Inlining a reference monitor

Applied Static Analysis 2016

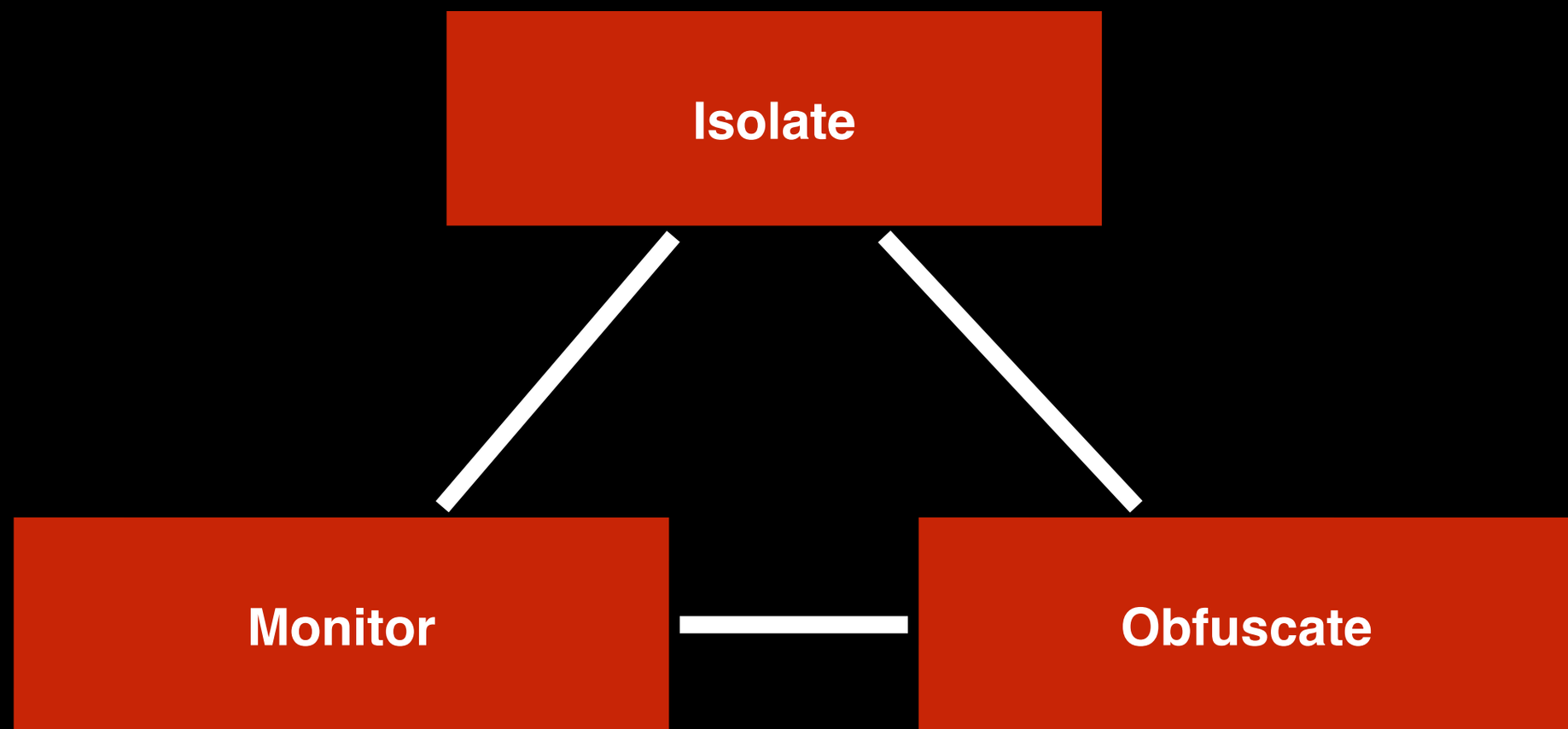
Ben Hermann
@benhermann

Dr. Michael Eichberg, Johannes Lerch, Sebastian Proksch, Karim Ali Ph.D.

