High-Level Languages for Safety/Security

- Java, C#, Haskell, F*...
- JavaScript for web applications

Benefits
- Better support for safety and security
- Portability
- Better programming abstractions
- ...

So why bother enforcing security at the binary level?
Why Binary-Level Software Security?

- **Programming language agnostic**
  - Eventually all software is turned into native code
  - Apply to all languages: C, C++, OCaml, assembly ...
  - Accommodate **legacy code/libraries** written in C/C++
    - E.g., zlib, codec, image libraries (JPEG), fast FFT libraries ...
  - Apply to applications that are developed in multiple languages
    - Native code is an unifying representation
Why Binary-Level Software Security?

- Low-level languages (i.e. C/C++) have better **Performance**
  - Compilers for high-level languages still not as good as you might hope
  - Example: Box2D physics engine for games (C++)
    - Java: 3x slowdown
    - Javascript V8: 15-25x slowdown
C vs. Java vs. JavaScript Speed Comparison

Source: The Computer Language Benchmarks Game
Why Binary-Level Software Security?

- **Buggy compilers** and language runtimes
  - May invalidate the guarantees provided by source-level techniques
  - Example [Howard 2002]:

  ```
  ...  
  memset(password, 0, len); // zeroing out the password
  ... // password never used again
  ```

  
- Csmith discovered 325 compiler bugs [Yang et al. PLDI 2011]
Yet the Binary Level is Challenging

- High-level abstractions disappear
  - No notion of variables, classes, objects, functions, ...
  - Relevant concepts: registers, memory, ...

- Security policies can use only low-level concepts
  - E.g., can’t use pre- and post-conditions of functions
  - Semantic gap between what’s expressible at high level and at low level
Challenges at the Binary Level

- No guarantee of basic safety
  - Lack of control-flow graph: a computed jump can jump to any byte offset
    - Enable return-oriented programming (ROP)
  - A memory op can access any memory in the address space
    - Modifiable code
  - Can invoke OS syscalls to cause damages

Much harder to perform analysis and enforce security at the binary level
Two Extremes of Dealing With Native Code

- Allow native code
  - With some code-signing mechanism
  - Examples: Microsoft ActiveX controls; browser plug-ins
- Disallow native code
  - By default, Java applet cannot include native libraries
Approaches for Obtaining Safe Native Code

- Certifying compilers
  - Proof-carrying code (PCC) [Necula & Lee 1996]
  - Typed assembly languages (TAL) [Morrisett et al. 1999]
  - ...
  - However, producing proofs (annotations) in code is nontrivial

- Certified compilers: proving compiler correctness
  - CompCert [Leroy POPL 06]

- An alternative approach: use reference monitors to implement a sandbox in which to execute the native code
Reference Monitors
Reference Monitor

- Observe the execution of a program and halt the program if it’s going to violate the security policy.
Common Examples of RM

- Operating system: syscall interface
- Interpreters, language virtual machines, software-based fault isolation
- Firewalls
- ...
- Claim: majority of today’s enforcement mechanisms are instances of reference monitors.
What Policies Can be Enforced?

- Some liberal assumptions:
  - Monitor can have infinite state
  - Monitor can have access to entire history of computation
  - But monitor can’t guess the future – the predicate it uses to determine whether to halt a program must be computable

- Under these assumptions:
  - There is a nice class of policies that reference monitors can enforce: safety properties
  - There are desirable policies that no reference monitor can enforce precisely
Classification of Policies

- “Enforceable Security Policies” [Schneider 00]

Diagram:
- Security policies
  - Security properties
    - Safety properties
    - Liveness properties
A system is modeled as traces of system events
- E.g., A trace of memory operations (reads and writes)
  - Events: read(addr); write(addr, v)

A security policy: a predicate on sets of allowable traces

A security policy is a property if its predicate specifies whether an individual trace is legal
- E.g., a trace is legal if all its memory access is within address range [1,1000]
What is a Non-Property?

