# Security Research & Development with LLVM

Andrew R. Reiter

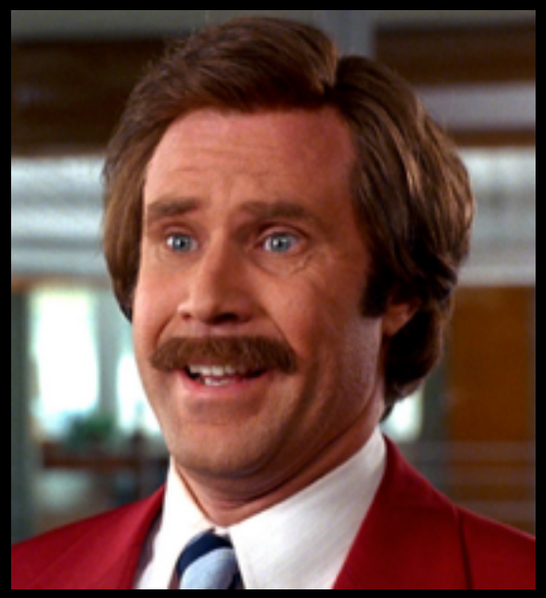
OPCDE 2017

Part of longer term research effort between speaker and Jared Carlson

# Bio

- Principal Security Researcher, Veracode (USA)
- Past: vuln research, exploit dev, malware analysis, OS dev, RF work
- Ancient history: w00w00, HERT, FreeBSD

*"…standing on the shoulders of giants" — Newton*



# Obvious trend?

- Increase in mingling: academia and "security scene"
- Demand for increased rigor in security research
- Thus increasingly specialized….
- Desire for reusability & modularity

1 example of this trend… LLVM

# Code Share

- Some *small*, **basic** example codes
- Introduce APIs and workflow, so can…
- More easily read research code for *meaning*

https://github.com/roachspray/opcde2017

Mostly look at: intflip, npassert, comminute

# LLVM Compiler Infrastructure

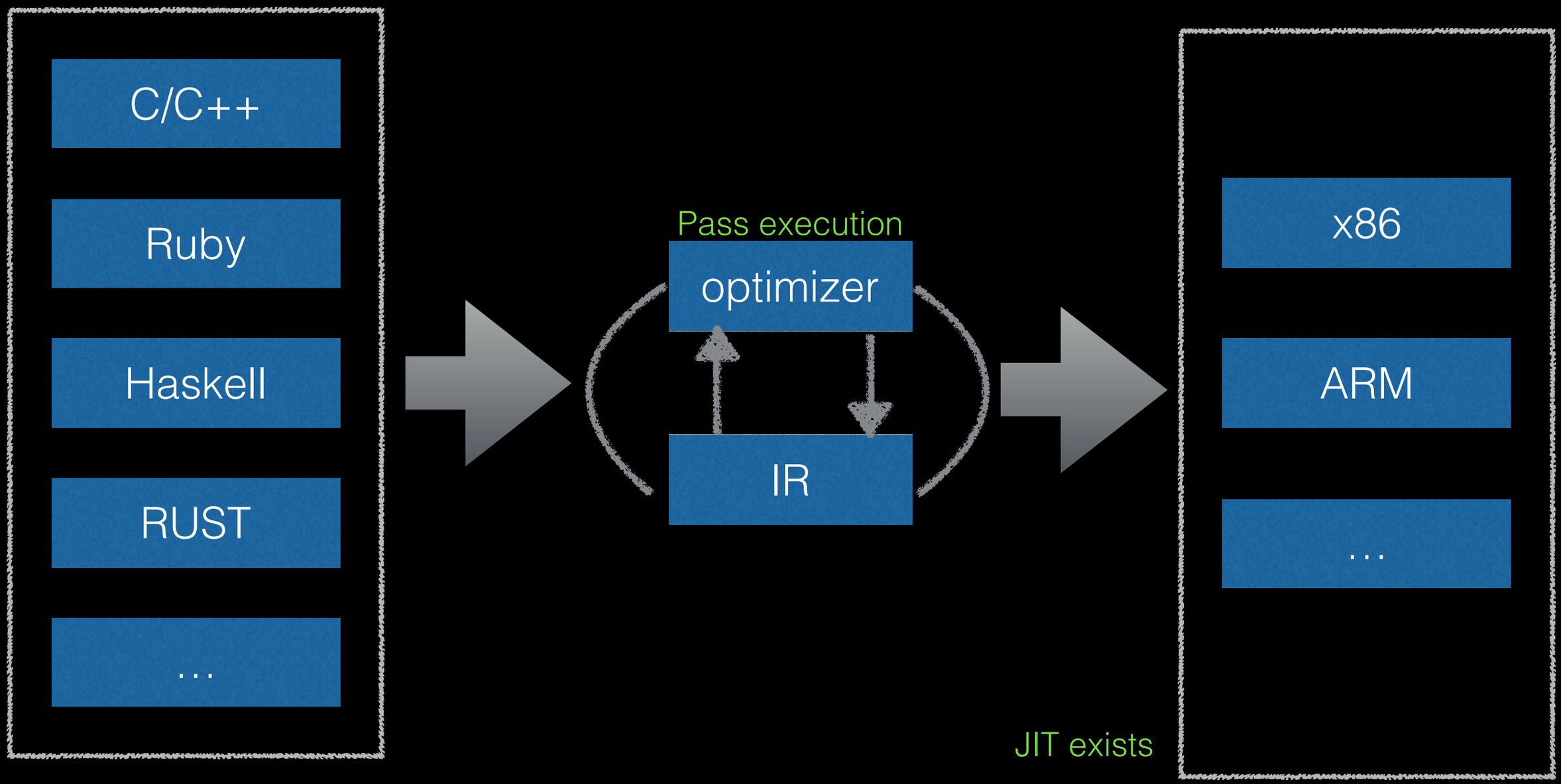
- Intended for modern compiler research:
  - Optimization
  - Program analysis
- Started UIUC in early 2000s: Vikram Adve, Chris Lattner
- Open source — the core is the community
- (Major) Corporate support






http://llvm.org/pubs/2002-12-LattnerMSThesis.html

# High Level Workflow|Arch



# Intermediate Representation

- Static Single Assignment (SSA)
- Strongly typed
- Architecture and Language agnostic
- 3 forms:
    1. in-memory
    2. on-disk bitcode (.bc)
    3. on-disk human readable (.ll) (looks assemblerish)

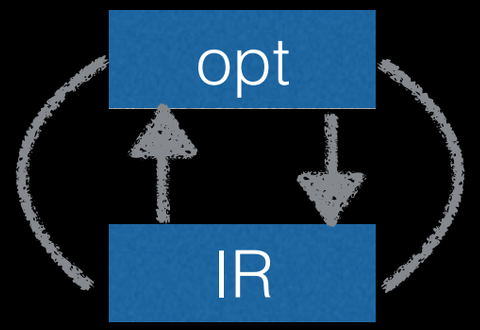
# Core API & Utils

- Stable/robust APIs for:
    - IR generation and manipulation
    - Analysis and optimization of IR
    - Machine code generation
- Still: research areas
- Tools: clang, opt, llvm-dis, klee, many more.