Software Technology Group

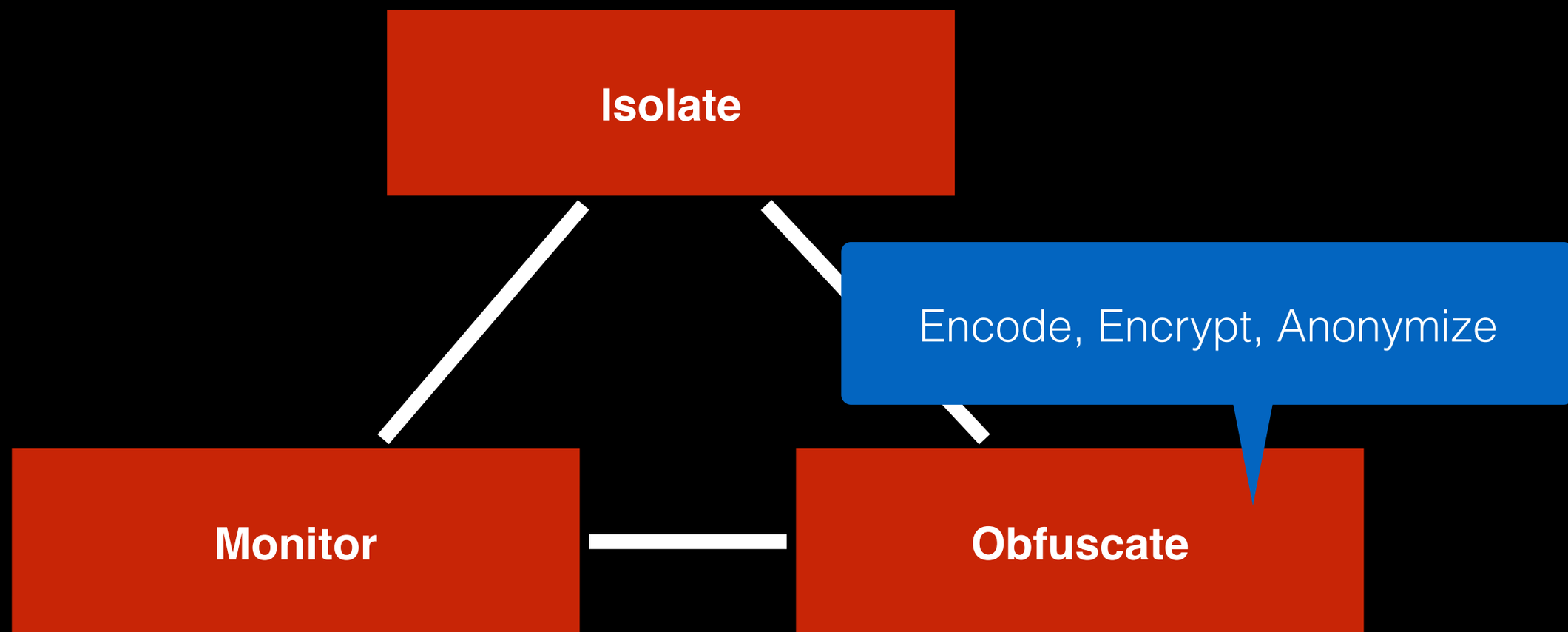
TECHNISCHE UNIVERSITÄT DARMSTADT

Dimensions of Security



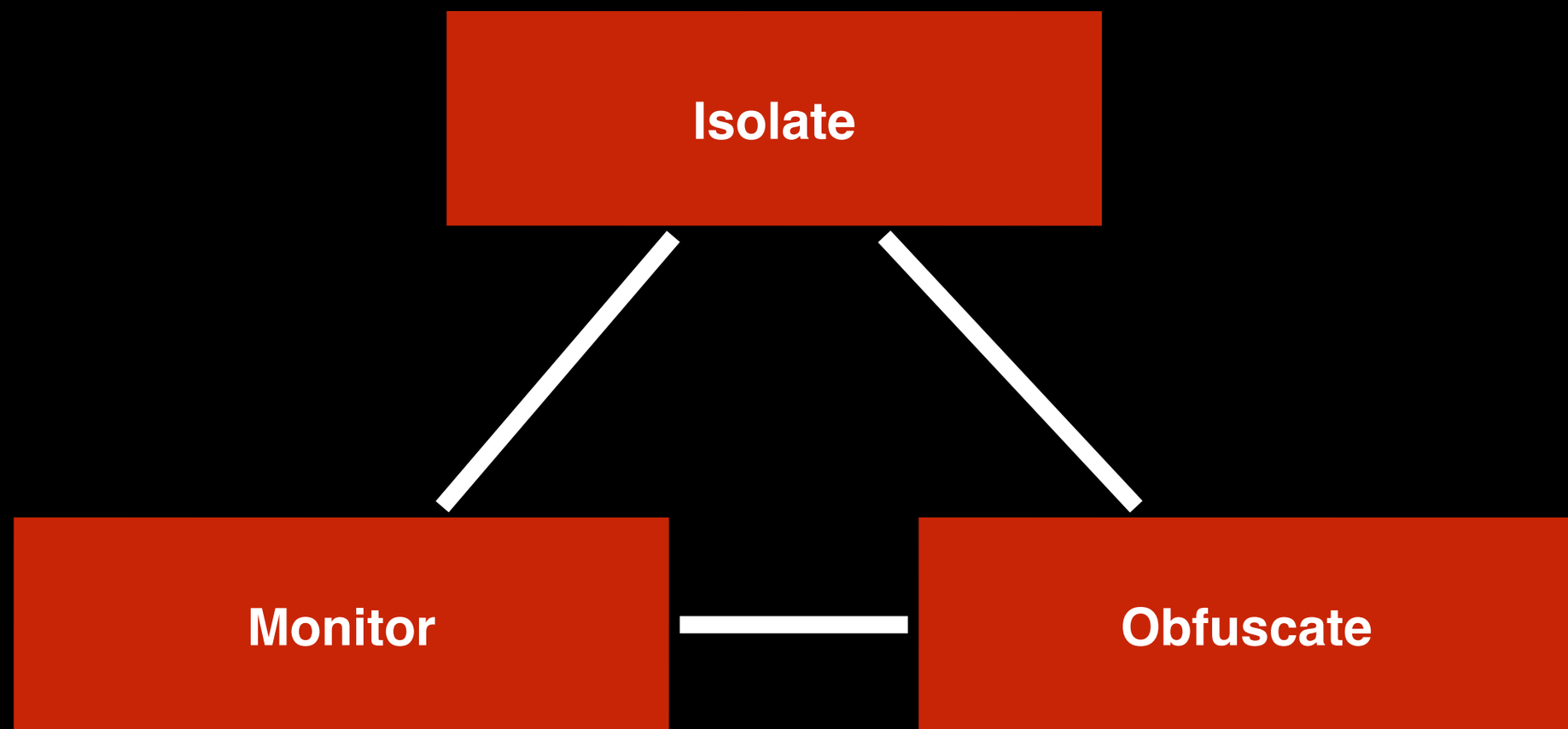
exercise 7.6