- A policy that may depend on multiple execution traces
- Information flow polices
  - Sensitive information should not flow to unauthorized person implicitly
  - Example: a system protected by passwords
    - Suppose the password checking time correlates closely to the length of the prefix that matches the true password
    - Then there is a timing channel
    - To rule this out, a policy should say: no matter what the input is, the password checking time should be the same in all traces
Safety and Liveness Properties [Alpern & Schneider 85,87]

- **Safety**: Some “bad thing” doesn’t happen.
  - Proscribes traces that contain some “bad” prefix
  - Example: the program won’t read memory outside of range [1,1000]

- **Liveness**: Some “good thing” does happen
  - Example: program will terminate
  - Example: program will eventually release the lock

- **Theorem**: Every security property is the conjunction of a safety property and a liveness property
Policies Enforceable by Reference Monitors

- Reference monitor **can** enforce any safety property
  - Intuitively, the monitor can inspect the history of computation and prevent bad things from happening
- Reference monitor **cannot** enforce liveness properties
  - The monitor cannot predict the future of computation
- Reference monitor **cannot** enforce non-properties
  - The monitor inspects one trace at a time
Inlined Reference Monitors (IRM)
Reference Monitor, Inlined

- Lower performance overhead
  - Enforcement doesn’t require context switches
- Policies can depend on application semantics
- Environment independent---portable

Integrate reference monitor into program code

Program being monitored

RM
IRM via Program Rewriting

- The rewritten program should satisfy the desired security policy

- Examples:
  - Source-code level
    - CCured [Necula et al. 02]
    - [Ganapathy Jaeger Jha 06, 07]
  - Java bytecode-level rewriting: PoET [Erlingsson and Schneider 99]; Naccio [Evans and Twyman 99]
This Lecture: Binary-Level IRM

- Software-based Fault Isolation (SFI)
- Control-Flow Integrity (CFI)
- Data-Flow Integrity (DFI)
  - [Castro et al. 06]
- Fine-grained data integrity and confidentiality
  - Protecting small buffers
    - [Castro et al. SOSP 09]; [Akritidis et al. Security 09]
- ...
Enforceable Policies via IRM

- Clearly, it can enforce any safety property
- Surprisingly, it goes beyond safety properties [Hamlen et al. TOPLAS 2006]
  - Intuition: the rewriter can statically analyze all possible executions of programs and rewrite accordingly
  - Timing channels could be removed [Agat POPL 2000]
Verifier: checking the reference monitor is inlined correctly (so that the proper policy is enforced)

Benefit: no need to trust the RM-insertion phase
Software-Based Fault Isolation (SFI)
Software-Based Fault Isolation (SFI)

- Originally proposed for MISP [Wahbe et al. SOSP 93]
  - PittSFIeld [McCamant & Morrisett 06] extended it to x86
- Use an IRM to isolate components into “logical” address spaces in a process
  - Conceptually: check each read, write, & jump to make sure it’s within the component’s logical address space
SFI Policy

Fault Domain

- CB
- CL
- DB
- DL

1) All jumps remain in CR
2) Reference monitor not bypassed by jumps

Code Region (readable, executable)

Data Region (readable, writable)

All R/W remain in DR [DB, DL]
Enforcing SFI Policy

- Insert monitor code into the target program before unsafe instructions (reads, writes, jumps, ...)

```
[r3+12] := r4 //unsafe mem write
```

```
r10 := r3 + 12
if r10 < DB then goto error
if r10 > DL then goto error
[r10] := r4
```
Optimizations for Better Performance

- Naïve SFI is OK for security
  - But the runtime overhead is too high
- Performance can be improved through a set of optimizations
Optimization: Special Address Pattern

- Both code and data regions form contiguous segments
  - Upper bits are all the same and form a region ID
  - Address validity checking: only one check is necessary
- Example: DB = 0x12340000 ; DL = 0x1234FFFF
  - The region ID is 0x1234
  - “[r3+12]:= r4” becomes

```
r10 := r3 + 12
r10 := r10 >> 16 // right shift 16 bits to get the region ID
if r10 <> 0x1234 then goto error
[r10] := r4
```
Optimization: Ensure, but don’t check

- Force the upper bits in the address to be the region ID
  - Called **masking**
  - no branch penalty
- Example: DB = 0x12340000 ; DL = 0x1234FFFF
  - “[r3+12]:= r4” becomes

```
\[
\begin{align*}
  r10 &:= r3 + 12 \\
  r10 &:= r10 \& 0x0000FFFF \\
  r10 &:= r10 | 0x12340000 \\
  [r10] &:= r4
\end{align*}
\]
```

Force the address to be in data region
Wait! What about Program Semantics?

