then retrieved and validated. Table 2 exhibits clear, understandable trends: enforcement overhead is reduced by using either a larger buffer per transaction, or faster and fewer guards. (The table also exhibits small anomalies: sometimes the use of more, slower guards slightly improves performance, possibly because of cache effects.) For this benchmark it was important that XFI is able to guard use of the x86 REP instructions; otherwise, a guard might have to be executed for every buffer byte, resulting in up to three-fold slowpath overhead.

In addition, we experimented with a WDF implementation of a RAM disk driver. We copied 8 megabytes back and forth, between two directories on the RAM disk, flushing the file cache after each copy. We repeated these copies several thousand times, both using single-byte files and files of varying sizes. The RAM disk did not exhibit any measurable end-to-end slowdown, even with slowpath guards. (Similarly, for many modules, the overhead of XFI protection may be overshadowed by other processing.)

	NOP	fastpath	slowpath
Δ sz	1.3x (1.6x)	1.7x (2.5x)	2.1x (3.7x)
4K	14% (34%)	18% (78%)	42% (112%)
14K	15% (36%)	18% (80%)	43% (116%)
63K	12% (31%)	17% (75%)	40% (108%)
229K	11% (28%)	15% (68%)	35% (98%)

Table 3: Code-size increase and slowdown for different-size input data for JPEG decoding, with XFI. The unprotected decoder is 59KB of x86 machine code; the baseline for the slowdown shown is decoding time.

	NOP	fastpath	slowpath
adpcm_encode	0% (4%)	2% (49%)	13% (149%)
adpcm_decode	−3% (2%)	3% (12%)	36% (112%)
gsm_decode	3% (1%)	79% (97%)	125% (230%)
epic_decode	3% (9%)	7% (19%)	119% (220%)

Table 4: Slowdown of Mediabench kernels, with XFI.

Between the above drivers and the WDF, we measured up to 900,000 transitions per second—but each had little cost. In comparison, placing drivers in hardware-supported address spaces causes overhead on each transition (because new page tables must be loaded), as well as during code execution (because of higher TLB miss rate). Such overhead is correlated with the rate of transitions between the kernel and the driver, and at 23,000 transitions per second it can cause more than three-fold slowdown of kernel processing, as reported in [39]. Our experiments concern WDF drivers that have a much higher transition rate; correspondingly, their processing slowdown with hardware-based protection could be much worse. With XFI protection, these hardware-related overheads are fully eliminated.

Table 3 shows results for the JPEG decoder for inputs of four different sizes. Each of the inputs yields an image about eight times larger. Although JPEG processes images in 64-pixel blocks, overhead is somewhat smaller for larger images—again because of fast guards for REP loops. Table 4 shows results for other multimedia codecs, namely three Mediabench kernels (defined as in [9]). Overall, the performance of these XFI multimedia codecs seems acceptable, considering that they are based on unmodified, standard C-language sources, and that they can be used safely within any x86 hardware protection ring.

7 Related Work

Since there is a wealth of work on protection, we briefly review the landscape and include details on only a few specific pieces of related research.

Unfortunately, no single approach to protection will be suitable for all scenarios in modern, commodity systems. Some approaches, like Nooks, are designed for "fault resistance, not fault tolerance" and can be easily circumvented by malicious code [39]. Others, like Mondrix, require sophisticated, new hardware support, or changes to the system foundations for the inclusion of a protection supervisor [4, 23, 44]. In its published incarnations, SFI [16, 25, 36, 41] has enforced a policy more basic than that of hardware-supported address spaces, yet placed hard-to-meet constraints on memory layout and other system aspects, such as signals and multithreading. Language-based approaches [5, 19, 24, 28] usually require re-writing software; they can also require elaborate runtime support that may be difficult to include at all levels of a system.

XFI is founded on the view that protection should be mostly a software issue [6] (cf. [42]). In comparison with traditional hardware-supported address spaces, software machinery can offer almost arbitrary expressiveness. It may also be easier to deploy than new architectures (e.g., [44]). Finally, it can make it efficient to switch protection domains, and it avoids technical difficulties such as the efficient hardware implementation of permission tables. Of course, even software-based approaches require appropriate hardware support [6], sometimes to a non-trivial extent (e.g., x86 segments for protecting a call stack in our previous work on SMAC [2]).

Software protection systems vary widely in their contexts, goals, and techniques:

- Some systems address protection between kernel environments (e.g., [4, 23, 39]), while others target applications isolated by typing (e.g., [19]). XFI aims to be applicable in a broad range of contexts.
- Inlined reference monitors [17] and PCC [29] aim to support the enforcement of a large class of properties. More targeted systems include ours for CFI. XFI leverages CFI in order to offer, systematically, external properties and critical-state integrity. Going beyond protection, some software techniques also address rollback and recovery (e.g., [38]); we have not yet studied them in the context of XFI.
- Some systems rely on interpretation techniques; program shepherding [8, 21] and the use of virtual machine monitors [4, 23, 44] are examples of this approach. Many others (including XFI) rely on language constraints, static and dynamic typing, and other kinds of analyses [15, 19, 27, 33]. Generally, static analysis aims to verify that dynamic checks are not necessary. In XFI and a few other recent systems with other objectives (e.g., [2, 10, 18]), the static analysis aims to ensure that dynamic checks have been applied properly, and it is the responsibility of an independent verifier.

