XFI: Software Guards for System Address Spaces

Ulfar Erlingsson[⋆] Martin Abadi[⋆†] Michael Vrable[‡]
Mihai Budiu[⋆] George C. Necula[§]

[⋆]Microsoft Research Silicon Valley [†]UC Santa Cruz
[‡] UC San Diego [§] UC Berkeley

Presented by:
Muli Ben-Yehuda

muli@cs.technion.ac.il

Outline

Introduction

- The problem is untrusted software
- Assume a host system, such a kernel, or a web browser
- And a plugin module, such as a driver, or a JPEG decoder
- How can we execute third party, untrusted modules safely?

Introduction

- The problem is untrusted software
- Assume a host system, such a kernel, or a web browser
- And a plugin module, such as a driver, or a JPEG decoder
- How can we execute third party, untrusted modules safely?

Introduction

- The problem is untrusted software
- Assume a host system, such a kernel, or a web browser
- And a plugin module, such as a driver, or a JPEG decoder
- How can we execute third party, untrusted modules safely?

Introduction

- The problem is untrusted software
- Assume a host system, such a kernel, or a web browser
- And a plugin module, such as a driver, or a JPEG decoder
- How can we execute third party, untrusted modules safely?

Introduction

- The problem is untrusted software
- Assume a host system, such a kernel, or a web browser
- And a plugin module, such as a driver, or a JPEG decoder
- How can we execute third party, untrusted modules safely?

Our target

- Software modules may be written in any language
- We do not have source code access
- They might be running at the highest privilege level!
- We want to protect against various attacks
- . . . without slowing things down to a crawl

Our target

- Software modules may be written in any language
- We do not have source code access
- They might be running at the highest privilege level!
- We want to protect against various attacks
- ...without slowing things down to a crawl

Our target

- Software modules may be written in any language
- We do not have source code access
- They might be running at the highest privilege level!
- We want to protect against various attacks
- ...without slowing things down to a crawl

Our target

- Software modules may be written in any language
- We do not have source code access
- They might be running at the highest privilege level!
- We want to protect against various attacks
- ...without slowing things down to a crawl

Our target

- Software modules may be written in any language
- We do not have source code access
- They might be running at the highest privilege level!
- We want to protect against various attacks
- ...without slowing things down to a crawl

Our target

- Software modules may be written in any language
- We do not have source code access
- They might be running at the highest privilege level!
- We want to protect against various attacks
- . . . without slowing things down to a crawl

Outline

What is XFI?

- XFI is a "comprehensive protection system"
- Provides both access control and integrity guarantees
- Combines static analysis with inline software guards
- Implemented on x86 Windows for both kernel and browser
- No additional hardware support required (although it would help)

What is XFI?

- XFI is a "comprehensive protection system"
- Provides both access control and integrity guarantees
- Combines static analysis with inline software guards
- Implemented on x86 Windows for both kernel and browser
- No additional hardware support required (although it would help)

What is XFI?

- XFI is a "comprehensive protection system"
- Provides both access control and integrity guarantees
- Combines static analysis with inline software guards
- Implemented on x86 Windows for both kernel and browser
- No additional hardware support required (although it would help)

What is XFI?

- XFI is a "comprehensive protection system"
- Provides both access control and integrity guarantees
- Combines static analysis with inline software guards
- Implemented on x86 Windows for both kernel and browser
- No additional hardware support required (although it would help)

What is XFI?

- XFI is a "comprehensive protection system"
- Provides both access control and integrity guarantees
- Combines static analysis with inline software guards
- Implemented on x86 Windows for both kernel and browser
- No additional hardware support required (although it would help)

What is XFI?

- XFI is a "comprehensive protection system"
- Provides both access control and integrity guarantees
- Combines static analysis with inline software guards
- Implemented on x86 Windows for both kernel and browser
- No additional hardware support required (although it would help)

How does it work

- First, we create an XFI module by binary-rewriting an existing module (e.g., rewriting a windows kernel driver)
- The rewriter adds inline software guards (we'll talk about them later)
- Then, we run a static verifier to verify that the module is safe (we'll see how this works in the next sections)
- Then we load the module and let it run

How does it work

- First, we create an XFI module by binary-rewriting an existing module (e.g., rewriting a windows kernel driver)
- The rewriter adds inline software guards (we'll talk about them later)
- Then, we run a static verifier to verify that the module is safe (we'll see how this works in the next sections)
- Then we load the module and let it run