Dimensions of Security



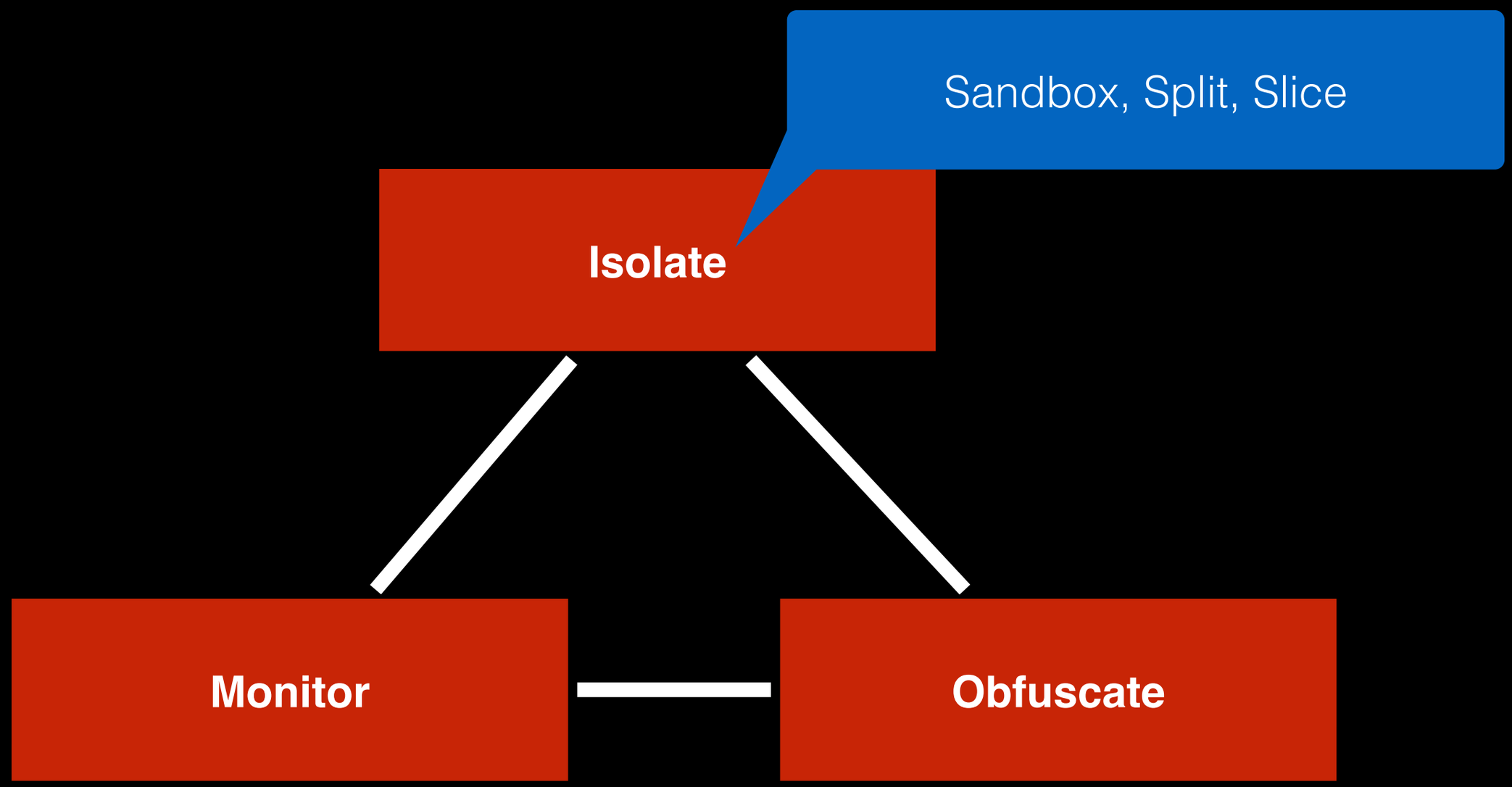
exercise 7.6

Dimensions of Security



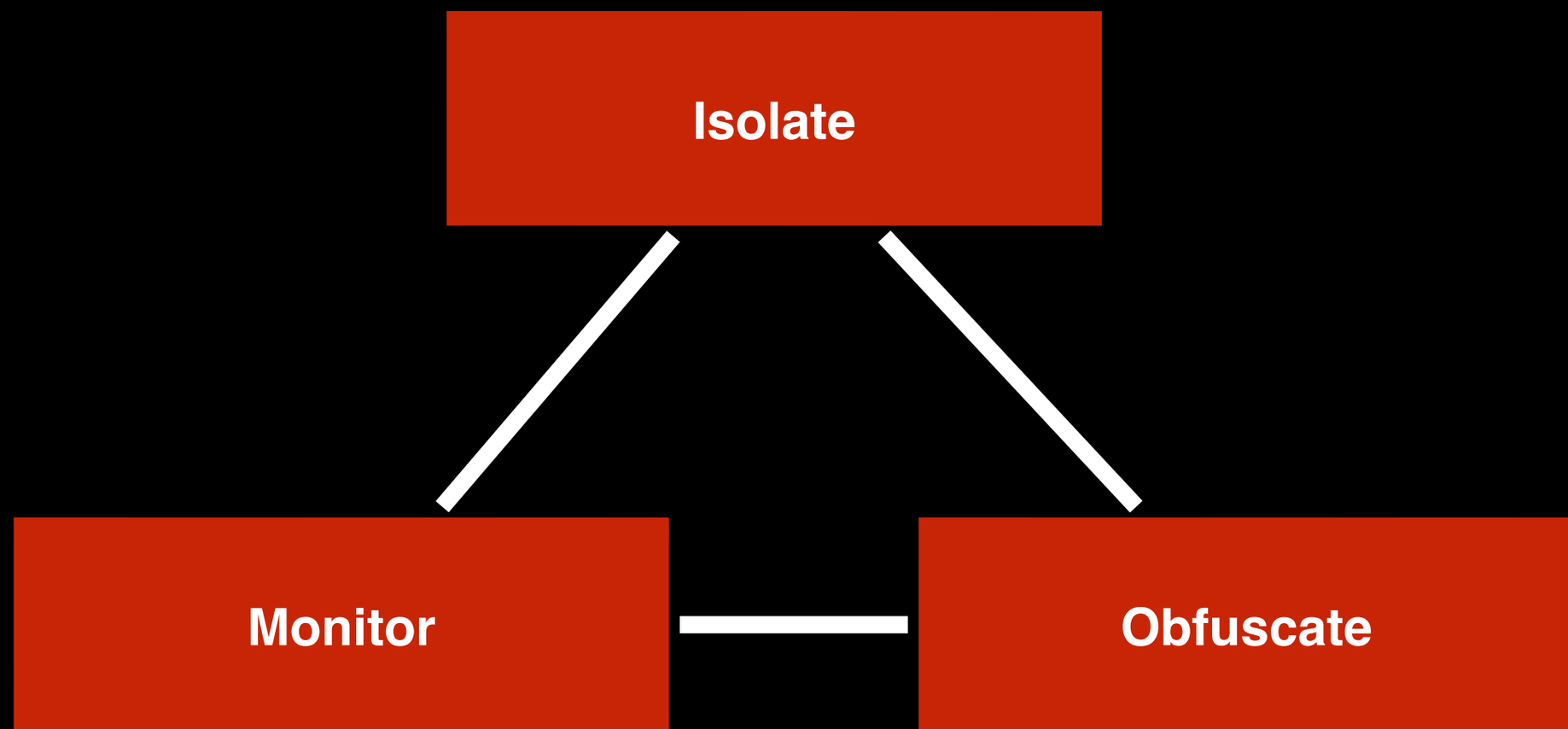
exercise 7.6

Dimensions of Security