* http://llvm.org/docs/CommandGuide/

# Passes are King

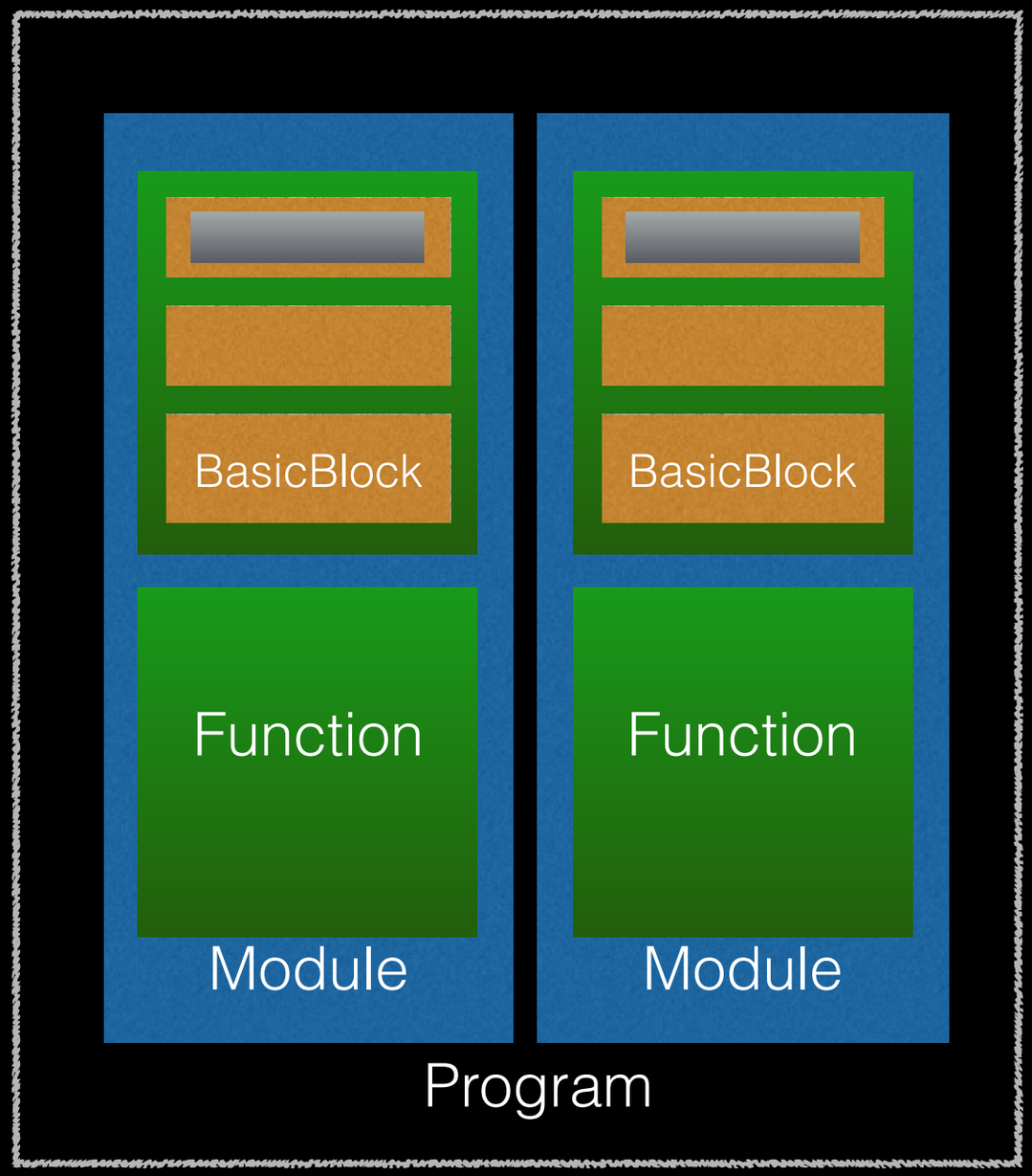
- Mapping IR to IR
- Either:
  - Analysis passes ~ read-only
  - Transform passes ~ read/write




- **Chain** ‘em — analyze, then use results in another
- Many provided by LLVM; Build your own for fun|profit

# Pass Impl Types

- ModulePass
- FunctionPass or MachineFunctionPass
  - BasicBlockPass
  - LoopPass
  - RegionPass
  - CallGraphSCCPass
- Atleast implement *runOnTYPE() function*