How does it work

- First, we create an XFI module by binary-rewriting an existing module (e.g., rewriting a windows kernel driver)
- The rewriter adds inline software guards (we'll talk about them later)
- Then, we run a static verifier to verify that the module is safe (we'll see how this works in the next sections)
- Then we load the module and let it run

How does it work

- First, we create an XFI module by binary-rewriting an existing module (e.g., rewriting a windows kernel driver)
- The rewriter adds inline software guards (we'll talk about them later)
- Then, we run a static verifier to verify that the module is safe (we'll see how this works in the next sections)
- Then we load the module and let it run

How does it work

- First, we create an XFI module by binary-rewriting an existing module (e.g., rewriting a windows kernel driver)
- The rewriter adds inline software guards (we'll talk about them later)
- Then, we run a static verifier to verify that the module is safe (we'll see how this works in the next sections)
- Then we load the module and let it run

What is XFI continued

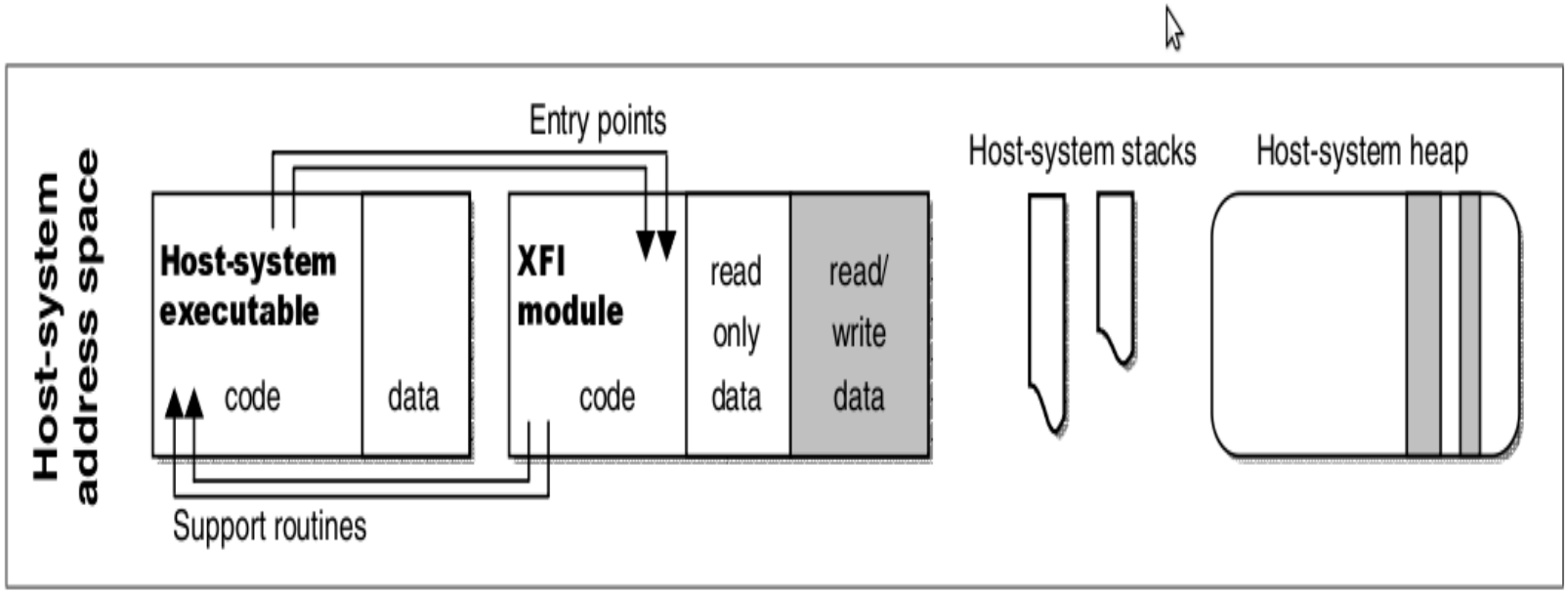

Figure 1: The address space of a host system with an XFI module. The module's external interfaces are restricted, as shown by the arrows. When the module uses a host-system stack, it is as a protected, scoped stack of virtual registers. Shaded areas are subject to arbitrary writes by the XFI module.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

- Note: controlled entry points and support routines
- Note: memory access controls
- Note: dual stacks