- “Good” programs won’t get affected
  - For bad programs, we don’t care about whether its semantics is destroyed
- PittSField reported 12% performance gain for this optimization
- Cons: does not pinpoint the policy-violating instruction
Optimization: One-Instruction Masking (PittSField)

- **Idea**
  - Make the region ID to have only a single bit on
  - Make the zero-tag region unmapped in the virtual address space
- **Benefit:** cut down one instruction for masking
- **Example:** DB = 0x20000000 ; DL = 0x2000FFFF
  - Region ID is 0x2000
  - “[r3+12]:= r4” becomes
    
    \[
    \begin{align*}
    r10 &:= r3 + 12 \\
    r10 &:= r10 \& 0x2000FFFF \\
    [r10] &:= r4
    \end{align*}
    \]
  - Result is an address in DR or in the (unmapped) zero-tag region
- **PittSField** reported 10% performance gain for this optimization
Optimization: Fault Isolation vs. Protection

- Protection is fail stop
  - Sandbox reads, writes, and jumps
  - Guarantee integrity and confidentiality
  - 20% overhead on 1993 RISC machines
  - XFI JPEG decoder: 70-80%

- Fault isolation: covers only writes and jumps
  - Guarantee integrity, but not confidentiality
  - 5% overhead on 1993 RISC machines
  - XFI JPEG decoder: Writes only: 15-18%

- As a result, most SFI systems do not sandbox reads
Risk of Computed (Indirect) Jumps

- Worry: what if the return address is modified so that the ret instruction jumps directly to the address of “r[10] := r4”? 
  - The attack bypasses the masking before “r[10] := r4”!
  - If attacker can further control the value in r10, then he can write to arbitrary memory location
- In general, any computed jump might cause such a worry
  - jmp %eax
- BTW, direct jumps (pc-relative jumps) are easy to deal with
The Original SFI Solution [Wahbe et al. 1993]

- Make r10 a dedicated register
  - r10 only used in the monitor code, not used by application code
  - Also maintain the invariant that r10 always contains an address with the correct region ID before any computed jumps
- Cons?
  - Reduce the number of registers available to application code
  - OK for most CISC machines (E.g., MIPS has 32 registers)
  - x86-32 has only 8 integer registers (6 general purpose ones);
    - x86-64: 16
A Solution for x86 (PittSFIELD)

- Divide the code into chunks of some size
  - E.g., 16 bytes

- Make unsafe ops and their checks stay within one chunk
  - E.g., “r10 := r10 & 0x2000ffff; [r10] := r4”

- Mask jump targets so that they are aligned: multiples of the chunk size
  - E.g., “jmp r5” becomes
    - r5 := r5 & 0x1000FFFF0
    - jmp r5

Note: the above assumes the region ID for the code region is 0x1000; a single instruction for sandboxing and alignment requirement
Downside of the alignment solution

- All legitimate jump targets have to be aligned
  - No-op instructions have to be inserted sometimes
  - For example: “i1; i2; i3”
    - Suppose both i1 and i3 are possible jump targets
    - Then it becomes “i1; i2; nop; nop; ...; nop; i3”

- Cons: slow down execution and increase code size
Jumping Outside of Fault Domains

- Sometimes need to invoke code outside of the domain
  - For system calls; for communication with other domains
  - Danger: Cannot allow untrusted code to invoke code outside of the fault domain arbitrarily

- Idea:
  - Insert a jump table into the (immutable) code region
  - Each entry is a control transfer instruction whose target address is a legal entry point outside of the domain
A Fixed Jumptable (Trampolines)

For example
- Trampolines for system calls: fopen; fread; ...
- Trampoline for communication with other fault domains

Fault Domain

Trampolines

Code Region

Data Region

stubs to trusted routines
Trusted Stubs

- Stubs are outside of the fault domain
- Stubs can implement security checks
  - E.g., can restrict fopen to open files only in a particular directory
  - Or can disallow fopen completely
    - Just not install a jump table entry for it
- It can implement system call interposition
Incorporating SFI in Applications
Google’s Native Client (NaCl)

- New SFI service in Chrome
  - [Yee et al. Oakland 09]
- Goal: download native code and run it safely in the Chrome browser
  - Much safer than ActiveX controls
  - Much better performance than JavaScript, Java, etc.
NaCl: Code Verification

- Code is verified before running
  - Allow restricted subset of x86 instructions
    - No unsafe instructions: memory-dependent jmp and call, privileged instructions, modifications of segment state ...
  - Ensure SFI checks are correctly implemented for memory safety
NaCl Sandboxing

- x86-32 sandboxing based on hardware segments
  - Sandboxing reads and writes for free
  - 5% overhead for SPEC2000
- However, hardware segments not available in x86-64 or ARM
  - Still need masking instructions [Sehr et al. 10]
  - x86-64/ARM: 20% for sandboxing mem writes and computed jumps
NaCl SDK

