Introduction to Static Program Analysis, Among Other Things

Basics of Reverse Engineering Winter 2022

Margin Research

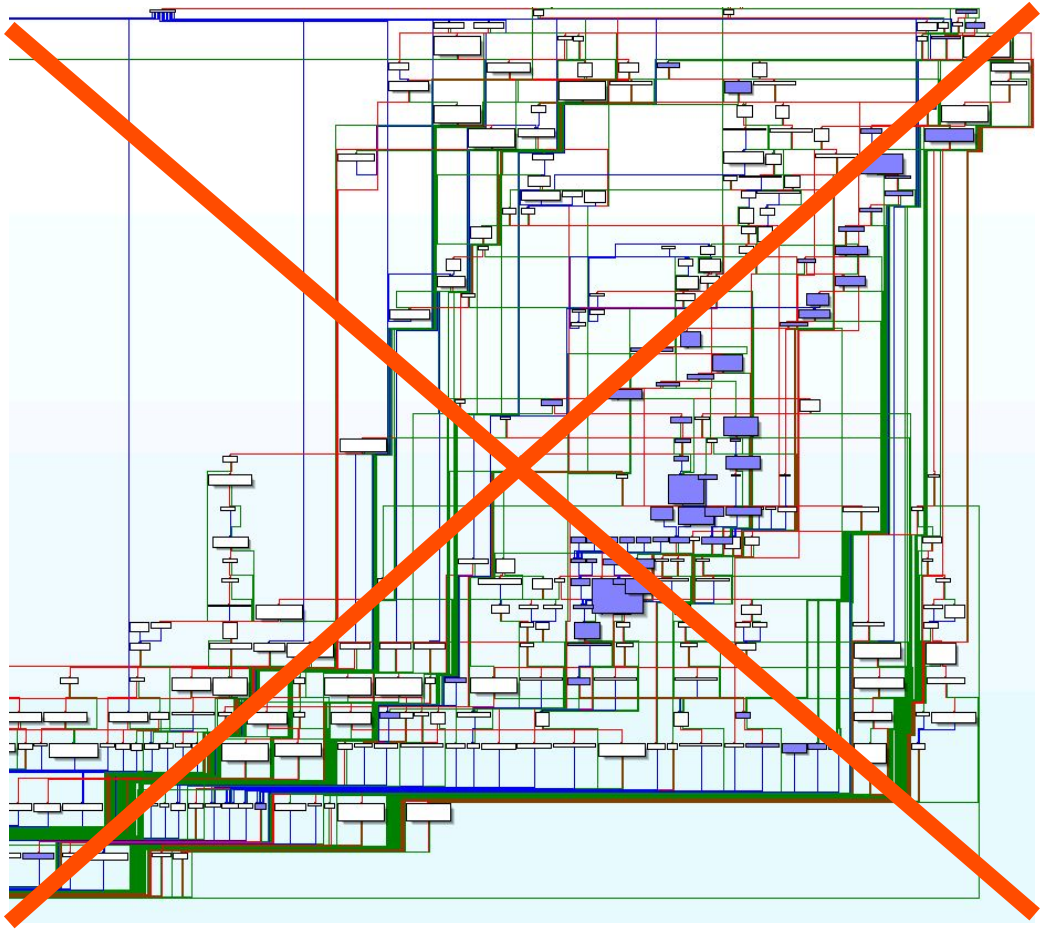
- Based in New York City
- Cleared team of Cybersecurity experts
- R&D to develop security products and tools to assist with secure development and vulnerability research
- Expertise in program analysis research and low level systems

MARGIN RESEARCH

Sophia d'Antoine

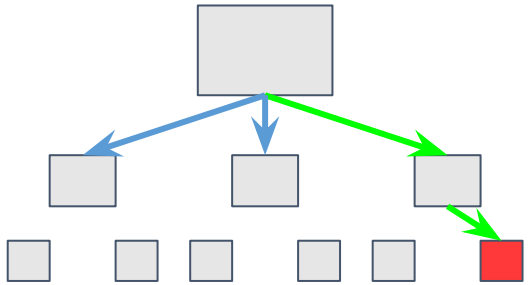
- Founder of Margin Research
- Cited by Google for Specter/ Meltdown
- Previously worked at the NSA
- Member of NATO CYDEF advising committee


step...
step...
step...
step...
step...
step...
step...
step...
step...
step...