What is XFI continued

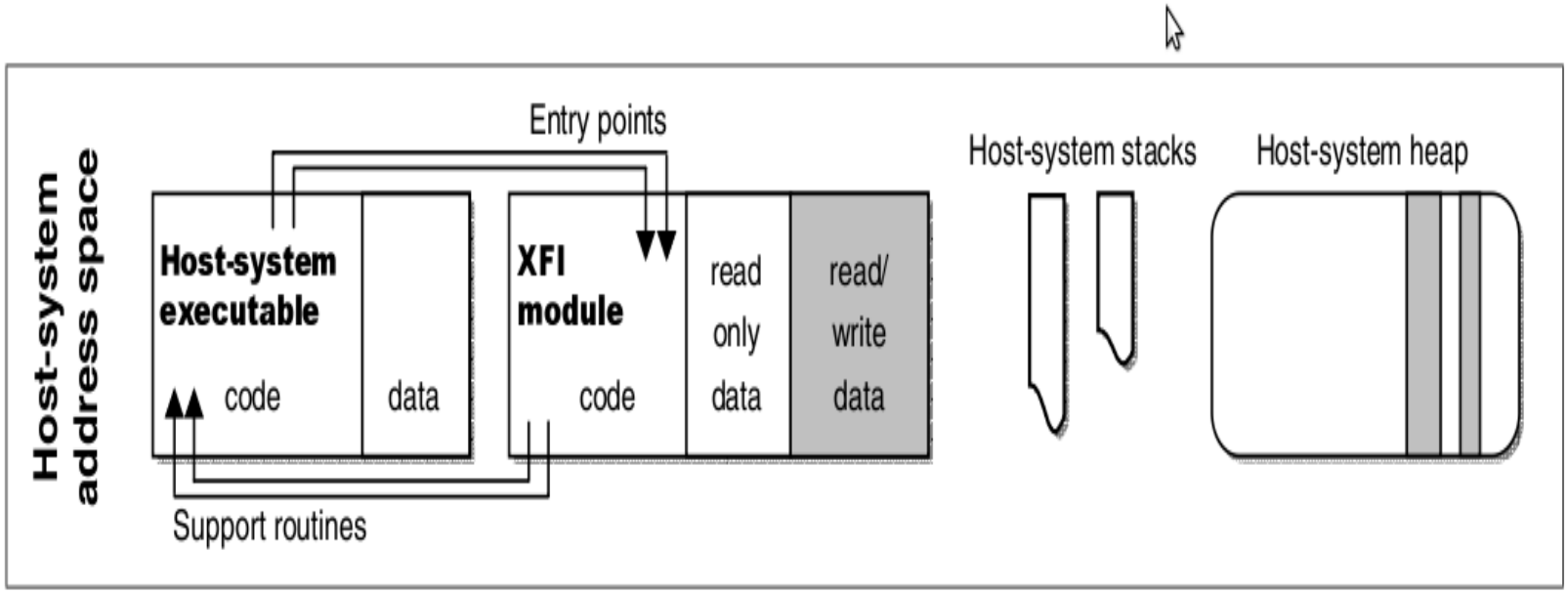

Figure 1: The address space of a host system with an XFI module. The module's external interfaces are restricted, as shown by the arrows. When the module uses a host-system stack, it is as a protected, scoped stack of virtual registers. Shaded areas are subject to arbitrary writes by the XFI module.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

- Note: controlled entry points and support routines
- Note: memory access controls
- Note: dual stacks