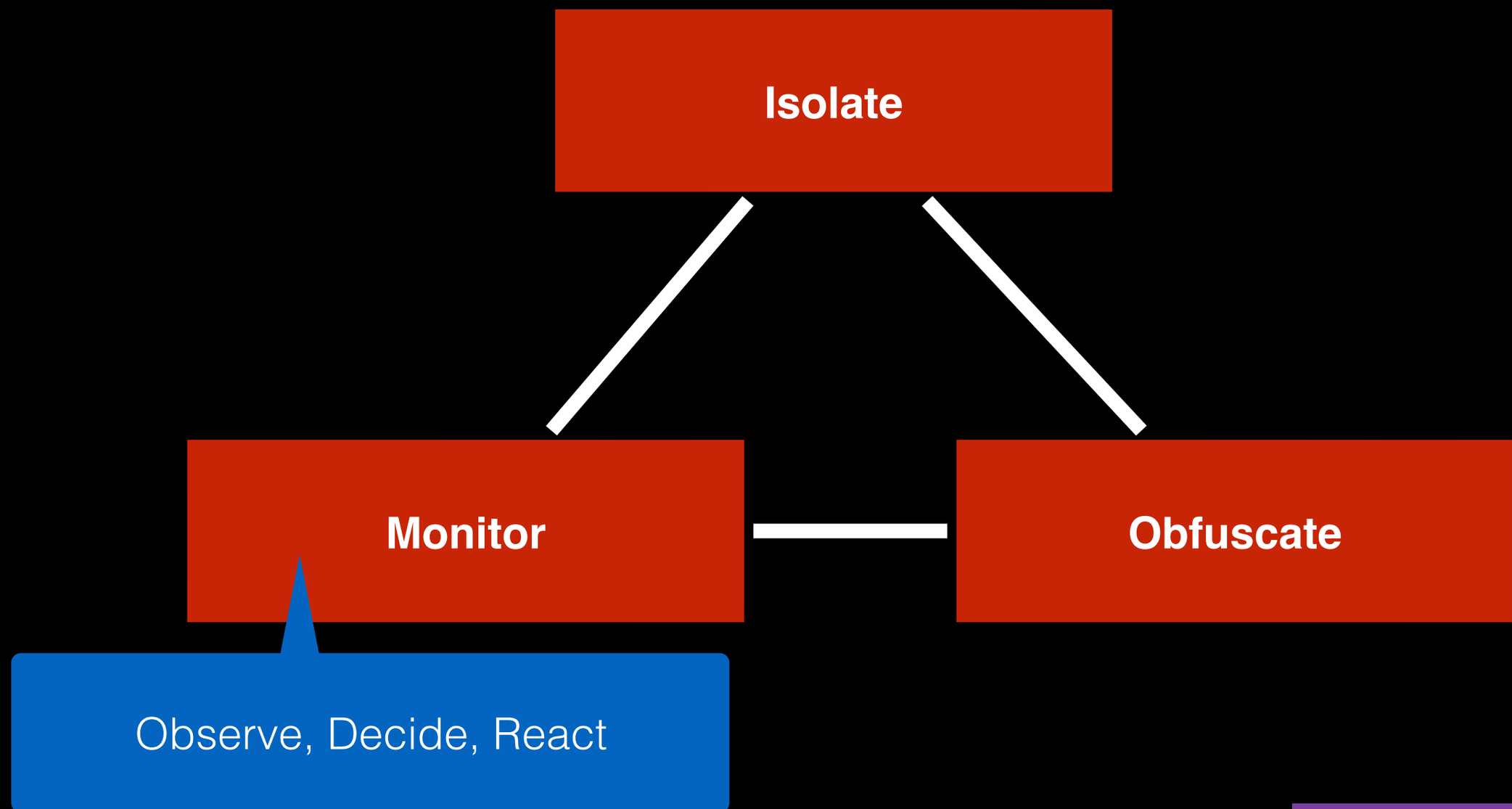
exercise 7.6

Dimensions of Security



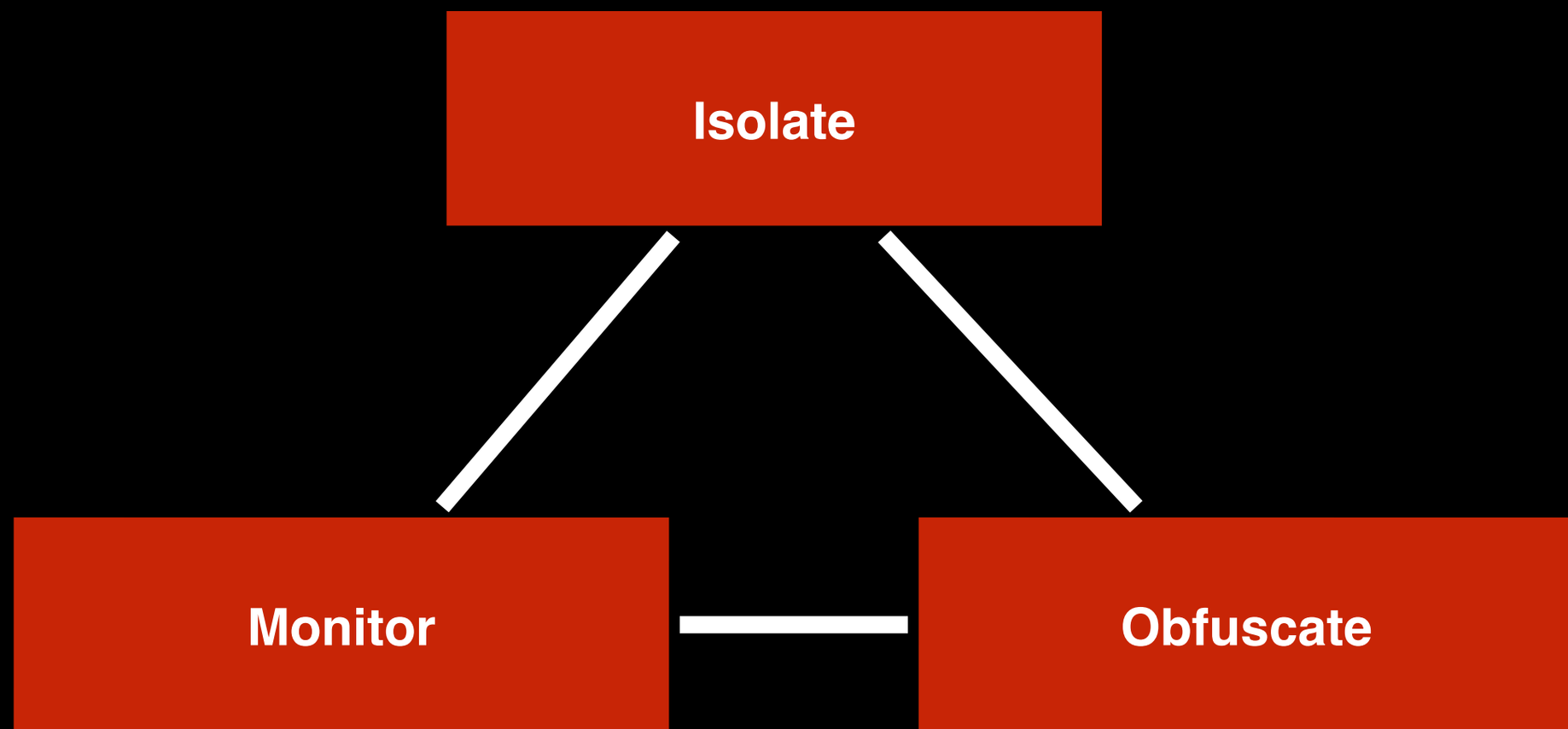
exercise 7.6

Dimensions of Security



exercise 7.6