*The code share has skeletons of each

# Pass Rules Everything Around Me, P.R.E.A.M.

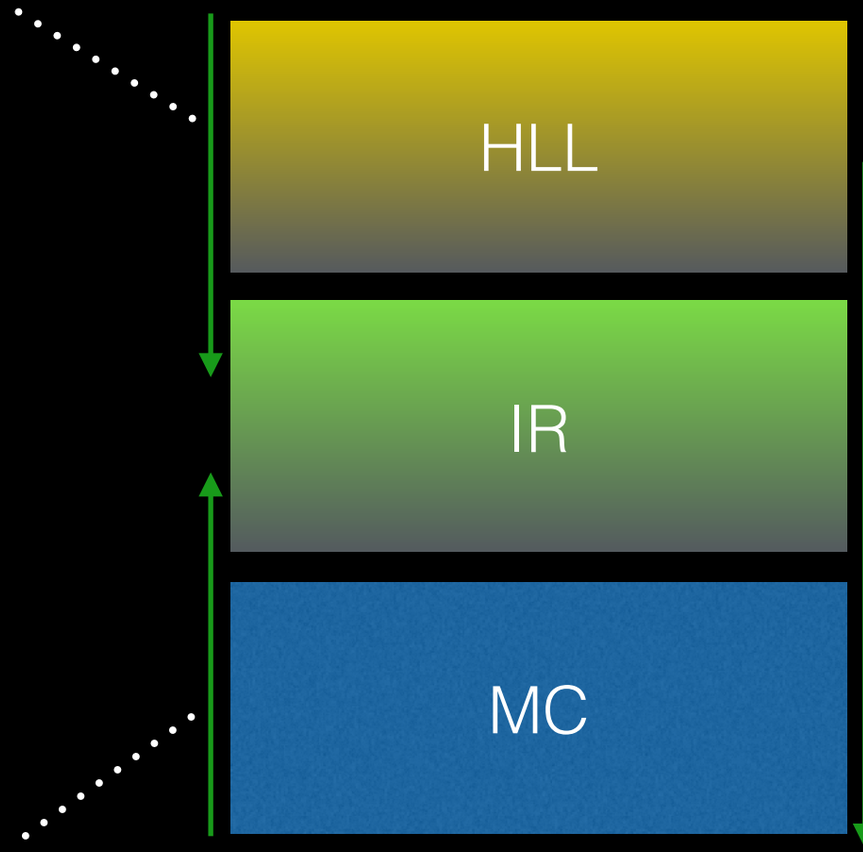

- Different pass type => different restrictions*

1. Determine the needs of your analysis|transform
2. Find Pass type supporting your needs.
3. Use “Principle of least privilege” mindset.

*Find the restrictions listed: http://llvm.org/docs/WritingAnLLVMPass.html

# Security R&D + LLVM

- static analysis
- proving properties
- symbolic exec
- reverse engineering




- fuzzing
- harden code
- code obfuscation

Quarkslab

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Carnegie
Mellon
University

Penn

NC STATE UNIVERSITY

heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Verimag

UNSW SYDNEY Engineering
Computer Science and Engineering

POLITECNICO
MILANO 1863

UNIVERSITY OF
CAMBRIDGE

PRINCETON
UNIVERSITY

Grenoble INP

NASA

Microsoft

THE
UNIVERSITY
OF UTAH

deep
spec

université
PARIS-SACLAY

UNIVERSITY
OF TWENTE.

galois

FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

SCIENTIA
IMPERII DECUS ET TUTAMEN

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
MENS ET MANUS
1861

informatics mathematics
Inria

VUSec

institute
IMdea

TRAIL OF BITS

UNIVERSITÉ
Grenoble
Alpes

I wanted to list everything in this slide deck, but too much… I have a listing below

https://github.com/roachspray/opcde2017/blob/master/projects.md

- Please send me additions


# Vellvm: Verified LLVM


- Model syntax and semantics of LLVM IR  so as to…
- Reason about code expressed in IR in order to…
- Prove properties about LLVM passes in Coq *
- OCaml extracted, ran unit tests validating model



Recently revived!


* https://coq.inria.fr/

# Software Analysis Workbench (SAW)

|galois|

- Formal verification via equivalency checking
- Map IR to logical form (formal model)
- Uses multiple SMT/SAT solvers
- Able to ingest bitcode (.bc) file

http://saw.galois.com/

# Mc Sema | Remill | VMill

- Lift!
    - instruction lift
    - data inclusion and making LLVM modules


in order to perform:

- Recompilation of IR to MC
- Retargeting of IR to another arch
- Symbolic execution of IR (e.g. for fuzzing)
- Snapshot native->symbolic exec w/ memory

https://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge/
https://github.com/trailofbits/mcsema
https://github.com/trailofbits/remill

# Sanitizers

- Hybrid dynamic/static:
    - Statically instrument IR
    - Link with library to catch issues at RT

- AddressSanitizer: catch UAF, UAR, etc 
- DataFlowSanitizer: implement own taint analysis
- TypeSan: C++ cast errors 


https://github.com/google/sanitizers
https://github.com/vusec/typesan

# Fuzzing

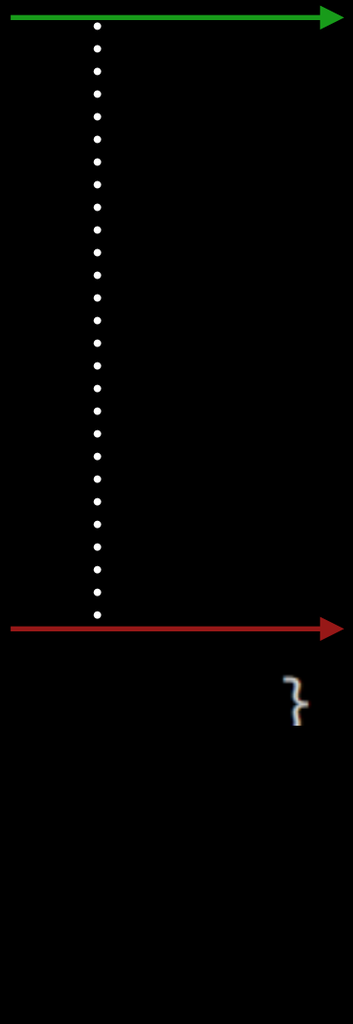
- LibFuzzer:
    - evolutionary, in-process guided fuzzer
    - *part* of LLVM project
- Improving action of existing:
    - lafindel's compare splitting passes
    - TokenCap: find magics/fuzz roadblocks & remove

# More…

- Application hardening
- Translation to formal languages
- Software resiliency
- … ok I'll stop and move forward.. but please view the list on github and check the projects out!

Ok..to some code

# Motivate: Ultra-Contrived C sensitive leak



```
char *p;
struct addrinfo hints, *result;

p = getpass("enter passwd: ");
/* l.v. p is now tainted with sensitive data */
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = 0;
hints.ai_protocol = 0;
/* leak password via getaddrinfo() DNS lookup. contrived af. */
(void)getaddrinfo(p, "http", &hints, &result);
}
```

```
[awr@anathema tests] clang-3.9 -g -emit-llvm -o NSDL001.bc -c NSDL001.c
```

```
define void @leaks_passwd() #0 !dbg !18 {
  %1 = alloca i8*, align 8
  %2 = alloca %struct.addrinfo, align 8
  %3 = alloca %struct.addrinfo*, align 8
  call void @llvm.dbg.declare(metadata i8** %1, metadata !22, metadata !25), !dbg !26
  call void @llvm.dbg.declare(metadata %struct.addrinfo* %2, metadata !27, metadata !25), !dbg !58
  call void @llvm.dbg.declare(metadata %struct.addrinfo** %3, metadata !59, metadata !25), !dbg !60
  %4 = call i8* @getpass(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i32 0, i32 0)),
  store i8* %4, i8** %1, align 8, !dbg !62
  %5 = bitcast %struct.addrinfo* %2 to i8*, !dbg !63
  call void @llvm.memset.p0i8.i64(i8* %5, i8 0, i64 48, i32 8, i1 false), !dbg !63
  %6 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %2, i32 0, i32 1, !dbg !64
  store i32 0, i32* %6, align 4, !dbg !65
  %7 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %2, i32 0, i32 2, !dbg !66
  store i32 2, i32* %7, align 8, !dbg !67
  %8 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %2, i32 0, i32 0, !dbg !68
  store i32 0, i32* %8, align 8, !dbg !69
  %9 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %2, i32 0, i32 3, !dbg !70
  store i32 0, i32* %9, align 4, !dbg !71
  %10 = load i8*, i8** %1, align 8, !dbg !72
  %11 = call i32 @getaddrinfo(i8* %10, i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.1, i32
```

No mem2reg run, so let's..

