Using LLVM to guarantee program integrity

Simon Cook

Background

- Compiling for security is becoming increasingly important
  - Finding bugs through AddressSanitizer, MemorySanitizer, etc.
  - Research programs such as LADA

- Use of security-enhancing hardware can added to existing programs by extending their use in the compiler

Topics to Discuss

- Hardware
- C attributes
- Clang/Sema, Clang/Codegen
- LLVM Optimization Tweaks
- Instruction Lowering/Selection
- AsmPrinting
- Creating post-link tools using MC

What are we trying to protect?

- Instruction integrity
  - Detection of any modification to program code at runtime
- Control flow integrity
  - Ensuring that calls/branches only go to known locations and that return values are correct
- If either of these are invalid the hardware should trap as soon as possible

Encoding Instructions: Hardware

Each instruction becomes dependent on the previous one

Given an instruction I_1, and internal state S_0, we can produce the encoded instruction E_1 and output state S_1

`add r0, r1` $encode\,(I_1, S_0) \rightarrow (E_1, S_1)$ `0xbeef`

At run time, the hardware can use the same state, and using the encoded instruction, reproduce the original instruction

`0xbeef` $decode\,(E_1, S_0) \rightarrow (I_1, S_1)$ `add r0, r1`

Encoding a Function

```
int foo(int x, int y) { return (4*x) + (y&5); }
```

lsli	$r10, $r2, 2	I_1	919a 4000
andi	$r13, $r3, 5	I_2	5d87 4002
add	$r2, $r13, $r10	I_3	aa82 0900
jmp	$r0	I_4	0050

lsli	$e(I_1, S_0) \rightarrow$	E_1	0001 0203
andi	$e(I_2, S_1) \rightarrow$	E_2	0405 0607
add	$e(I_3, S_2) \rightarrow$	E_3	0809 0a0b
jmp	$e(I_4, S_3) \rightarrow$	E_4	0c0d

Encoding Branches

```
int foo(int x, int y, bool z) { return z ? x : y; }
```

```
; BB#0:
    movi    $r10, 0                 I1  809e 4000
    bne     .LBB0_2, $r4, $r10      I2  e2c6 0100
; BB#1:
    mov     $r2, $r3                I3  9812
.LBB0_2:
    jmp     $r0                     I4  0050
```

$$e\,(\,I_4\,,\,S_3\,) \rightarrow E_4$$
$$e\,(\,I_4\,,\,S_2\,) \rightarrow E_4$$

For two cases, this may be solvable, but not for blocks with many direct predecessors

Encoding Branches

```
int foo(int x, int y, bool z) { return z ? x : y; }
```

```
; BB#0:
    movi    $r10, 0                  I1  809e 4000
    bne     .LBB0_2, $r4, $r10       I2  e2c6 0100
    _correction_value_               C   ....
; BB#1:
    mov     $r2, $r3                 I3  9812
.LBB0_2:
    jmp     $r0                      I4  0050
```

$$e(I_4, S_3) \rightarrow E_4 \qquad e(I_2, S_1) \rightarrow E_2$$

$$e(I_4, C) \rightarrow E_4 \qquad e(C, S_1) \rightarrow E_C$$

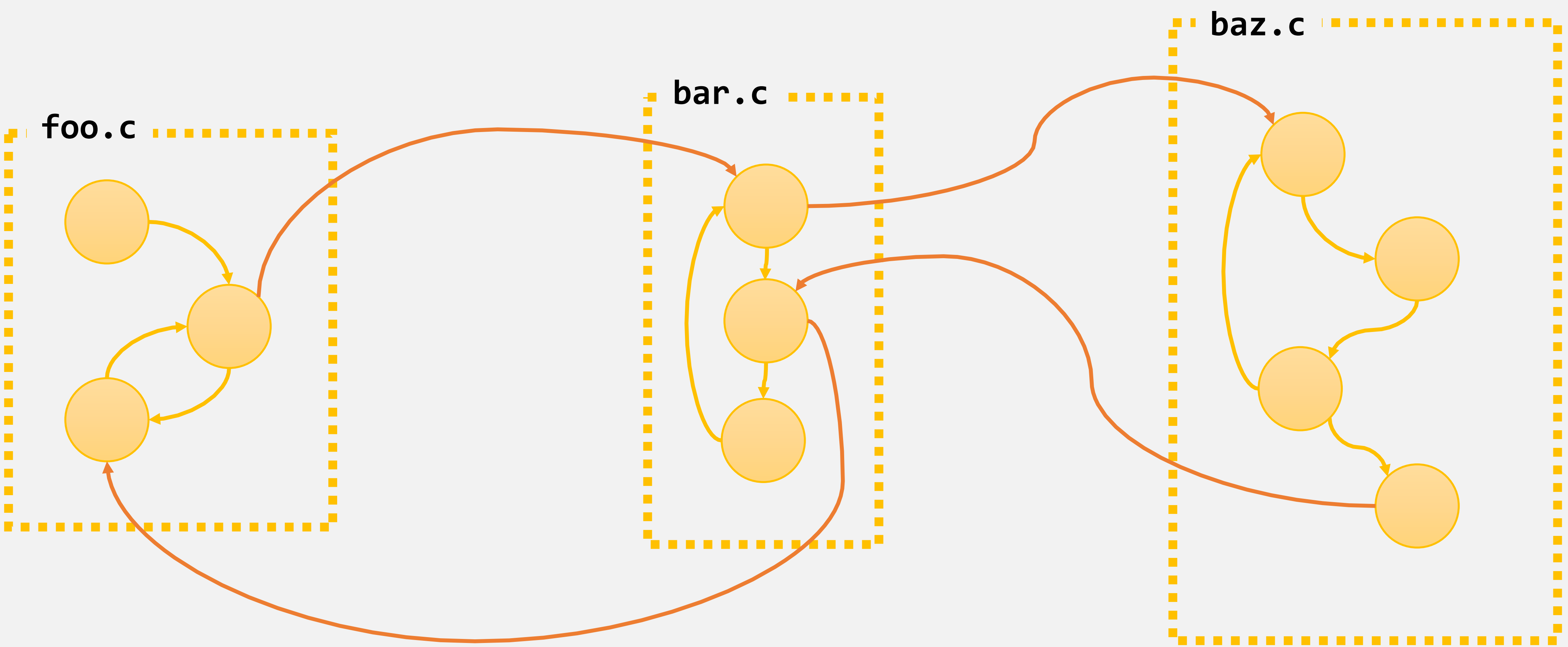
Function Calls

```
int foo(int x) { return bar(x+2); }
```