Automation

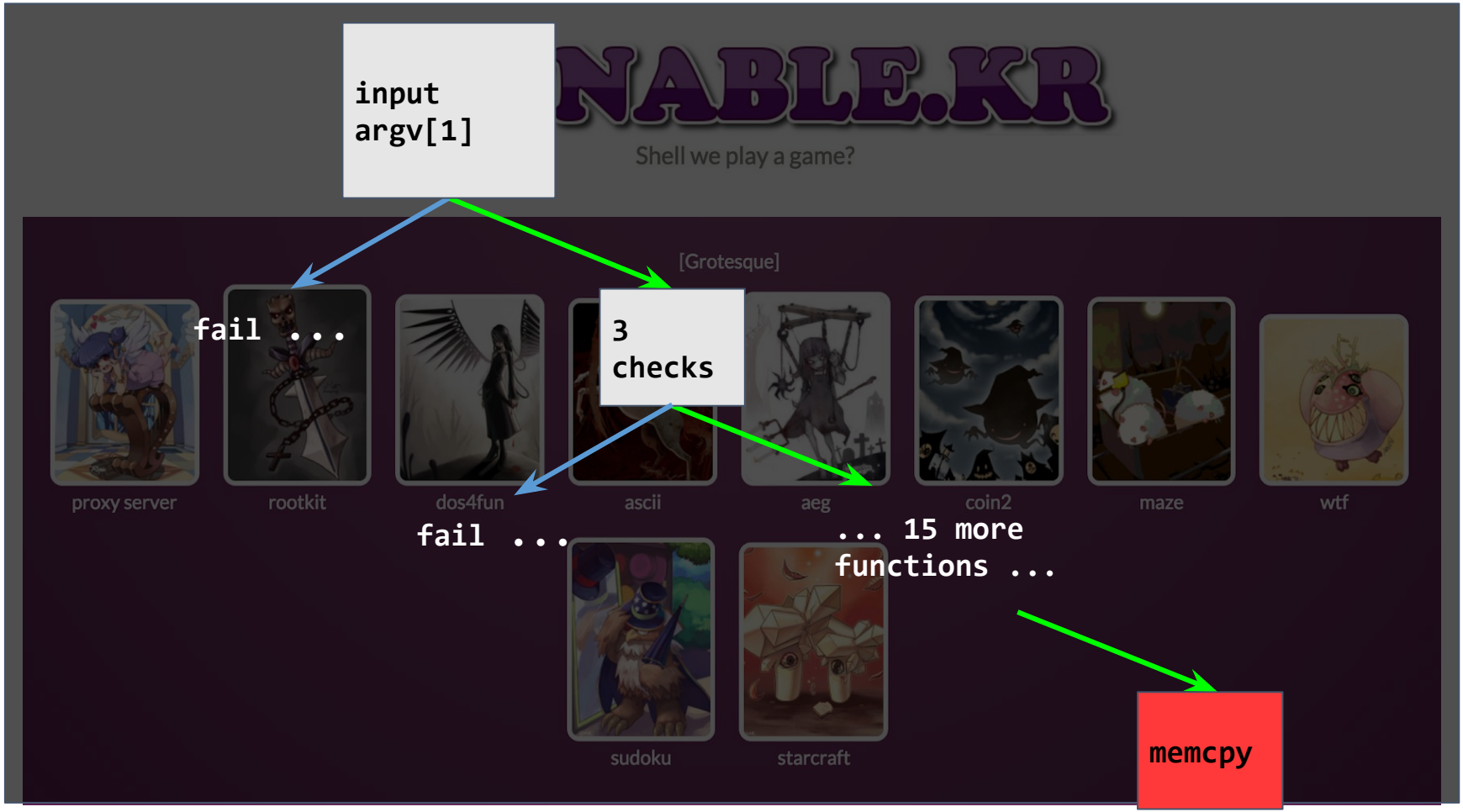
1. Find bugs
2. Solve path for reachability
3. Generate input to trigger the bug
4. Generate exploit to achieve rwx primitives
5. Integrate payload
6. Continuation of execution