What is XFI continued

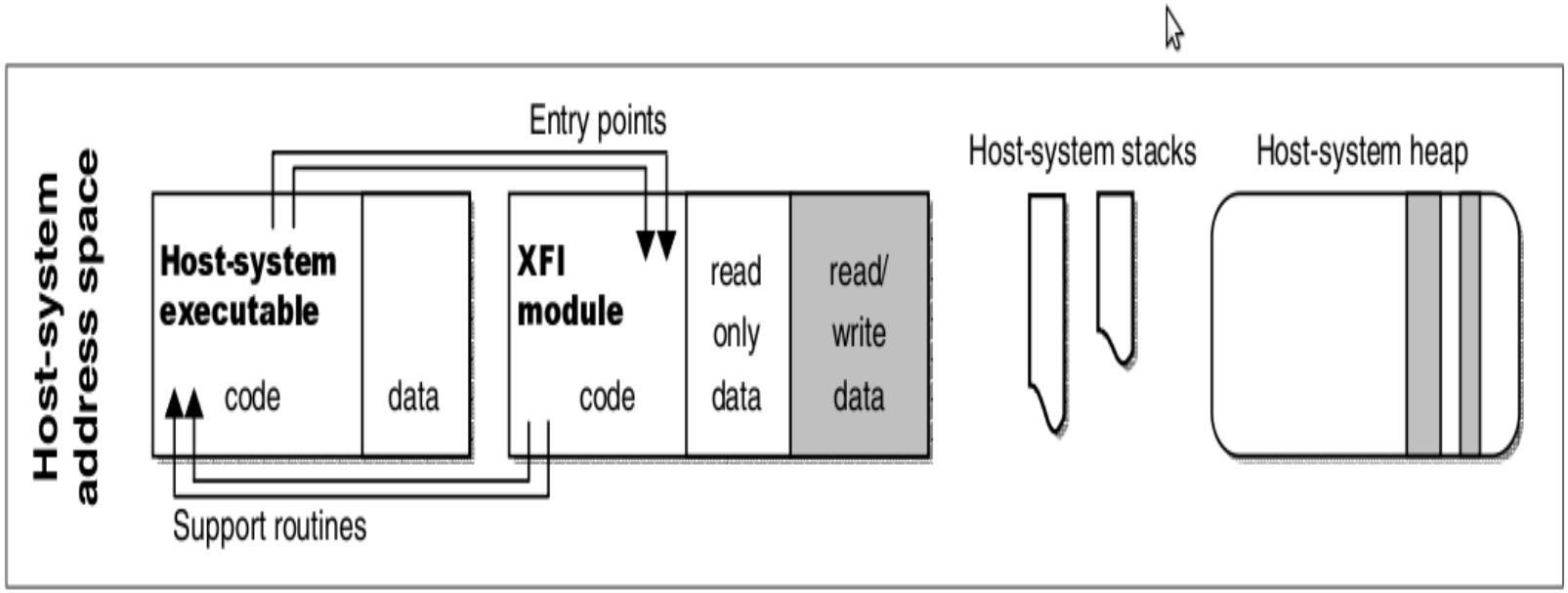

Figure 1: The address space of a host system with an XFI module. The module's external interfaces are restricted, as shown by the arrows. When the module uses a host-system stack, it is as a protected, scoped stack of virtual registers. Shaded areas are subject to arbitrary writes by the XFI module.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

- Note: controlled entry points and support routines
- Note: memory access controls
- Note: dual stacks