- Modified GCC tool-chain
  - Inserts appropriates masks, alignment requirements
- Trampolines allow restricted system-call interface and also interaction with the browser
  - Pepper API: access to the browser, DOM, 3D acceleration, etc.
Robusta [Siefers, Tan, Morrisett CCS 2010]

- New SFI service in a Java Virtual Machine (JVM)
  - Allow Java code to invoke native code safely through the Java Native Interface (JNI)
- The basic idea
  - Put native code in an SFI sandbox and allows only controlled access to JVM services
Robusta [Siefers, Tan, Morrisett CCS 2010]

Native Code Threat
- Direct JVM mem access
- Abusive JNI calls
- OS syscalls

Robusta Remedy
- SFI: Prevent direct JVM access
- Perform JNI safety checking
- Reroute syscall requests to Java’s security manager
Control-Flow Integrity (CFI)
Main Idea

1) Pre-determine the control flow graph (CFG) of an application
2) Enforce the CFG through a binary-level IRM

CFI Policy: execution must follow the pre-determined control flow graph, even under attacks

Attack model: the attacker can change memory between instructions, but cannot directly change contents in registers
Why is it Useful?

Lots of attacks induce illegal control-flow transfers: buffer overflow, return-to-libc, ROP
Control-Flow Graph (CFG)

- The CFG is part of the policy
  - Can be coarse grained or fine grained

- Examples:
  - A control-flow transfer must target the beginning of a legal machine instruction
  - A control-flow transfer must target the beginning of a 16-byte trunk (required by NaCl and PittSFleld)
  - An indirect jump must target the beginning of a libc function

- How to get the CFG?
  - Explicit specification; Static analysis of source code; Execution profiling; Static binary analysis
bool lt(int x, int y) {return x<y;}
bool gt(int x, int y) {return x>y;}
void sort(...) {...; return }
void sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}
CFI Enforcement

- Can be enforced through an IRM [Abadi, Budiu, Erlingsson, Ligatti CCS 2005]
- A direct jump can be verified statically
- For computed jumps
  - Insert an ID at every destination given by the CFG
  - Insert a runtime check to compare whether the ID of the target instruction matches the expected ID
A side-effect free instruction with an ID embedded

call sort
prefetchnta [$ID]

call sort

sort:
...
ret

Opcode of prefetch takes 3 bytes

sort:
...
ecx := [esp]
esp := esp + 4
if [ecx+3] <> $ID goto error
jmp ecx
CFI Assumptions

- Non-writable code region
  - IDs are embedded into the code

- Non-executable data region
  - Otherwise, the attacker can fake an ID

- Unique IDs
  - Bit patterns chosen as IDs must not appear anywhere else in the code region
CFI Imprecision

- Equivalent destinations
  - Two destinations are equivalent if CFG contains edges to each from the same source
  - Use same ID for equivalent destinations
- This is imprecise
Example of Imprecision

- Return in `bar()` can return to either `foo1` or `foo2`
- Essentially, CFI allows unmatched calls and returns
  - `foo1 -> bar -> return to foo2`
- It enforces a FSA, instead of PDA

```c
void foo1 () {
    ...; bar(); ...
}
void foo2 () {
    ...; bar(); ...
}

void bar () {
    ...; return;
}
```
CFI: Security Guarantees

- Effective against attacks based on illegal control-flow transfer
  - Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge

- Does not protect against attacks that do not violate the program’s original CFG
  - Incorrect arguments to system calls
  - Substitution of file names
  - Non-control data attacks
CFI and Static Analysis
Going Beyond Simple IRM

- In simple IRM, a check is inserted right before each unsafe instruction

Can we do better than that? Do we have to insert a check right before each unsafe instruction?
IRM Optimization

- IRM optimization through **static analysis**
  - Analyze contexts where checks are inserted
  - Simplify, eliminate, and move checks

- Challenges
  - Static analysis requires a control-flow graph
    - That is exactly what CFI gives you
  - Verifier harder to construct: need to verify the result of optimizations
CFI and Static Analysis

- CFI enables static analysis
  - **Optimization**: eliminate safety checks if they are statically proven unnecessary
  - **Verification**: use static analysis to verify the result of optimizations.
Efficient Data SFI [Zeng, Tan, Morrisett CCS 2011]

- We tried this idea to optimize data SFI
- Sandbox both memory writes and reads
  - Previous software-based SFI systems have high overheads when sandboxing both reads and writes
  - JPEG image decoder in XFI
    - Writes only: 15-18%
    - Reads and writes: 70-80%
Data SFI Policy

A memory read/write is safe if the address is in $[DB-GSize, DL+GSize]$.