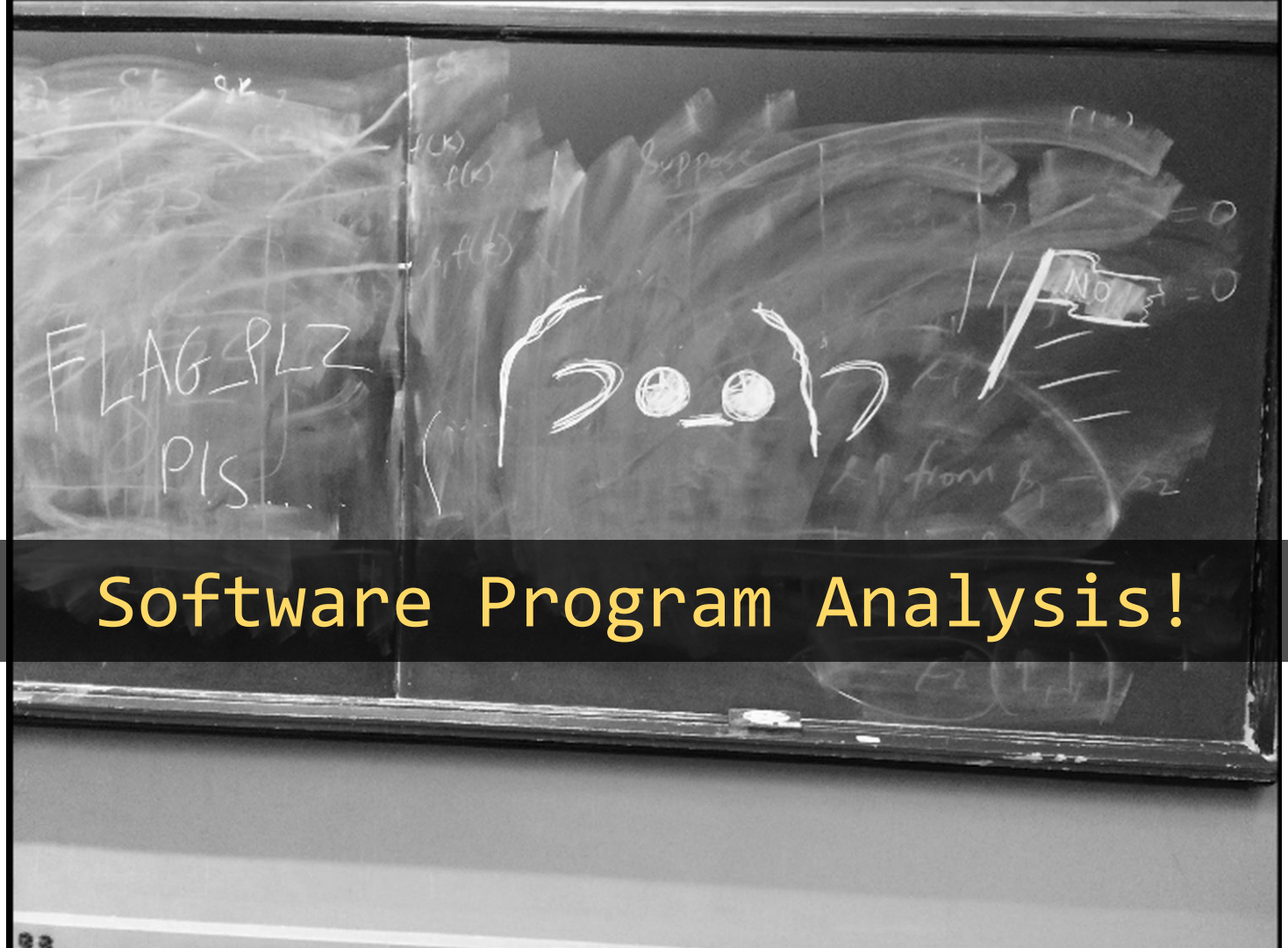


Automation

The ability to generate a successful computer attack with reduced or entirely without human interaction

- Focus on discovery and combination of write and read primitives
- Existing AE work focused on Restricted Models:
  - Sean Heelan's "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities"
  - David Brumley (@ Carnegie Mellon) et al. (AEG, MAYHEM, etc)
  - Cyber Grand Challenge! (CGC)


input
argv[1]
NABLE.KR
Shell we play a game?
[Grotesque]
fail ...
3
checks
proxy server
rootkit
dos4fun
ascii
aeg
coin2
maze
wtf
fail ...
... 15 more
functions ...
sudoku
starcraft
memcpy



Software Program Analysis!