# After mem2reg

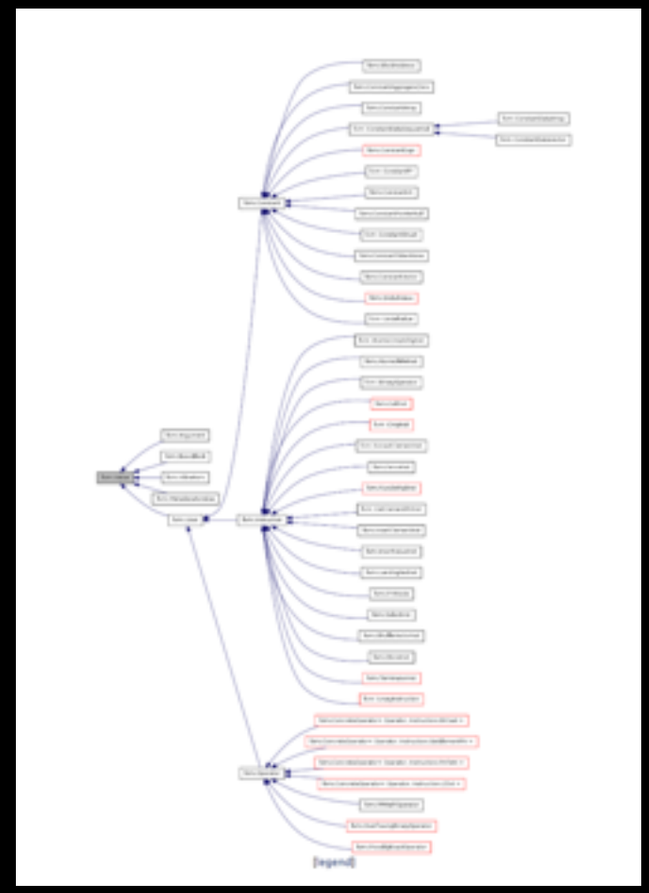
```
[awr@anathema tests] opt-3.9 -mem2reg -o NSDL001.bc < NSDL001.bc
[awr@anathema tests]
```

```
define void @leaks_passwd() #0 {
  %1 = alloca %struct.addrinfo, align 8
  %2 = alloca %struct.addrinfo*, align 8
  %3 = call i8* @getpass(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i32 0, i32 0))
  %4 = bitcast %struct.addrinfo* %1 to i8*
  call void @llvm.memset.p0i8.i64(i8* %4, i8 0, i64 48, i32 8, i1 false)
  %5 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %1, i32 0, i32 1
  store i32 0, i32* %5, align 4
  %6 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %1, i32 0, i32 2
  store i32 2, i32* %6, align 8
  %7 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %1, i32 0, i32 0
  store i32 0, i32* %7, align 8
  %8 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %1, i32 0, i32 3
  store i32 0, i32* %8, align 4
  %9 = call i32 @getaddrinfo(i8* %3, i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.1, i32 0, i
```

Just using the value as the arg

# Value class

- Base class for all operands, but others as well
  - *Function*
  - *Module*
  - *Instruction (BranchInst, AllocaInst..)*
- Provides *User* list
- *Value* has a *Type*



http://llvm.org/docs/doxygen/html/classllvm_1_1Value.html

*User* in action from mpskel

```
for (auto &F : M) { /* Iterate through all functions in this module */
  std::string fname = "not named";
  if (F.hasName()) {
    fname = F.getName().str();
  }
  if (F.user_empty()) { // If no uses, don't look further.
    errs() << "Function (" << fname << ") not used.\n";
    continue;
  }
  errs() << "Listing uses for function (" << fname << ")\n";
  for (auto uit = F.user_begin(); uit != F.user_end(); ++uit) {
    User *u = *uit;
    errs() << "   ";
    std::string pn = "";
    if (isa<CallInst>(u) || isa<InvokeInst>(u)) { // Is this use a Call or Invoke instruction?
      CallSite cs(dyn_cast<Instruction>(u)); // It is, so let's use the common class CallSite
      Function *caller = cs.getParent()->getParent(); // Instruction in a BasicBlock in a Function.
      if (caller->hasName()) {
        pn = caller->getName().str();
      } else {
        pn = "not named";
      }
      errs() << pn << ": ";
    }
    u->dump();
```

```
[awr@anathema mpskel] opt-3.9 -load built/MPSkel.so -mem2reg -mpskel ../comminute/tests/NSDL001.bc 1> /dev/null
...
Listing uses for function (getpass)
   leaks_passwd:   %3 = call i8* @getpass(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i32 0, i32 0)
...
Listing uses for function (getaddrinfo)
   leaks_passwd:   %9 = call i32 @getaddrinfo(i8* %3, i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.1, i
%struct.addrinfo** %2), !dbg !71
```

Goal: Basic tool, meant to support multiple analyses; start with:

A. find leaking of sensitive data

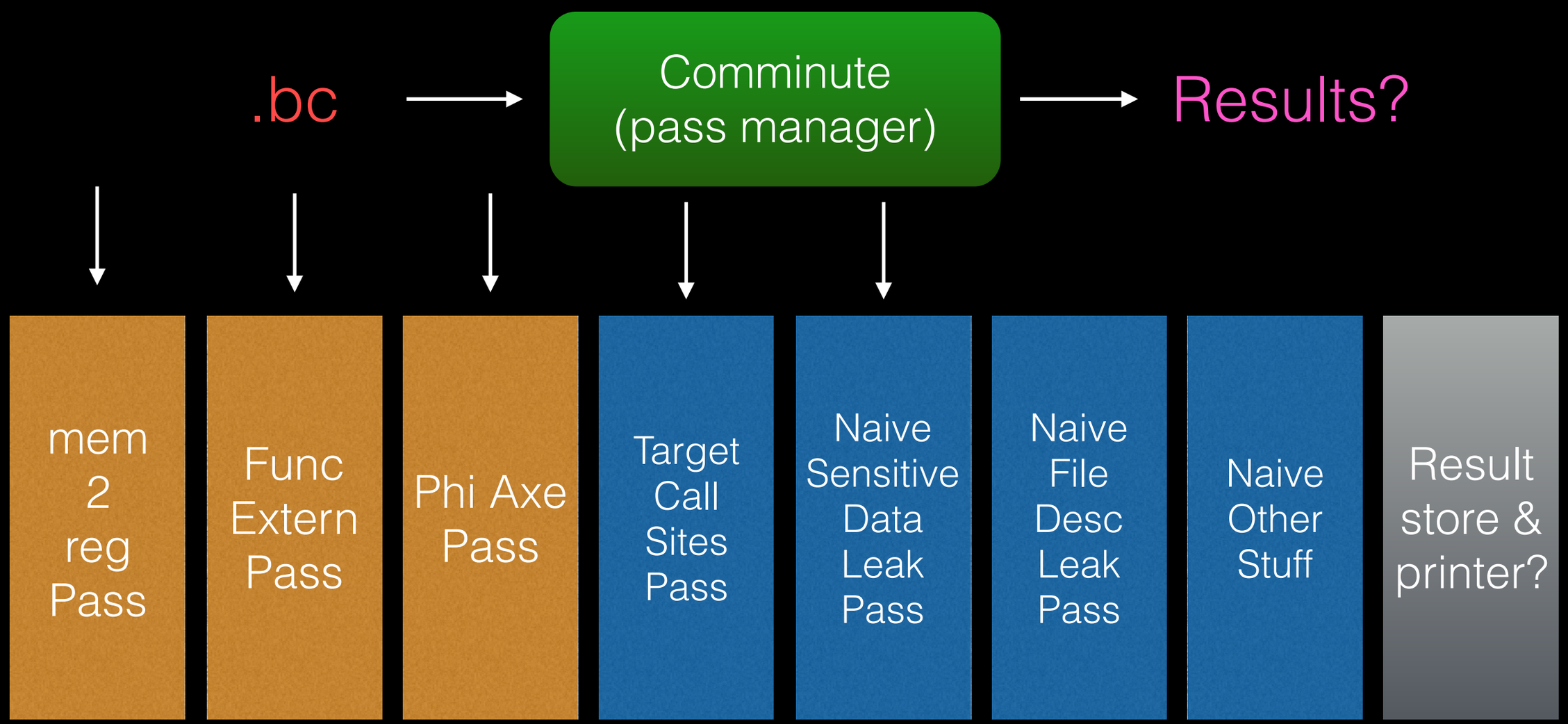
B. find leaking of file descriptors

C. use of “bad” functions

Method:

- transform code to forms easier to deal with
- find sources of such data
- find sinks of such data
- attempt some sort of reachability analysis
- pass manager to control running of passes and be a front end

# A Possible Design



# Basic PassManager

LLVM's
CommandLine 2.0 API

parseIRFile

Add in
transform passes

Setup the source and sink
gather pass and basic
sensitive leak pass

Add .bc output writer pass

Run 'em!

```
...
cl::opt<std::string> InputBitcodeFile(cl::Positional, cl::desc("<input.bc>"),
  cl::Required);
cl::opt<std::string> OutputBitcodeFile(cl::Positional, cl::desc("<output.bc>"),
  cl::Required);
cl::opt<bool> NaiveSDL("naive-sensitive-data-leak",
  cl::desc("Perform Naive Sensitive Data Leak Analysis"), cl::init(false));
...
int
main(int argc, char **argv)
{
    std::error_code ec;
    legacy::PassManager passManager;
    std::unique_ptr<Module> irModule;
    ModulePass *modPass;
    raw_fd_ostream *outputStream;

    cl::ParseCommandLineOptions(argc, argv);

    std::cout << "<C> Reading input bitcode file: " << InputBitcodeFile << "\n";
    irModule = parseIRFile(InputBitcodeFile, err,
      *unwrap(LLVMGetGlobalContext()));
    std::cout << "<C> Adding function externalizer pass.\n";
    FunctionExternalizer *fe = new FunctionExternalizer();
    fe->setFunctionListFilePath("conf/fexternalizer.txt");
    passManager.add(fe);
    passManager.add(createPromoteMemoryToRegisterPass());
    ...
    if (NaiveSDL) {
        std::cout << "<C> Adding naive sensitive data leak pass.\n";
        TargetCallSitesPass *pt = new TargetCallSitesPass();
        pt->setConfig(TargetCallSitesPass::SourceCall,
          "conf/sensitivesource.cfg");
        pt->setConfig(TargetCallSitesPass::SinkCall,
          "conf/sensitivesink.cfg");
        passManager.add(pt);
        NaiveSensitiveDataLeak *n = new NaiveSensitiveDataLeak();
        passManager.add(n);
    }
    ...
    outputStream = new raw_fd_ostream(OutputBitcodeFile, ec, sys::fs::F_None);
    passManager.add(createBitcodeWriterPass(*outputStream, false, true));
    /* Actually run the passes added on this module */
    passManager.run(*irModule.get());
    outputStream->close();
```

legacy PM

# Function Externalizer

Transform to simplify

```
FunctionExternalizer::runOnModule(Module &M)
{
  errs() << "Running function externalizer pass.\n";
  std::ifstream fileHandle(this->_functionListFile);
  std::string fnName;
  // each line is a function name to externalize. XXX 0 checking :-P
  while (std::getline(fileHandle, fnName)) {
    // skip comment line.
    if (fnName.find("#", 0) == 0) {
      continue;
    }
    // Does the function exist within this module?
    Function *f = M.getFunction(fnName);
    if (f == NULL) {
      continue;
    }
    // Definition is already outside of this module.
    if (f->isDeclaration()) {
      continue;
    }
    // Remove the body (definition) of the function. Leave declaration.
    errs() << "Deleting body of function: " << f->getName().str() << "\n";
    f->deleteBody();
  }
```

# Find TargetCallSites

Reusable Information

```
TargetCallSitesPass::parseConfig(std::string configFilePath, TargetCallType tct,
....
    for (auto memIt = mems.begin(); memIt != mems.end(); ++memIt) {
        // Does the function we'd like to check in this Module?
        std::string fnName = *memIt;
        Function *fp = M->getFunction(fnName);
...
        // Check to see if we could be this function based on arg count
        int argIdx = dict[fnName].asInt();

        // Target is a void return :-/ No dice.
        Type *rt = fp->getReturnType();
        if (argcIdx == -1 && rt->isVoidTy()) continue;
        if (argIdx != -1 && \
          (fp->arg_size() == 0 || fp->arg_size() <= (unsigned)argIdx)) continue;

        /* If no User's, then no call/invoke instruction. */
        if (fp->user_empty() == true) continue;

        /* Ok, so we have name, argument, function, and a non-empty user list */
        for (auto userIt = fp->user_begin(); userIt != fp->user_end(); ++userIt) {
            User *targUser = *userIt;
            // Just handle call/invoke's for now
            if (!isa<CallInst>(targUser) && !isa<InvokeInst>(targUser)) continue;
            Instruction *targInst = cast<Instruction>(targUser);
            std::unique_ptr<TargetCallSite> tcs(new TargetCallSite(targInst, argIdx));
            targetCallMap[tct].push_back(std::move(tcs));
...
bool
TargetCallSitesPass::runOnModule(Module &M)
{
    ...
        parseConfig(p, t, &M);
    ...
    return false;
}
```

Is function in this module?