Assumption: access to guard zones are trapped by hardware.
Data SFI Optimizations

- Liveness analysis to find spare registers for masking
- In-place sandboxing
- Redundant check elimination
- Loop check hoisting

Similar to those classic optimizations performed in an optimizing compiler
Example: Redundant Check Elimination

Before optimization
ecx := mask(ecx)
eax := [ecx + 4]
ecx := mask(ecx)
eax := [ecx + 8]

After optimization
ecx := mask(ecx)
eax := [ecx + 4]
ecx := mask(ecx)
eax := [ecx + 8]

The masking forces ecx to be in DR; then ecx+4 must be in DR or guard zones
Example: Loop Check Hoisting

Before optimization
esi := eax
ecx := eax + ebx * 4
edx := 0
loop:
  if esi >= ecx goto end
  esi := mask(esi)
edx := edx + [esi]
esi := esi + 4
jmp loop
end:

After optimization
esi := eax
ecx := eax + ebx * 4
edx := 0
esi := mask(esi)
loop:
  if esi >= ecx goto end
edx := edx + [esi]
esi := esi + 4
jmp loop
end:
Without optimizations, the logic of the verifier is easy

- Just check there is a masking instruction immediately before each memory operation

Our new verifier

1. Perform **range analysis** to compute the ranges of values in registers
2. Traverse the program and check the range of the address of each mem operation
   - if the address range is within \([DB-GSize, DL+GSize]\),
     - then OK
   - else report_error()
Checking the Safety of the Loop-Hoisting Example

esi := eax
ecx := eax + ebx * 4
edx := 0
esi := mask(esi)
esi ∈ [DB, DL]
loop:
esi ∈ [DB, DL+4]
if esi >= ecx goto end
esi ∈ [DB, DL+4]
edx := edx + [esi]
esi ∈ [DB, DL]
esi := esi + 4
esi ∈ [DB+4, DL+4]
jmp loop
end:

[DB, DL+4] ⊆ [DB-GSize, DL+GSize]
SPECint2000 Evaluation

W+CFI: 10.4%
R+W+CFI: 27.1%
Verifying the Verifier
One Key Issue in IRM

- Code is verified before execution
  - Google NaCl’s verifier: pile of C code with manually written decoder for x86 binaries
- A bug in the verifier could result in a security breach.
  - Google ran a security contest early on its NaCl verifier: bugs found!

Question: How to construct high-fidelity verifiers?
Verifying the Verifier

- Goal: a provable correct SFI verifier
- Theorem: if some binary passes the verifier, then the execution of the binary should obey the SFI policy
RockSalt Punchline

- **RockSalt**: a new verifier for x86-32 NaCl
  - [Morrisett, Tan, Tassarotti, Gan, Tristan PLDI 2012]

- **Smaller**
  - Google: 600 lines of C with manually written code for partial decoding
  - RockSalt: 80 lines of C + regexps for partial decoding

- **Faster**: on 200Kloc of C
  - Google’s: 0.9s
  - RockSalt: 0.2s

- **Stronger**: (mostly) proven correct
  - The proof is machine checked in Coq
RockSalt Architecture

- Verifier
  - Regexps for decoding
  - Code for checking SFI constraints

- Correctness Proof
  - SFI theorem and proof
  - Decoding correctness
  - Properties of instructions

- x86 model
  - Decoder Spec
  - Instruction semantics
  - RTL machine

- ~10,000 Coq
- ~5,000 Coq
How RockSalt’s Verifier Works

- Specify regular expressions (regexps) for partial decoding of x86 instructions
  - One regexp to recognize all legal non-control-flow instructions
  - One regexp for all direct control flow instructions
  - One regexp for a masking instruction followed by indirect jumps
- Compile regexps to DFA tables
- Run DFAs and check SFI constraints
  - Record start positions of instructions
  - Check jump and alignment constraints
x86 Decoder Specification

- A decoder spec language: a set of regular expression parsing combinators
  - Used in the partial decoder of the verifier
  - Also used in the full decoder
- Extracted an executable decoder from the spec
  - Based on derivative-based parsing [Brzozowski 1964; Owens et al. 2009; Might et al. 2001]
Definition CALL_p : grammar instr :=
   "1110" $$ "1000" $$ word @
   (fun w => CALL true false (Imm_op w) None)
| | "1111" $$ "1111" $$ ext_op_modrm2 "010" @
   (fun op => CALL true true op None)
| | "1001" $$ "1010" $$ halfword $ word @
   (fun p => CALL false false (Imm_op (snd p)) (Some (fst p)))
| | "1111" $$ "1111" $$ ext_op_modrm2 "011" @
   (fun op => CALL false true op None).