Program Analysis

The process of automatically analyzing the behavior of applications

In terms of a property:

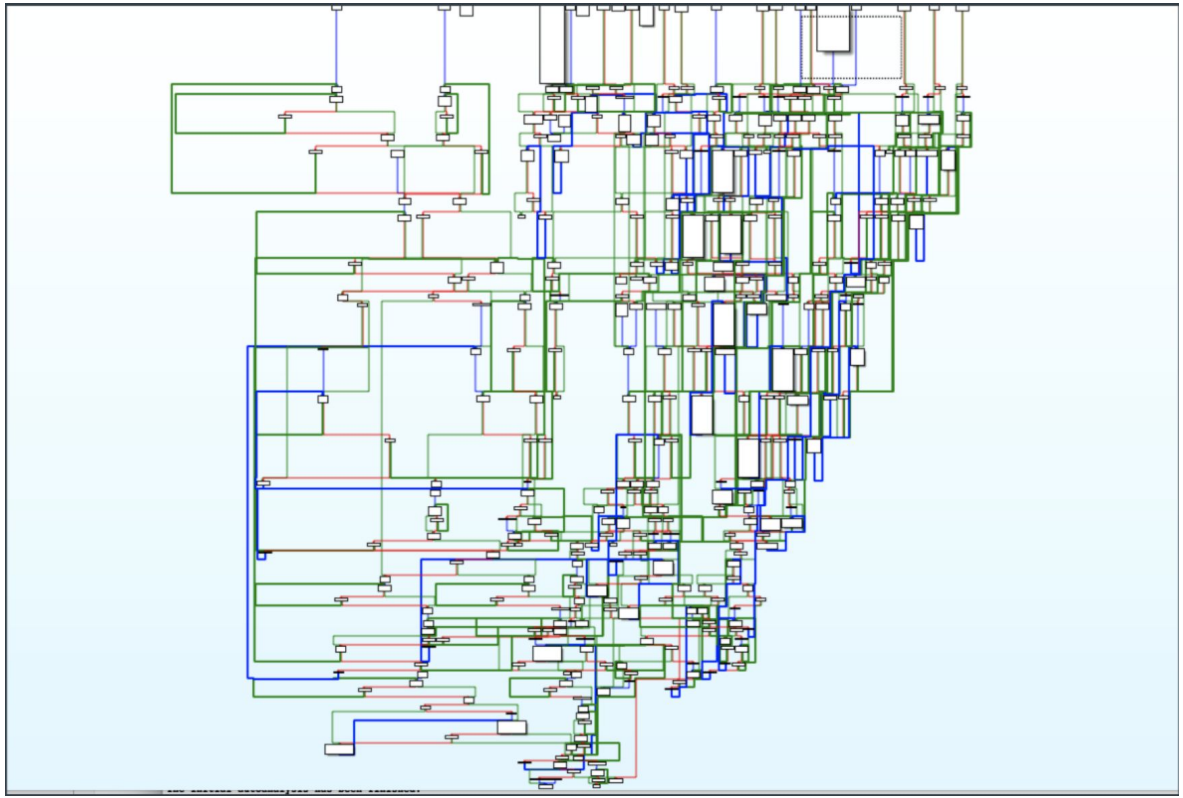
- Program correctness: set of paths == expected paths
- Program optimization: minimum expense => expected paths

Compiler Theorists were decades ahead of us!

- Even decomplication is just a spin off on optimization

Win

- Find a lot of bugs
- Find a lot of bugs faster
- Find a lot of better bugs faster



Normalization

Why?