XFI is most closely related to SMAC, mentioned above, and to SFI implementations such as MiSFIT [36]. Indeed, one of the motivations for XFI was our desire for a practical protection mechanism in the spirit of SFI. Originally, SFI was a carefully designed system for enforcing a relatively basic protection policy, with impressive performance [41]. Its RISC-based implementation relied on several significant assumptions: fixed-length instructions, several registers free for dedicated use, a single aligned, contiguous block of memory, and system support based on hardware protection. While these assumptions can be easily satisfied in a new, 64-bit RISC system, they are problematic in many other settings—such as x86 commodity systems. Later SFI implementations have added further assumptions about memory use and alignment, trusted compilers, and hardware support [16, 25, 36]; some maintain additional invariants on certain state, in a limited fashion (e.g., on return addresses in single-threaded programs [36]). In particular PittSFIeld [25] is an attractive, new SFI implementation that applies to CISC architectures. PittSFIeld achieves efficiency by using hardware support for guard pages, reserving large, aligned regions of memory, and not handling race conditions on function returns. Furthermore, it offers limited protection for kernel-mode code (e.g., as it pops the x86 flags register from arbitrary memory).

In comparison with SFI, XFI supports richer policies, for instance for fine-grained access control and the limited, safe use of privileged instructions. Because it depends on few assumptions, XFI is applicable to legacy software, on commodity systems, and even despite vulnerabilities such as buffer overflows. While some SFI performance improvements reduce fault isolation, XFI guarantees protection even with optimizations. Finally, XFI does well in terms of performance; for example, our basic XFI implementation has lower overhead than PittSFIeld for JPEG, and it runs MD5 faster, even when inputs are in slowpath memory.

8 Conclusions

XFI protection allows code to be executed with strong safety and security guarantees, without the need for a separate process or the creation of new software written in a type-safe language. Like hardware protection, however, XFI addresses low-level architectural features; like language-based protection, it leverages static analysis and offers expressiveness. Thus, XFI inhabits an interesting and practical middle ground. In this respect, XFI has much in common with SFI and PCC. Building on SFI, PCC, and related efforts, the design and implementation of XFI include a number of ideas and techniques that contribute to its flexibility, completeness, efficiency, and wide applicability. As a result, XFI offers protection even for legacy code that is run natively in the most privileged ring of x86 systems.

Acknowledgments This work was done at Microsoft Research, Silicon Valley. John Richardson motivated and supported our WDF work. Venugopalan Ramasubramanian helped with experiments. We are grateful to Brad Chen for his careful shepherding, and to the OSDI reviewers and Greg Morrisett for their useful comments.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control flow. In *International Conf. on Formal Engineering Methods*, 2005.

[2] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Computer and Communications Security Conf.*, 2005.

[3] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL*, 2000.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, 1995.

[6] B. N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E. G. Sirer. Protection is a software issue. In *HotOS*, 1995.

[7] D. Box. *Essential COM*. Addison-Wesley, 1997.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Symp. on Code Generation and Optimization*, 2003.

[9] M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. Technical Report MSR-TR-2006-115, Microsoft Research, 2006.

[10] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symp. on Programming*, 2005.

[11] B.-Y. E. Chang, A. Chlipala, and G. C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *International Conf. on Verification, Model Checking, and Abstract Interpretation*, 2006.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security*, 2005.

[13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, 1998.

[14] R. Desikan, D. C. Burger, S. W. Keckler, and T. Austin. Sim-alpha: a validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, U.T. Austin, Dept. of C.S., 2003.

[15] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *Information Processing (Proc. of the IFIP Congress)*, 1971.

[16] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms*, 1999.

[17] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symp. on Security and Privacy*, 2000.

[18] K. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *ACM Workshop on Programming Languages and Analysis for Security*, 2006.

[19] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[20] Independent JPEG Group. Reference implementation for JPEG image compression, 1998. `http://www.ijg.org/`.

[21] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Usenix Security*, 2002.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, 1997.

[23] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, 2004.

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

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

[26] Microsoft Corp. Windows Driver Foundation (WDF), 2006. `http://www.microsoft.com/whdc/driver/wdf/`.

[27] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. *ACM Operating Systems Review, SIGOPS*, 21(5), 1987.

[28] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL*, 1998.

[29] G. C. Necula. Proof-carrying code. In *POPL*, 1997.

[30] W. Oney. *Programming the Microsoft Windows Driver Model*. Microsoft Press, second edition, 2002.

[31] M. Pietrek. A crash course on the depths of Win32 structured exception handling. *Microsoft Systems Journal*, 1997. `http://www.microsoft.com/msj/0197/`.

[32] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4), 2004.

[33] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Ottawa Linux Symp.*, 2005.

[34] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals*. Microsoft Press, fourth edition, 2004.

[35] S. Savage and B. N. Bershad. Some issues in the design of extensible operating systems. In *OSDI*, 1994.

[36] C. Small and M. I. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency: Parallel, Distributed and Mobile Computing*, 6(3), 1998.

[37] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[38] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *OSDI*, 2004.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, 2003.

[40] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *SOSP*, 2003.

[41] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[42] E. Witchel and K. Asanovic. Hardware works, software doesn't: Enforcing modularity with Mondriaan memory protection. In *HotOS*, 2003.

[43] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *ASPLOS*, 2002.

[44] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *SOSP*, 2005.