Weak signature check

skip unused functions

Only care about invokes

# Sensitive Leak Pass

Use info for analysis

```
void
NaiveSensitiveDataLeak::getAnalysisUsage(AnalysisUsage &AU) const
{
    AU.addRequired<TargetCallSitesPass>();
}
```

Inform PassManager this pass relies on TargetCallSitesPass

```
bool
NaiveSensitiveDataLeak::runOnModule(Module &M)
{
    errs() << "Running naive sensitive data leak pass\n";
    TargetCallSitesPass &p = getAnalysis<TargetCallSitesPass>();

    if (p.src_empty()) {
        return false;
    }
    if (p.snk_empty()) {
        return false;
    }
...
```

Get reference to the pass that ran
&
make use of it's API for results

# Check if sensitive Values are arguments to leaky functions

For each sink, get ptr to leaked Value

For each source we have, compare leaked Value with sourced Value

```
for (auto snkIt = p.snk_begin(); snkIt != p.snk_end(); ++snkIt) {
    TargetCallSite *snkSite = &*snkIt->get();
    Value *leakData = snkSite->getTarget();
    auto srcIt = p.src_end();
    --srcIt;
    bool brk_back = false;
    for (; brk_back == false; --srcIt) {
        if (srcIt == p.src_begin()) {
            brk_back = true;
        }
        TargetCallSite *srcSite = &*srcIt->get();
        Value *originalSourceData = srcSite->getTarget();
        Value *sourceData = originalSourceData;
        if (isa<CallInst>(leakData) || isa<InvokeInst>(leakData)) {
            if (leakData == sourceData) {
                printResult(srcSite, snkSite);
                break;
            }
        }
    }
}
```

```
[awr@anathema comminute] build/bin/Comminute -naive-sensitive-data-leak tests/NSDL001.bc foo.bc
<C> Reading input bitcode file: tests/NSDL001.bc
<C> Adding function externalizer pass.
<C> Adding mem2reg pass.
<C> Adding constant propagation passes.
<C> Adding naive sensitive data leak pass.
<C> Adding bitcode writer pass
<C> Running passes
Running function externalizer pass.
Running target call sites pass.
Running naive sensitive data leak pass
  ! sensitive data leak
     leaks_passwd calls getaddrinfo where arg idx #0 is tainted sensitive. file: NSDL001.c line: 28
        source: leaks_passwd calls getpass at line: 20 of file: NSDL001.c
<C> Finished...
```

Ok… that *was* quite the contrived sensitive data leak :P

What about Phi nodes?

```
.eaks_passwd(unsigned lookup)
:
    char *p, *a2l = "www.cw-complex.com";
    struct addrinfo hints, *result;
    if (lookup) {
        p = getpass("enter passwd: ");
    } else {
        p = a2l;
    }
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = 0;
    hints.ai_protocol = 0;
    (void)getaddrinfo(p, "http", &hints, &result);  /* which path did p take? */
```

Slightly different C

```
define void @leaks_passwd(i32) #0 {
  %2 = alloca %struct.addrinfo, align 8
  %3 = alloca %struct.addrinfo*, align 8
  %4 = icmp ne i32 %0, 0
  br i1 %4, label %5, label %7

; <label>:5:                                      ; preds = %1
  %6 = call i8* @getpass(i8* getelementptr inbounds ([15 x i8], [15 x i8]* @.str.1, i32 0, i32 0))
  br label %8

; <label>:7:                                      ; preds = %1
  br label %8

; <label>:8:                                      ; preds = %7, %5
  %.0 = phi i8* [ %6, %5 ], [ getelementptr inbounds ([19 x i8], [19 x i8]* @.str, i32 0, i32 0), %7 ]
  %9 = bitcast %struct.addrinfo* %2 to i8*
  call void @llvm.memset.p0i8.i64(i8* %9, i8 0, i64 48, i32 8, i1 false)
  %10 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %2, i32 0, i32 1
  store i32 0, i32* %10, align 4
  %11 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %2, i32 0, i32 2
  store i32 2, i32* %11, align 8
  %12 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %2, i32 0, i32 0
  store i32 0, i32* %12, align 8
  %13 = getelementptr inbounds %struct.addrinfo, %struct.addrinfo* %2, i32 0, i32 3
  store i32 0, i32* %13, align 4
  %14 = call i32 @getaddrinfo(i8* %.0, i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.2, i32 0,
```

IR output (w/ mem2reg)

Obviously fails! Values don’t match.

```
[awr@anathema comminute] build/bin/Comminute -naive-sensitive-data-leak tests/NSDL003.bc foo.bc
<C> Reading input bitcode file: tests/NSDL003.bc
<C> Adding function externalizer pass.
<C> Adding mem2reg pass.
<C> Adding constant propagation passes.
<C> Adding naive sensitive data leak pass.
<C> Adding bitcode writer pass
<C> Running passes
Running function externalizer pass.
Running target call sites pass.
Running naive sensitive data leak pass
<C> Finished...
```

# Axe Method

- Goal: remove use of PHINode value “hurting” our analysis
- Axe:

```
cl::opt<int> ChoosePhiValuePass("choose-phi-value",
  cl::desc("Choose value to use from PhiNode (defaults to first)"),
  cl::init(-1));
```