subi	$r1, $r1, 2	I_1	4a16
stw	[$r1, 0], $r0	I_2	4038
addi	$r2, $r2, 2	I_3	9214
bal	bar, $r0	I_4	00c2 0000
ldw	$r0, [$r1, 0]	I_5	0828
addi	$r1, $r1, 2	I_6	4a14
jmp	$r0	I_7	0050

- Calling `bar` pushes state S_4 to the encoding stack
- Returning pops this value, so calls can be treated as part of same BB

Scaling up to an entire program



Clang: `-mencode-instructions`?

Pros

- Easy to enable, one flag enables system for entire CU

Cons

- ABI break, flag required across entire project
- Only affects C, assembly still needs patching
- Potential concerns about code size

In the end we decided not to go down this route

Clang: `__attribute__((protected))`

Pros

- Per function granularity
- Lower cost overhead for "non-secure" functions
- ABI change is limited to those functions it was requested for

Cons

- Only affects C, assembly still needs patching
- Risk of user neglecting to add attribute to all declarations of a function

Clang Function Attribute

- Added as a TypeAttr
  - We want to add error checking as pointers to protected functions are not the same as to unprotected
- Extend FunctionType to support having protected as a property
- For calls, add protected as bit in ExtInfo
- This is not the same as a different calling convention, as we use different CCs and want to turn this on independently
- For CodeGen, we map this down to a LLVM function attribute "protected"

`int (*__attribute__((protected)))()`

- Function pointers present a challenge
  - We need to know what S_0 the target function is expecting
  - If S_0 based on address of function, we have no problem…
  - … otherwise we need to calculate it
- Could use same for each function? Defeats security benefits.
- Calculate all possible call targets? Not necessarily possible.
- User should know, let's ask them!
  - Attribute becomes `__attribute__((protected("somestring")))`

Changes to Middle-End LLVM

- None, really…
- … except one small change to the inliner
  - Avoid inlining secure functions into non-secure
  - Merging non-secure into secure is generally safe

- Update call target nodes with custom flag field

```
let isCall = 1 in
  def JAL : Inst_rrr <0x2, 0x9, (outs),
                      (ins i64imm:$flags, GR64:$rD, GR64:$rB),
                      "jal\t $rD, $rB",
                      [(AAPcall timm:$flags, GR64:$rD, GR64:$rB)]>;
```

- Flag field contains:
  - Bit indicating whether function expects security
  - 16-bit representation of group name

Encoding Control Flow I

- Just before emission, SecurityAnalysisPass:
  - Prepares a function for annotation
  - Builds lists of branches/calls/jump tables
  - Adds placeholders for correction values
  - Generates report on code size impact

```
===--- CF encoding statistics for 'main' ---===
                   Bytes added: 10
                   Words added: 5
                NOP gaps added: 3
    Enable/Disable insns added: 1
```

.debug_secure Record Format

- Start function:

1	Function Start Address	Group

- End function:

2	Function End Address

- Direct Call:

6	Call Site	Call Target

- Jump Table:

11	Count	Target 1	Target 2

Encoding Control Flow II

- AsmPrinterHandler – Adds hooks to assembly printing
  - Used by us for adding labels/emitting encoding at end of module
  - beginInstruction
  - endInstruction
  - beginFunction
  - endFunction
  - endModule

Resolving Values

1. Reconstruct the control flow graph of all secure functions
2. Assign correction values/S_0 to all functions/groups
3. Encode each basic block, noting state of each reloc
4. Validate all values are known
5. Fill in relocations
6. Writeback

End result

```
simon@shadowfax$ llvm-objdump -d a.out

a.out:  file format ELF32-aap

Disassembly of section .text:
Section has correction values, printing real instructions
foo:
 8000000:       [8f39] 91 9a 40 00     lsli    $r10, $r2, 2
 8000004:       [81ca] 5d 87 40 02     andi    $r13, $r3, 5
 8000008:       [053b] aa 82 09 00     add     $r2, $r13, $r10
 800000c:       [93e4] 00 50           jmp     $r0
```

Thank you