What is XFI continued

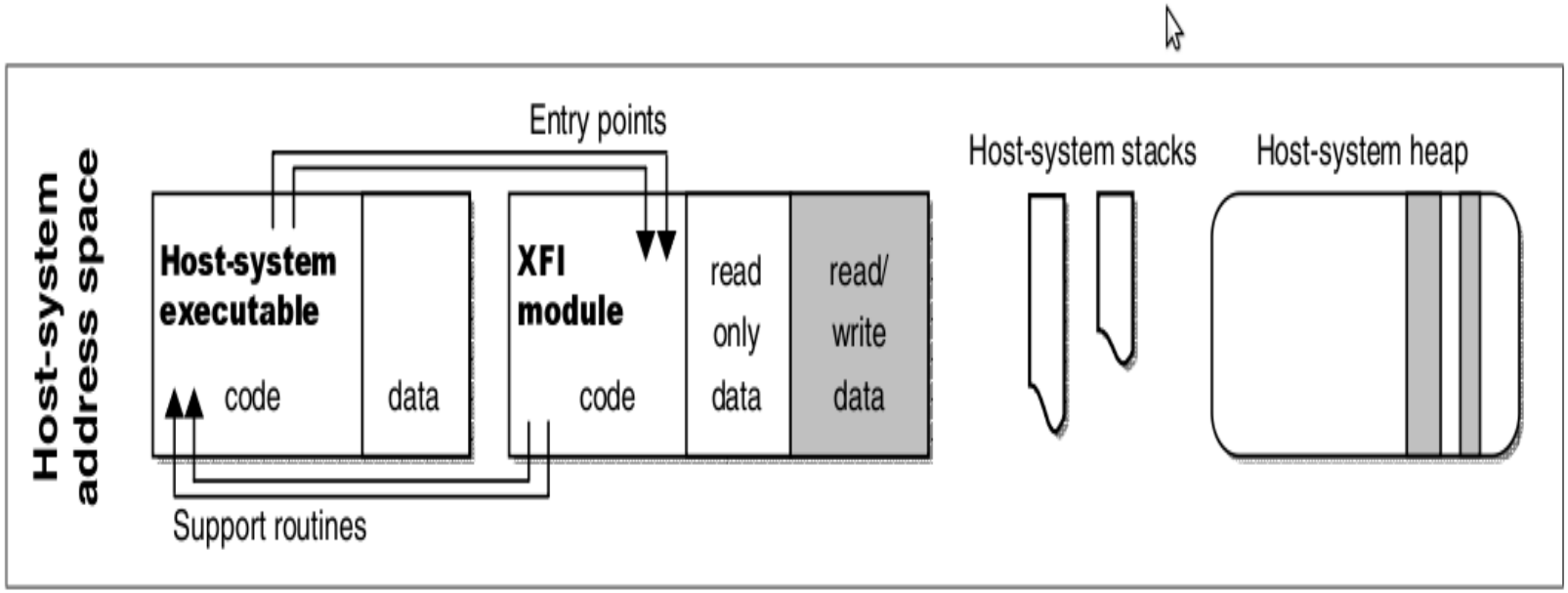

Figure 1: The address space of a host system with an XFI module. The module's external interfaces are restricted, as shown by the arrows. When the module uses a host-system stack, it is as a protected, scoped stack of virtual registers. Shaded areas are subject to arbitrary writes by the XFI module.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

- Note: controlled entry points and support routines
- Note: memory access controls
- Note: dual stacks

Outline

Constraints

For XFI to work, certain properties must hold during execution.

External properties

P1: Memory accesses are either into the memory of the XFI module or into allowed memory regions

P2: Control can never flow outside the module code except through support routines and returns to external call sites

P3: The scoped stack is always well-formed

P4: Simplified instruction semantics: problematic instructions are not allowed

P5: System-environment integrity: for example, x86 segment registers may not be modified. Particularly important for kernel-level code.

External properties

P1: Memory accesses are either into the memory of the XFI module or into allowed memory regions

P2: Control can never flow outside the module code except through support routines and returns to external call sites

P3: The scoped stack is always well-formed

P4: Simplified instruction semantics: problematic instructions are not allowed

P5: System-environment integrity: for example, x86 segment registers may not be modified. Particularly important for kernel-level code.

External properties

P1: Memory accesses are either into the memory of the XFI module or into allowed memory regions

P2: Control can never flow outside the module code except through support routines and returns to external call sites

P3: The scoped stack is always well-formed

P4: Simplified instruction semantics: problematic instructions are not allowed

P5: System-environment integrity: for example, x86 segment registers may not be modified. Particularly important for kernel-level code.

External properties

P1: Memory accesses are either into the memory of the XFI module or into allowed memory regions

P2: Control can never flow outside the module code except through support routines and returns to external call sites

P3: The scoped stack is always well-formed

P4: Simplified instruction semantics: problematic instructions are not allowed

P5: System-environment integrity: for example, x86 segment registers may not be modified. Particularly important for kernel-level code.

External properties

P1: Memory accesses are either into the memory of the XFI module or into allowed memory regions

P2: Control can never flow outside the module code except through support routines and returns to external call sites

P3: The scoped stack is always well-formed

P4: Simplified instruction semantics: problematic instructions are not allowed

P5: System-environment integrity: for example, x86 segment registers may not be modified. Particularly important for kernel-level code.

External properties

P1: Memory accesses are either into the memory of the XFI module or into allowed memory regions

P2: Control can never flow outside the module code except through support routines and returns to external call sites

P3: The scoped stack is always well-formed

P4: Simplified instruction semantics: problematic instructions are not allowed

P5: System-environment integrity: for example, x86 segment registers may not be modified. Particularly important for kernel-level code.

Internal properties

P6 Control-flow integrity: execution must follow a static, expected control-flow graph (even on computed calls and jumps). Why?

- Answer: to prevent jumps that could circumvent guards or jumping into the middle of an instruction

P7 Program-data integrity: some global and local variables can be accessed only via static references, not computed memory accesses.

Internal properties

P6 Control-flow integrity: execution must follow a static, expected control-flow graph (even on computed calls and jumps). Why?

- Answer: to prevent jumps that could circumvent guards or jumping into the middle of an instruction

P7 Program-data integrity: some global and local variables can be accessed only via static references, not computed memory accesses.

Internal properties

P6 Control-flow integrity: execution must follow a static, expected control-flow graph (even on computed calls and jumps). Why?

- Answer: to prevent jumps that could circumvent guards or jumping into the middle of an instruction

P7 Program-data integrity: some global and local variables can be accessed only via static references, not computed memory accesses.

Internal properties

P6 Control-flow integrity: execution must follow a static, expected control-flow graph (even on computed calls and jumps). Why?

- Answer: to prevent jumps that could circumvent guards or jumping into the middle of an instruction

P7 Program-data integrity: some global and local variables can be accessed only via static references, not computed memory accesses.