  - Choose incoming branch for ALL PHINodes
  - Replace all those uses with chosen branch Value
  - Attempts to remove basic blocks for no longer used branches
- *Awfulness*: no real reasoning going on, just elimination of edges and nodes of a graph :(

```
for (auto &f : M) {
  std::vector<std::pair<PHINode *, Value *>> replaceList;
  for (auto ii = inst_begin(f); ii != inst_end(f); ++ii) {
      Instruction *in = &*ii;
      if (PHINode *pn = dyn_cast<PHINode>(in)) {
        unsigned usedEdge = edgeIndex;
        if (pn->getNumIncomingValues() <= edgeIndex) {
          errs() << "Not enough incoming values...using 0\n";
          usedEdge = 0;
        }
        Value *x = pn->removeIncomingValue(usedEdge, false);
        replaceList.push_back(std::make_pair(pn, x));
      }
  }
  if (replaceList.empty() == false) {
    rv = true;
  }
  for (auto pc : replaceList) {
    /* Replace all uses of the PHINode with the selected Value */
    pc.first->replaceAllUsesWith(pc.second);
    while (pc.first->getNumIncomingValues() > 0) {
      Value *d = pc.first->removeIncomingValue((unsigned)0, false);
      /* Each instruction resides in a BasicBlock */
      assert(isa<Instruction>(d) == true);
      Instruction *vi = cast<Instruction>(d);
      BasicBlock *bb = vi->getParent();
      if (bb->user_empty()) {
        bb->eraseFromParent();
        continue;
      }
      /* Attempt to remove users of BasicBlock so we can axe it */
      attemptUserReduction(bb);
      if (bb->user_empty()) {
        bb->eraseFromParent();
        continue;
      }
    }
    assert(pc.first->users_empty());
    pc.first->eraseFromParent();
  }
```

Select incoming branch to use

Replace all PHINode uses with branch's Value

Attempt to remove stale Basic Blocks (See code)

```
[awr@anathema comminute] build/bin/Comminute -naive-sensitive-data-leak -choose-phi-value 0 tests/NSDL003.bc foo.bc
<C> Reading input bitcode file: tests/NSDL003.bc
<C> Adding function externalizer pass.
<C> Adding mem2reg pass.
<C> Adding constant propagation passes.
<C> Adding phi value selector pass
<C> Using edge index: 0
<C> Adding naive sensitive data leak pass.
<C> Adding bitcode writer pass
<C> Running passes
Running function externalizer pass.
Running choose phi value pass.
Running target call sites pass.
Running naive sensitive data leak pass
  ! sensitive data leak
     leaks_passwd calls getaddrinfo where arg idx #0 is tainted sensitive. file: NSDL003.c line: 30
        source: leaks_passwd calls getpass at line: 21 of file: NSDL003.c
<C> Finished...
```

```
[awr@anathema comminute] build/bin/Comminute -naive-sensitive-data-leak -choose-phi-value 1 tests/NSDL003.bc foo.bc
<C> Reading input bitcode file: tests/NSDL003.bc
<C> Adding function externalizer pass.
<C> Adding mem2reg pass.
<C> Adding constant propagation passes.
<C> Adding phi value selector pass
<C> Using edge index: 1
<C> Adding naive sensitive data leak pass.
<C> Adding bitcode writer pass
<C> Running passes
Running function externalizer pass.
Running choose phi value pass.
Running target call sites pass.
Running naive sensitive data leak pass
<C> Finished...
```

# **Much** more to worry about

- Code provided relies on simplicity, but that's a joke.
- Want to reason about:
  - execution order  / control flow (or make assumptions)
  - pointers (i.e. aliasing)
  - inter-procedural analysis
- **T**hat is where a lot of research is being done…
  - see SVF project and how they handle value flow tracking
  - See ValueFlow API
  - See Andersen's Alias Analysis for pointer reasoning


Time for next section?

…if not… please view slides via git…

# Small Example: IntFlip

- Find function calls w/ at least 1 argument is an integer
- RT replace int used w/ different value with probability 1-*p*
- Use *arc4random*() with a given *mean* value…
- Either return random value or bit flip of original

# Lift a constant to variable

Instruction visitor to reach all Call and Invoke

is argument a ConstantInt?

Get space for it
Save constant value to it
Replace argument to fn call

```
visitCallSite(CallSite callSite)
{
  unsigned numArgOps = callSite.getNumArgOperands();
  unsigned argIdx;

  for (argIdx = 0; argIdx < numArgOps; argIdx++) {
    Value *va = callSite.getArgOperand(argIdx);

    if (ConstantInt *con = dyn_cast<ConstantInt>(va)) {
      unsigned nBits = con->getBitWidth();
      AllocaInst *localized__alloc = new AllocaInst(
        IntegerType::get(callSite.getParent()->getContext(), nBits),    // type to allocate
        "__intflip_localized",  // give the slot a label
        callSite.getInstruction());     // Insert before call instruction
      StoreInst *localized__store = new StoreInst(
        con,    // value to store
        localized__alloc,  // where to store it
        callSite.getInstruction());     // Insert before call instruction
      LoadInst *localized__load = new LoadInst(
        localized__alloc,       // pointer to load from
        (const char *)"__intflip_loaded",       // label the slot
        callSite.getInstruction());     // Insert before call instruction
      /* replace the constant in the function call */
      callSite.setArgument(argIdx, localized__load);
      new_vars++;
      modified = true;
    }
  }
}
```

** This is from the *intflip* code

# Add Existing Function (libc)

- Lookup|insert declaration:

```
Constant *lookupRand = M.getOrInsertFunction("arc4random", Type::getInt32Ty(ctx), NULL);
```

Find function named “arc4random” with return type int32 and no arguments

** This is from the *intflip* code

C representation

```
int8_t
__bitflip_randomizer_i8__(int8_t inArg0)
{
  unsigned rv = arc4random();
  if (rv <= (unsigned)2^31) {
    rv = 1 << (rv % 8);
    return inArg0 ^ rv;
  }
  return inArg0;
}
```

IR version

```
define i8 @__bitflip_randomizer_i8__(i8 %intToFlip, i32 %meanValue) {
entry:
  %__bf_rand_ = call i32 @arc4random()
  %__bf_lessthan_ = icmp ule i32 %__bf_rand_, %meanValue
  br i1 %__bf_lessthan_, label %bf_it, label %return

bf_it:                                            ; preds = %entry
  %__bf_bitflip_ = urem i32 %__bf_rand_, 8
  %__bf_cast_randrem_ = trunc i32 %__bf_bitflip_ to i8
  %__bf_shifted_bit_ = shl i8 1, %__bf_cast_randrem_
  %__bf_xord_retval_ = xor i8 %intToFlip, %__bf_shifted_bit_
  ret i8 %__bf_xord_retval_

return:                                           ; preds = %entry
  ret i8 %intToFlip
}
```

