Hardware-Assisted Fine-Grained CFI
Towards Efficient Protection of Embedded Systems Against Software Exploitation

Truls Asheim
December 5, 2016
Agenda

Introduction

Principles of Operation

Implementation Details

Implementation Results

Conclusions
Introduction
Motivation of CFI

- Eliminating attack vectors enabled by the circumvention of the prescribed control flow of programs
- These attacks are becoming increasingly common as “traditional” attacks are becoming harder to execute.
- An example of such attack is ROP
- The actual attack mechanisms has been explained previous weeks so we won’t go into details here
- This presentation a compilation of the paper by Davi et. al. titled the same as this presentation and HAFIX: Hardware-Assisted Flow Integrity Extension by Davi. et. al. We mostly focus on the first, but the latter provides implementation details.
Feasibility of implementations

- Performance is the big challenge
- Coarse grained CFI has a limited performance penalty, but is insufficient to protect against ROP attacks.
- Calculating static pre-determined Control Flow Graphs not possible for complex programs.
- Pure software approaches to Fine Grained CFI has severe performance penalties
Changing device landscapes

Changing vulnerability landscapes.

- Relative and absolute increase in Embedded devices (Internet of Things).
- Central in increasing number of attacks. Recently a 1.1 Tbps DDoS was launched from security cameras.
- Performance penalty of Fine Grained CFI even more unacceptable for embedded systems and may render them infeasible.
Shift of strategy

- Moving implementation from software to hardware
- Common strategy for eliminating the overhead of operations that are excessively expensive to implement in software
- Specifically in this case, move the CFI instrumentation into the CPU by adding CPU instructions enforcing it
- Enables enforcing Fine Grained CFI while significantly reducing the performance overhead associated with purely software based implementations.
Threat Model

• Software running on an embedded system that contains exploitable memory errors such as stack or heap overflows.

• We assume that direct code injection attacks are infeasible due to non-executable memory policies.

• We assume that an attacker can bypass ASLR (Address Space Layout Randomization) which is vulnerable to several attacks.
Principles of Operation
Confining function returns to active call sites, or, forcing returns to call-preceded instructions.
Stateful program execution

• Adding special instructions for enforcing the prescribed control flow graph of a program.
• Stateful execution: Only allow certain instructions in certain states.
• Enforce that functions have to start with a label setting function prologues and end with label checking function epilogues.
Still something missing...

- Indirect jumps, present in any sort of jump-tables such as C++ vtable, etc... still not covered.
- Prevalent in a lot of software
- Gadgets ending in an indirect jump can be used in a ROP attack
- Enforced through a in-software heuristic
Indirect jumps proposed to be handled through a heuristic which enforces

- Uses a sliding window of 5 indirect jumps
- Counts direct to indirect jump ratio: direct jumps usually kill a ROP chain but are common in real-world code
- Counts stack **push** to **pop** ratio. Stack pushes will overwrite part of a ROP chain, so many **pop** instructions executed are indicative of a ROP attack.
Implementation Details
Restricting the validity of instructions based on processor states.

**State 0: Normal execution.** “Normal” instructions allowed.

**State 1: Function entry.** First instruction after a `call`. Only `cfibr` allowed.

**State 2: Function return.** First instruction following a call-site (first function executed after return). Only `cfiret` allowed and must refer to an active label.

**State 3: Stop execution.** Entered if above policies are violated.
# Instruction semantics

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFIBASE baseaddr</td>
<td>$\text{cfi_base_reg} := \text{baseaddr}$</td>
</tr>
<tr>
<td>CFIBR label</td>
<td>$[\text{cfi_base_reg} + \text{label}] := 1$</td>
</tr>
<tr>
<td>CFIRET label</td>
<td>$\text{if} [\text{cfi_base_reg} + \text{label}] \neq 1 \text{ then STOP}$</td>
</tr>
<tr>
<td>RET label</td>
<td>$[\text{cfi_base_reg} + \text{label}] := 0$</td>
</tr>
<tr>
<td>DEACT label</td>
<td>$[\text{cfi_base_reg} + \text{label}] := 0$</td>
</tr>
</tbody>
</table>

### Table 1: Instruction extensions and semantics

Note that the label parameters of instructions are not “labels” in the sense that a unique label is defined, but rather, they point into an array indicating if a given label ID is available or not.
Figure 1: Design of Hardware-Enforced ROP detection
An Example

Program Code

Function A
- CFIBR 0025
- Instruction 1
- Instruction 2
- CALL Function B
- CFIRET 0025
- Instruction 3
- RET 0025

Function B
- CFIBR 0050
- Instruction 4
- Instruction 5
- RET 0050

Function C
- CFIBR 0272
- CALL Function X
- CFIRET 0272
- RET 0272

Figure 2: CFI policy for function returns
Another Example

<funct_a>
1: cfibr 0x15
2: push %ebp
3: mov %ebp, %esp
4: call <funct_b>
5: cfiret 0x15
6: mov %esp, %ebp
7: pop %ebp
8: cfidel 0x15
9: ret

<funct_b>
a: cfibr 0x16
b: push %ebp
c: mov %ebp, %esp
d: asm_ins...
e: mov %esp %ebp
f: pop %ebp
10: cfidel 0x16
11: ret
Implementation Results
Hardware Implementations

Implemented as a CPU instruction set expansion called HAIFAX

Implemented for two CPU architectures:

- x86 on the Intel Siskiyou Peak 32-bit research architectures
- SPARC on the LEON 3 architecture
- Label spaces are sized respectively $16384 \times 1$ and $1024 \times 13$.

Both are syntehsizable hardware architectures tested on a FPGA
Compilers has been modified in order to add the CFI instrumentation

- For SPARC: the Aeroflex Gaisler Baremetal C Compiler (BCC)
- For x86: Modification of the modified LLVM toolchain shipped with Siskiyou Peak
Performance Analysis

Figure 3: Performance of HAIFAX implementation
Improved protection

- Gadget space reduced to 19.82%
- Performance overhead 2% on average
- Significant reduction in gadget space for a very modest increase in performance overhead.
- Pure return-into-libc not prevented, most attacks requires at list some ROP gadgets
- Reduction in gadget space sufficient to markedly reduce the change of an attack being successful?
Conclusions
Conclusions

• Successful demonstration of a low-overhead implementation of a fine-grained CFI policy protecting control graph backwards edges.
• Minimal performance penalty of usage
• Cannot be readily used: Requires changes of the executing hardware architectures
• Also changes in software:
  • Compilers needs to be changed to emit the CFI instrumentation instructions.
  • can possibly also be handled using binary rewriting.
Future work

• Adding operating system support for per-process label-spaces

• Resolving issues related to the label space of programs using several different libraries (emit unique label globally across entire program including libraries)

• Add support for HAIFAX to JIT-compilers.
Thank You!

Questions?