Outline

How do we enforce P1–P7?

- Picked mechanisms that allow for an efficient x86 implementation
- But could also be implemented on other architectures

How do we enforce P1–P7?

- Picked mechanisms that allow for an efficient x86 implementation
- But could also be implemented on other architectures

How do we enforce P1–P7?

- Picked mechanisms that allow for an efficient x86 implementation
- But could also be implemented on other architectures

Static verifier

- Correctness depends on the verifier
- Who verifies the verifier?
- Verifier establishes constraints on control flow and memory accesses
- If something cannot be verified statically (e.g., computed jump), verifier checks that the module has the necessary run-time guards
- Verifier can run offline or during load-time

Static verifier

- Correctness depends on the verifier
- Who verifies the verifier?
- Verifier establishes constraints on control flow and memory accesses
- If something cannot be verified statically (e.g., computed jump), verifier checks that the module has the necessary run-time guards
- Verifier can run offline or during load-time

Static verifier

- Correctness depends on the verifier
- Who verifies the verifier?
- Verifier establishes constraints on control flow and memory accesses
- If something cannot be verified statically (e.g., computed jump), verifier checks that the module has the necessary run-time guards
- Verifier can run offline or during load-time

Static verifier

- Correctness depends on the verifier
- Who verifies the verifier?
- Verifier establishes constraints on control flow and memory accesses
- If something cannot be verified statically (e.g., computed jump), verifier checks that the module has the necessary run-time guards
- Verifier can run offline or during load-time

Static verifier

- Correctness depends on the verifier
- Who verifies the verifier?
- Verifier establishes constraints on control flow and memory accesses
- If something cannot be verified statically (e.g., computed jump), verifier checks that the module has the necessary run-time guards
- Verifier can run offline or during load-time

Static verifier

- Correctness depends on the verifier
- Who verifies the verifier?
- Verifier establishes constraints on control flow and memory accesses
- If something cannot be verified statically (e.g., computed jump), verifier checks that the module has the necessary run-time guards
- Verifier can run offline or during load-time

Guarding computed control-flow transfers

```
    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.

- Q: Why not embed the identifier itself?
- A: So that the guard code bytes do not become a valid target

Guarding computed control-flow transfers

```
    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.

- Q: Why not embed the identifier itself?
- A: So that the guard code bytes do not become a valid target

Guarding computed control-flow transfers

```
    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.

- Q: Why not embed the identifier itself?
- A: So that the guard code bytes do not become a valid target

Guarding computed control-flow transfers

```
    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.

- Q: Why not embed the identifier itself?
- A: So that the guard code bytes do not become a valid target

Guarding the stack: two stacks

- The regular execution stack for the scoped stack
- A separate allocation stack
- Anything that might be accessed via a pointer will be on the allocation stack
- Q: Which stack are function calls and returns on?
- A: The scoped stack

Guarding the stack: two stacks

- The regular execution stack for the scoped stack
- A separate allocation stack
- Anything that might be accessed via a pointer will be on the allocation stack
- Q: Which stack are function calls and returns on?
- A: The scoped stack

Guarding the stack: two stacks

- The regular execution stack for the scoped stack
- A separate allocation stack
- Anything that might be accessed via a pointer will be on the allocation stack
- Q: Which stack are function calls and returns on?
- A: The scoped stack

Guarding the stack: two stacks

- The regular execution stack for the scoped stack
- A separate allocation stack
- Anything that might be accessed via a pointer will be on the allocation stack
- Q: Which stack are function calls and returns on?
- A: The scoped stack

Guarding the stack: two stacks

- The regular execution stack for the scoped stack
- A separate allocation stack
- Anything that might be accessed via a pointer will be on the allocation stack
- Q: Which stack are function calls and returns on?
- A: The scoped stack

Guarding the stack: two stacks

- The regular execution stack for the scoped stack
- A separate allocation stack
- Anything that might be accessed via a pointer will be on the allocation stack
- Q: Which stack are function calls and returns on?
- A: The scoped stack

Guarding computed memory accesses

```
      # 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.

- Q: Where do A, B, L, and H come from?
- A: Common case: accessible range established by the loader.

Guarding computed memory accesses

```
     # 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.

- Q: Where do A, B, L, and H come from?
- A: Common case: accessible range established by the loader.

Guarding computed memory accesses

```
     # 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.

- Q: Where do A, B, L, and H come from?
- A: Common case: accessible range established by the loader.

Guarding computed memory accesses

```
      # 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.

- Q: Where do A, B, L, and H come from?
- A: Common case: accessible range established by the loader.