alternatives

Decode pattern

Semantic actions
x86 Decoder Specification

- Specified the decoding of all integer x86-32 instructions
  - Over 130 instructions for the decoder
  - With prefixes
  - An almost direct translation from Intel’s decoding tables to patterns in the spec
- One undergraduate constructed a decoder for MIPS in just a few days
Semantics specified by translating an instruction into a sequence of instructions in a register transfer language (RTL)

- RTL is a RISC-like machine with a straightforward semantics
- With a few orthogonal instructions

- Over 70 instructions with semantics
  - With modeling of flags, segment registers, ...
Model Validation

- Extracted from the model an executable x86 interpreter
- Compared the interpreter with real processors
  - Used Intel’s PIN to instrument binaries to dump out intermediate states
- Testing
  - Csmith: generate random C programs, compile, test the interpreter against implementations.
    - Tested ~10M instructions in ~60 hours
  - Used decoder spec to generate fuzz tests.
What was Proved...

- Translation of regexps to DFA tables is correct.
- RockSalt verifier correctness
  - Program passing the verifier preserves a set of invariants that imply that the code obeys the SFI policy
- A lot of automation to make the proof scale
  - Relative easy to add a new instruction and extend the proof
Open Problems
Does SFI Scale to Secure Systems?

- SFI is good at isolating untrusted code in a trusted environment
- Can we partition a large system into domains of least privileges?
  - How to perform partitioning? At binary level?
  - Monitor information flow between domains?
  - What about performance?
Accommodating Dynamic Features

- IRM: requires statically known code for rewriting and verification
- Dynamic loading/unloading libraries
  - E.g., how to do CFI in the presence of dynamically loaded libraries?
- Dynamic code generation; JIT; self-modifying code
  - How to maintain SFI, CFI invariants when code is generated on the fly?
- Need **modular rewriting and verification techniques**
Binary Rewriting on Off-the-Shelf Binaries

- SFI implementations ask cooperation from code producers
  - NaCl has a modified GCC toolchain to emit policy-compliant binary
  - Our lab session: modify LLVM
- Ideally, want to statically rewrite off-the-shelf binaries
- Two key challenges
  - Disassembly: code mixed with data; obfuscation; ...
  - Adjusting jump targets after rewriting
- Possible way out: incorporating some dynamic component
  - DynamoRio; PIN; ...
  - E.g., [Smithson et al. 10] made some progress on rewriting binaries without relocation information
Processor Models

- Useful: certified software; binary analysis; ...
- Not ideal: each research group works on its own x86 model
- We want public spec of processors
  - Well tested
  - Incorporate commonly used features
  - Robust to processor evolution
  - Support formal reasoning
  - Support x86-32, x86-64, ARM
- A set of reusable tools is the key
Bibliography

- Classification of security policies
Bibliography

- **Inlined Reference Monitors**
Bibliography

- **Low-level IRM**
  - [Castro et al. 06]. Securing software by enforcing data-flow integrity. OSDI, 2006.
Low-level IRM, cont’d


[Siefers & Tan & Morrisett 10]. Robusta: Taming the native beast of the JVM. In 17th CCS, pages 201–211, 2010.


[Morrisett et al. 12]. RockSalt: Better, Faster, Stronger SFI for the x86. PLDI, 2012
Lab Session Overview
LLVM Compiler Architecture

- Optimizer: has multiple passes that perform bitcode-to-bitcode transformation
- LLVM command-line tool demo
Lab Setup

- We ask you add an extra LLVM pass to instrument memory writes
  - Add one masking instruction before each memory write
  - If you are new to LLVM, read some online tutorial about how to add a pass
Several steps

- **Step 1:**
  - Add a pass to Hello.cpp to dump every memory operation in bitcode

- **Step 2:**
  - Add a pass in InsMemWrite.cpp to instrument memory writes

- **Step 3**
  - An optimization that has less instrumentation overhead
  - I have a VirtualBox VM image, which you can use after the lab session
Notes

- Simplifications made for the lab exercise
  - Control-flow aspect is ignored
  - Because we perform bitcode-to-bitcode transform, we need to trust the code generator
- After instrumentation, the binary cannot run directly
  - You need a special loader that sets up the data and code regions at the correct place