Dimensions of Security



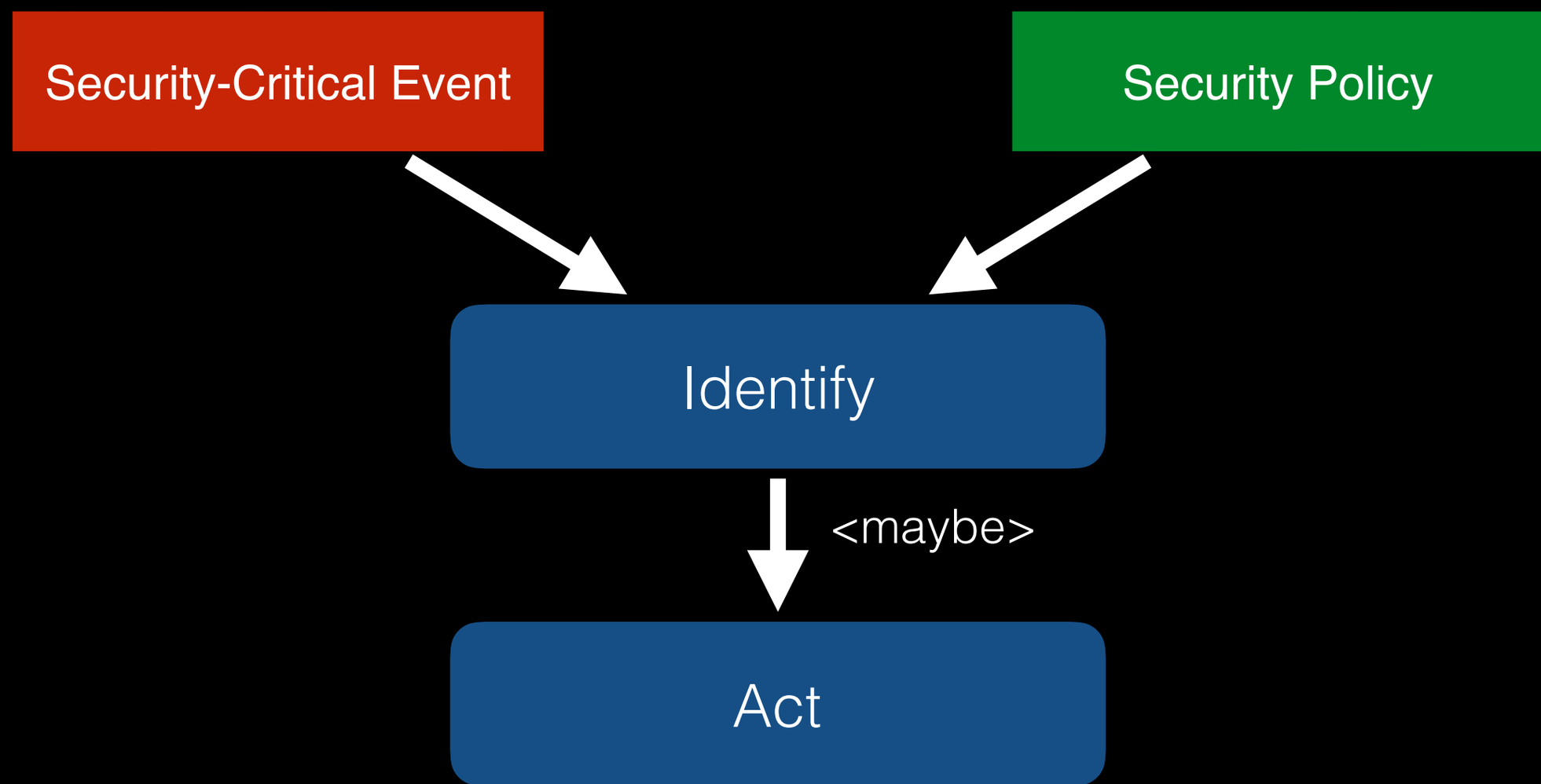
exercise 7.6

Exercise: Inline Reference Monitor

- A reference monitor observes the execution of a program
- It halt or pauses the execution if something “bad” happens
- What “bad” ist defines a security policy
- It makes sense to observe only security critical events

exercise 7.6

Exercise: Inline Reference Monitor



exercise 7.6

Exercise: Inline Reference Monitor

- Goal: Write a transformer that injects a reference monitor according to the following specification.