Outline

Implementation bits and bytes

- XFI modules: EXEs or DLLs with multiple sections
- The rewriter: take x86 assembly with debug information, outputs XFI module with inline guards
- The verifier: 3K lines of C++ code, independent from the rewriter. Single linear pass over module code, one basic block at the time. Verifies that P1–P7 hold (either statically or that sufficient guards exist for them to hold during runtime)
- Inline guards: chosen for architectural efficiency, exploit host properties (e.g., Windows holds the bottom of the current stack in `fs`)
- Host system support: memory guard slowpath permission tables, allocation-stack manager, software call gates, copying vs. granting access to parameters, . . .

Implementation bits and bytes

- XFI modules: EXEs or DLLs with multiple sections
- The rewriter: take x86 assembly with debug information, outputs XFI module with inline guards
- The verifier: 3K lines of C++ code, independent from the rewriter. Single linear pass over module code, one basic block at the time. Verifies that P1–P7 hold (either statically or that sufficient guards exist for them to hold during runtime)
- Inline guards: chosen for architectural efficiency, exploit host properties (e.g., Windows holds the bottom of the current stack in `fs`)
- Host system support: memory guard slowpath permission tables, allocation-stack manager, software call gates, copying vs. granting access to parameters, . . .

Implementation bits and bytes

- XFI modules: EXEs or DLLs with multiple sections
- The rewriter: take x86 assembly with debug information, outputs XFI module with inline guards
- The verifier: 3K lines of C++ code, independent from the rewriter. Single linear pass over module code, one basic block at the time. Verifies that P1–P7 hold (either statically or that sufficient guards exist for them to hold during runtime)
- Inline guards: chosen for architectural efficiency, exploit host properties (e.g., Windows holds the bottom of the current stack in `fs`)
- Host system support: memory guard slowpath permission tables, allocation-stack manager, software call gates, copying vs. granting access to parameters, . . .

Implementation bits and bytes

- XFI modules: EXEs or DLLs with multiple sections
- The rewriter: take x86 assembly with debug information, outputs XFI module with inline guards
- The verifier: 3K lines of C++ code, independent from the rewriter. Single linear pass over module code, one basic block at the time. Verifies that P1–P7 hold (either statically or that sufficient guards exist for them to hold during runtime)
- Inline guards: chosen for architectural efficiency, exploit host properties (e.g., Windows holds the bottom of the current stack in `fs`)
- Host system support: memory guard slowpath permission tables, allocation-stack manager, software call gates, copying vs. granting access to parameters, . . .

Implementation bits and bytes

- XFI modules: EXEs or DLLs with multiple sections
- The rewriter: take x86 assembly with debug information, outputs XFI module with inline guards
- The verifier: 3K lines of C++ code, independent from the rewriter. Single linear pass over module code, one basic block at the time. Verifies that P1–P7 hold (either statically or that sufficient guards exist for them to hold during runtime)
- Inline guards: chosen for architectural efficiency, exploit host properties (e.g., Windows holds the bottom of the current stack in `fs`)
- Host system support: memory guard slowpath permission tables, allocation-stack manager, software call gates, copying vs. granting access to parameters, . . .

Implementation bits and bytes

- XFI modules: EXEs or DLLs with multiple sections
- The rewriter: take x86 assembly with debug information, outputs XFI module with inline guards
- The verifier: 3K lines of C++ code, independent from the rewriter. Single linear pass over module code, one basic block at the time. Verifies that P1–P7 hold (either statically or that sufficient guards exist for them to hold during runtime)
- Inline guards: chosen for architectural efficiency, exploit host properties (e.g., Windows holds the bottom of the current stack in `fs`)
- Host system support: memory guard slowpath permission tables, allocation-stack manager, software call gates, copying vs. granting access to parameters, . . .

Architectural support

- Evaluated arch support for CFI and memory guards on Alpha simulator
- CFI: `cfilabel` and indirect jump/return/jump to subroutine
- Must execute `cfilabel` after transfer instruction with the right identifier ID
- Memory guard: `mrguard(Reg, L, H)` computes R-L and R+H and compares with A and B. Exception if compare fails.
- Implementation is efficient: replacing with NOPs gives same results!

Architectural support

- Evaluated arch support for CFI and memory guards on Alpha simulator
- CFI: `cfilabel` and indirect jump/return/jump to subroutine
- Must execute `cfilabel` after transfer instruction with the right identifier ID
- Memory guard: `mrguard(Reg, L, H)` computes R-L and R+H and compares with A and B. Exception if compare fails.
- Implementation is efficient: replacing with NOPs gives same results!