# LLVM generating the IR

```
/* declare i8 __bitflip_randomizer_i8__(i8, i32) */
std::string int8_rand = "__bitflip_randomizer_i8__";
Constant *cTmp = M.getOrInsertFunction(int8_rand,
    Type::getInt8Ty(ctx),                // return
    Type::getInt8Ty(ctx),                // arg 0
    Type::getInt32Ty(ctx),               // arg 1
    NULL);
Function *bf_i8 = cast<Function>(cTmp);
bf_i8->setCallingConv(CallingConv::C);
....
BasicBlock *blkEntry = BasicBlock::Create(ctx, "entry", bf_i8);
BasicBlock *blkBitFlipIt = BasicBlock::Create(ctx, "bf_it", bf_i8);
BasicBlock *blkReturn = BasicBlock::Create(ctx, "return", bf_i8);
/*
 * entry:
 *  %__bf_rand_ = call i32 @arc4random()
 *  %__bf_lessthan_ = icmp ule i32 %__bf_rand_, %meanValue
 *  br i1 %__bf_lessthan_, label %bf_it, label %return
 */
IRBuilder<> builder(blkEntry);
Value *callArc4Random = builder.CreateCall(fnRand, None, "__bf_rand_", nullptr);
Value *lessThan = builder.CreateICmpULE(callArc4Random, &inArg1, "__bf_lessthan_");
Value *branchBitFlip = builder.CreateCondBr(lessThan, blkBitFlipIt, blkReturn);
/*
 * bf_it:                                        ; preds = %entry
 *  %__bf_bitflip_ = urem i32 %__bf_rand_, 8
 *  %__bf_cast_randrem_ = trunc i32 %__bf_bitflip_ to i8
 *  %__bf_shifted_bit_ = shl i8 1, %__bf_cast_randrem_
 *  %__bf_xord_retval_ = xor i8 %intToFlip, %__bf_shifted_bit_
 *  ret i8 %__bf_xord_retval_
 */
builder.SetInsertPoint(blkBitFlipIt);
Value *randModulus = ConstantInt::get(IntegerType::get(ctx, 32), nBits, false);
Value *randRemainder = builder.CreateURem(callArc4Random, randModulus,
    "__bf_bitflip_");
Value *defaultBit = ConstantInt::get(IntegerType::get(ctx, nBits), 1, false);
Value *castRandRem = builder.CreateZExtOrTrunc(randRemainder,
    Type::getInt8Ty(ctx), "__bf_cast_randrem_");
Value *shiftedBit = builder.CreateShl(defaultBit, castRandRem,
    "__bf_shifted_bit_");
Value *xordReturnVal = builder.CreateXor(&inArg0, shiftedBit, "__bf_xord_retval_");
builder.CreateRet(xordReturnVal);
/*
 * return:                                       ; preds = %entry
 *  ret i8 %intToFlip
 */
builder.SetInsertPoint(blkReturn);
builder.CreateRet(&inArg0);
```

Declare function

Add some basic blocks

Use IRBuilder to add to blkEntry

re-use same IRBuilder

** This is from the *intflip* code

# Say we have code:

```
char
bf8(char in)
{
        return in;
}
```

```
static void *
thd_bf8(void *a)
{
        for (unsigned i = 0; i < MAXITER; i++)
                printf("thd_bf8: %d\n", bf8(0));
        return a;
}
```

# Inject call to new function

```
for (inst_iterator I = inst_begin(f), E = inst_end(f); I != E; ++I) {
    if (isa<CallInst>(&*I) || isa<InvokeInst>(&*I)) {
        CallSite cs(&*I);
        Function *called = cs.getCalledFunction();
        if (!called->hasName()) {
            continue;   // XXX Currently require functions to have names
        }
       ....
        unsigned numArgOps = cs.getNumArgOperands();
        for (unsigned ii = 0; ii < numArgOps; ii++) {
            Value *va = cs.getArgOperand(ii);
            Type *ta = va->getType();
            /*
             * If not a 8, 16, 32, or 64 bit integer, we skip it.
             */
            if (TypeValueSupport::isReplaceable(ta, va) == false) {
                continue;
            }
          ....
            unsigned nBits = ta->getIntegerBitWidth();
            Function *insertedRndFn = M.getFunction(rndFnName);
            assert(insertedRndFn != NULL);
            ConstantInt *mn = ConstantInt::get(M.getContext(), APInt(32, mFc.mean, false));
            /*
             * Insert call to randomizer with input integer and a mean value.
             * It will be inserted before the CallInst.
             */
            CallInst *callNewFunc = CallInst::Create(insertedRndFn,
                { va, mn }, // Arguments are the integer to maybe flip and the mean value
                "__rnd_replicant_",
                cs.getInstruction());   // insert our call to the rnd fn before the targeted call instruction
            cs.setArgument(ii, callNewFunc);

    }
```

** This is from the *intflip* code

# Performs following transform

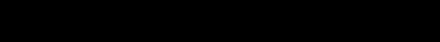
```
; <label>:7:                                     ; preds = %4
  %8 = call signext i8 @bf8(i8 signext 0)
  %9 = sext i8 %8 to i32
```



```
; <label>:7:                                     ; preds = %4
  %__intflip_localized = alloca i8
  store i8 0, i8* %__intflip_localized
  %__intflip_loaded = load i8, i8* %__intflip_localized
  %__rnd_replicant_ = call i8 @__bitflip_randomizer_i8__(i8 %__intflip_loaded, i32 100000000)
  %8 = call signext i8 @bf8(i8 signext %__rnd_replicant_)
  %9 = sext i8 %8 to i32
```

# And running…

```
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd bf8: 0
```



```
thd_bf8: 32
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 0
thd_bf8: 16
thd_bf8: 0
```

# Concluding Remarks

- Re-usability and modularity in developing security tools is essential
- Common platform makes research and sharing much easier
- LLVM provides robust APIs for doing both research & development
- Many (important?) groups are using LLVM…

Now, please go do great research! 🤓

# Some references


1. The LLVM Compiler Infrastructure, http://www.llvm.org
2. Lattner, “LLVM: An Infrastructure for Multi-Stage Optimization”, http://llvm.org/pubs/2002-12-LattnerMSThesis.html
3. “Writing an LLVM Pass”, http://llvm.org/docs/WritingAnLLVMPass.html
4. CommandLine 2.0 Library, http://llvm.org/docs/CommandLine.html
5. LLVM Language Reference, http://llvm.org/docs/LangRef.html
6. http://llvm.org/docs/CommandGuide/
7. http://llvm.org/docs/doxygen/html/index.html


Please see the projects github link for more references. Effectively all the comments related to sec r&d projects using llvm are based on those sources and reading code.