Security-Critical Event	Any function calls
Security Policy	Prevent function calls when the first argument is a number of value 42

- If you want to make that very elegant you decompose it into different passes and make it configurable

exercise 7.6

Identifying Events

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

exercise 7.6

Identifying Events

Protect the IRM itself from injection

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

exercise 7.6

Identifying Events

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

exercise 7.6

Identifying Events

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

Only process calls

exercise 7.6

Identifying Events

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

exercise 7.6

Identifying Events

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

At least one operand

exercise 7.6

Identifying Events

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

exercise 7.6

Identifying Events

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

First operand is an integer

exercise 7.6

Identifying Events

```
bool matchesEvent(Instruction* i) {
  if (i->getFunction()->getName().startswith("irm"))
    return false;

  CallInst* call = dyn_cast<CallInst>(i);
  if (!call) return false;
  if (call->getNumArgOperands() < 1) return false;

  Value* firstOperand = call->getArgOperand(0);

  if (!(firstOperand->getType()->isIntegerTy())) return false;

  return true;
}
```

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n";
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n"
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

Create a function prototype

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n";
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n";
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

Reuse call operand

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n";
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n";
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

Create constant for policy

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n";
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n";
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

Inject call

exercise 7.6

Injecting IRM call

```
for (CallInst *i : injectBefore) {
  errs() << "Injecting IRM call\n";
  IRBuilder<> builder(i);
  Function *irmCall = getIRMCallPrototype(i->getContext(),
                                          i->getModule());

  Value *irmCall_params[] = {
     i->getArgOperand(0),
     ConstantInt::get(i->getArgOperand(0)->getType(), 42)
   };

  builder.CreateCall(irmCall, irmCall_params);
}
```

exercise 7.6

Getting the IRM function

```
Function *getIRMCallPrototype(LLVMContext &ctx, Module *mod) {
  Function *existing = mod->getFunction("irmCall");
  if(existing) return existing;

  return createReferenceMonitor(ctx, mod);
}
```

Creating the IRM function

```
Function *createReferenceMonitor(LLVMContext &ctx, Module *mod) {
  Type *i32 = IntegerType::getInt32Ty(ctx);

  FunctionType *irmcall_type =
     TypeBuilder<void(int, int), false>::get(getGlobalContext());
  Function *func =
     cast<Function>(mod->getOrInsertFunction("irmCall",
                                             irmcall_type));

  IRBuilder<> *builder =
       new IRBuilder<>(BasicBlock::Create(ctx, "initial", func));
  // %1 = alloca i32, align 4
  Value *firstAlloc = builder->CreateAlloca(i32);
  // %2 = alloca i32, align 4
  Value *secondAlloc = builder->CreateAlloca(i32);
  // store i32 %actual, i32* %1, align 4

...
```

Demo Time!

Guard and Sanitizer Detection

```
void callerOne(int n) {
 if (n > 0 && n <= 4096)
   criticalOperation(n);
}
```

```
void callerFour(int n) {
 criticalOperation(n);
}
```

```
void callerOne(int n) {
 if (n > 0 || n <= 4096)
   criticalOperation(n);
}
```

```
void callerOne(int n) {
  n = clear(n);
  criticalOperation(n);
}
```

Guard and Sanitizer Detection

Shameless Advertisement

```
void callerOne(int n) {
 if (n > 0 && n <= 4096)
   criticalOperation(n);
}
```

```
void callerFour(int n) {
 criticalOperation(n);
}
```

```
void callerOne(int n) {
 if (n > 0 || n <= 4096)
   criticalOperation(n);
}
```

```
void callerOne(int n) {
  n = clear(n);
  criticalOperation(n);
}
```

Exercises in this Block

exercise 7.6 Inline Reference Monitor