Architectural support

- Evaluated arch support for CFI and memory guards on Alpha simulator
- CFI: `cfilabel` and indirect jump/return/jump to subroutine
- Must execute `cfilabel` after transfer instruction with the right identifier ID
- Memory guard: `mrguard(Reg, L, H)` computes R-L and R+H and compares with A and B. Exception if compare fails.
- Implementation is efficient: replacing with NOPs gives same results!

Architectural support

- Evaluated arch support for CFI and memory guards on Alpha simulator
- CFI: `cfilabel` and indirect jump/return/jump to subroutine
- Must execute `cfilabel` after transfer instruction with the right identifier ID
- Memory guard: `mrguard(Reg, L, H)` computes R-L and R+H and compares with A and B. Exception if compare fails.
- Implementation is efficient: replacing with NOPs gives same results!

Architectural support

- Evaluated arch support for CFI and memory guards on Alpha simulator
- CFI: `cfilabel` and indirect jump/return/jump to subroutine
- Must execute `cfilabel` after transfer instruction with the right identifier ID
- Memory guard: `mrguard(Reg, L, H)` computes R-L and R+H and compares with A and B. Exception if compare fails.
- Implementation is efficient: replacing with NOPs gives same results!

Architectural support

- Evaluated arch support for CFI and memory guards on Alpha simulator
- CFI: `cfilabel` and indirect jump/return/jump to subroutine
- Must execute `cfilabel` after transfer instruction with the right identifier ID
- Memory guard: `mrguard(Reg, L, H)` computes R-L and R+H and compares with A and B. Exception if compare fails.
- Implementation is efficient: replacing with NOPs gives same results!

Outline

Applications

- Windows kernel drivers
- Internet Explorer browser plugins
- Protect against:
  - Spurious writes to memory
  - Heap overflows
  - CFI guards protect against Blaster, Slammer, others
- Does not protect against Nimda-like attacks that abuse over-permissive support routines

Applications

- Windows kernel drivers
- Internet Explorer browser plugins
- Protect against:
  - Spurious writes to memory
  - Heap overflows
  - CFI guards protect against Blaster, Slammer, others
- Does not protect against Nimda-like attacks that abuse over-permissive support routines

Applications

- Windows kernel drivers
- Internet Explorer browser plugins
- Protect against:
  - Spurious writes to memory
  - Heap overflows
  - CFI guards protect against Blaster, Slammer, others
- Does not protect against Nimda-like attacks that abuse over-permissive support routines

Applications

- Windows kernel drivers
- Internet Explorer browser plugins
- Protect against:
  - Spurious writes to memory
  - Heap overflows
  - CFI guards protect against Blaster, Slammer, others
- Does not protect against Nimda-like attacks that abuse over-permissive support routines

Applications

- Windows kernel drivers
- Internet Explorer browser plugins
- Protect against:
  - Spurious writes to memory
  - Heap overflows
  - CFI guards protect against Blaster, Slammer, others
- Does not protect against Nimda-like attacks that abuse over-permissive support routines

Applications

- Windows kernel drivers
- Internet Explorer browser plugins
- Protect against:
  - Spurious writes to memory
  - Heap overflows
  - CFI guards protect against Blaster, Slammer, others
- Does not protect against Nimda-like attacks that abuse over-permissive support routines

Applications

- Windows kernel drivers
- Internet Explorer browser plugins
- Protect against:
  - Spurious writes to memory
  - Heap overflows
  - CFI guards protect against Blaster, Slammer, others
- Does not protect against Nimda-like attacks that abuse over-permissive support routines

Applications

- Windows kernel drivers
- Internet Explorer browser plugins
- Protect against:
  - Spurious writes to memory
  - Heap overflows
  - CFI guards protect against Blaster, Slammer, others
- Does not protect against Nimda-like attacks that abuse over-permissive support routines

Enforcement overhead: SFI and WDF

		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.

Enforcement overhead: JPEG and Mediabench

	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.

Outline

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

My conclusions

- The good:
  - Works for unmodified x86 kernel code
  - Appears to work :-)
- The bad:
  - Lack of formal proof for verifier is worrisome
  - As is lack of formal security model
  - Does not protect against: host-support attacks, multi-thread attacks, device-based attacks, ...?
  - Evaluation is weak; Benchmarks are limited. How would it perform if protecting all drivers in a given kernel?
  - Superseded(?) by “Fast Byte-Granularity Software Fault Isolation”, SOSP ’09

Questions?