- In order to ask generic property questions.
- A syntactically simpler language
  - Programs translated into a intermediate representation

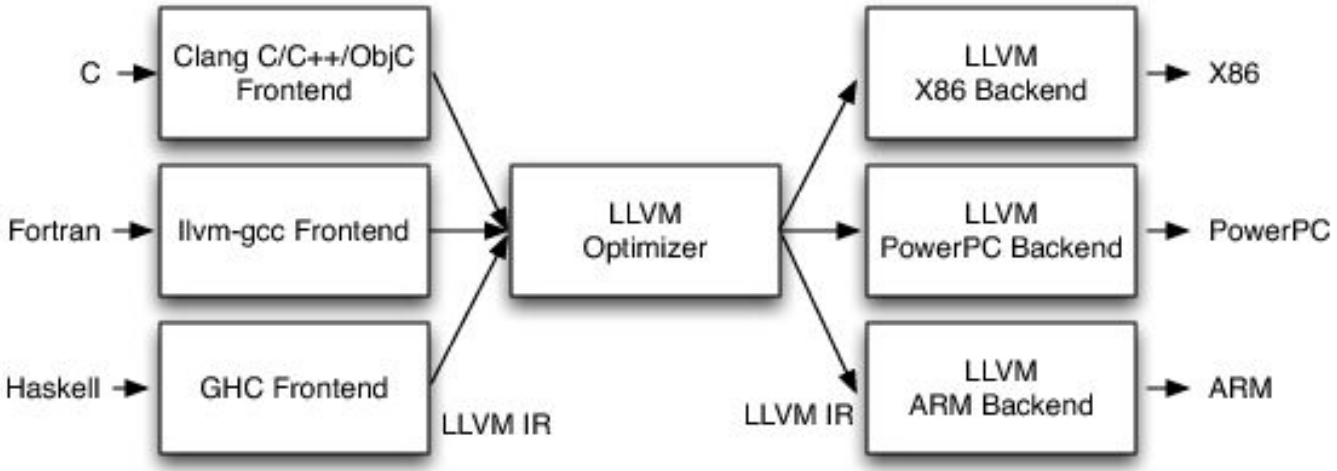
Example

x = f(y+3)*5;

Example

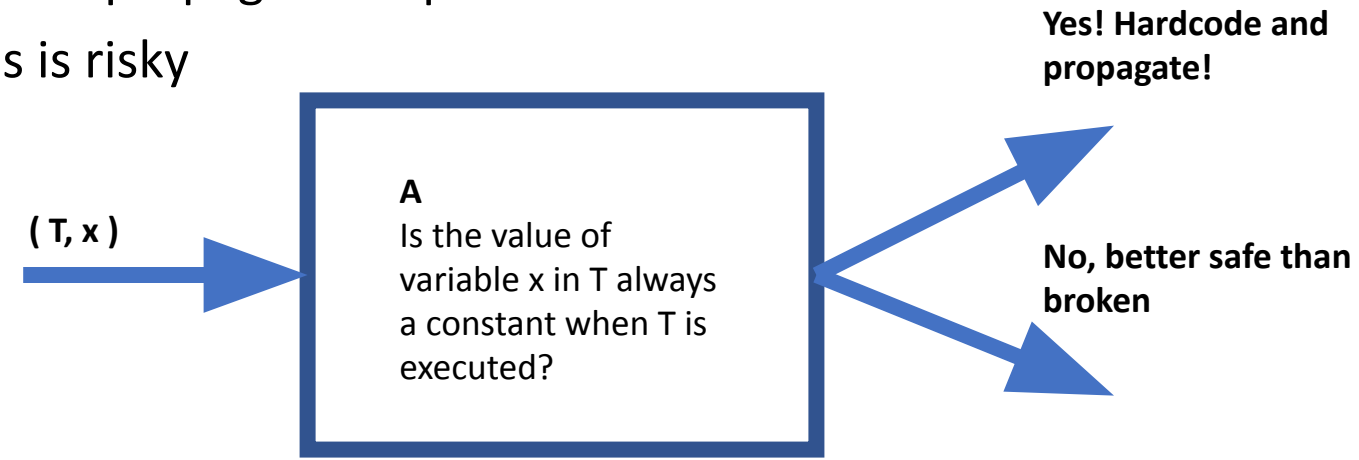
```
x = f(y+3)*5;


t1 = y+3;
t2 = f(t1);
x = t2*5;
```


C
Clang C/C++/ObjC Frontend
Fortran
llvm-gcc Frontend
Haskell
GHC Frontend
LLVM IR
LLVM Optimizer
LLVM IR
LLVM X86 Backend
X86
LLVM PowerPC Backend
PowerPC
LLVM ARM Backend
ARM

Correctness is Everything

- Analysis conservative with respect to the semantics the language and system
- Under-approximation is required for *Optimizers*
- Perform constant propagation optimization
- Answering yes is risky



Simple Vulnerability Script Over MLIL

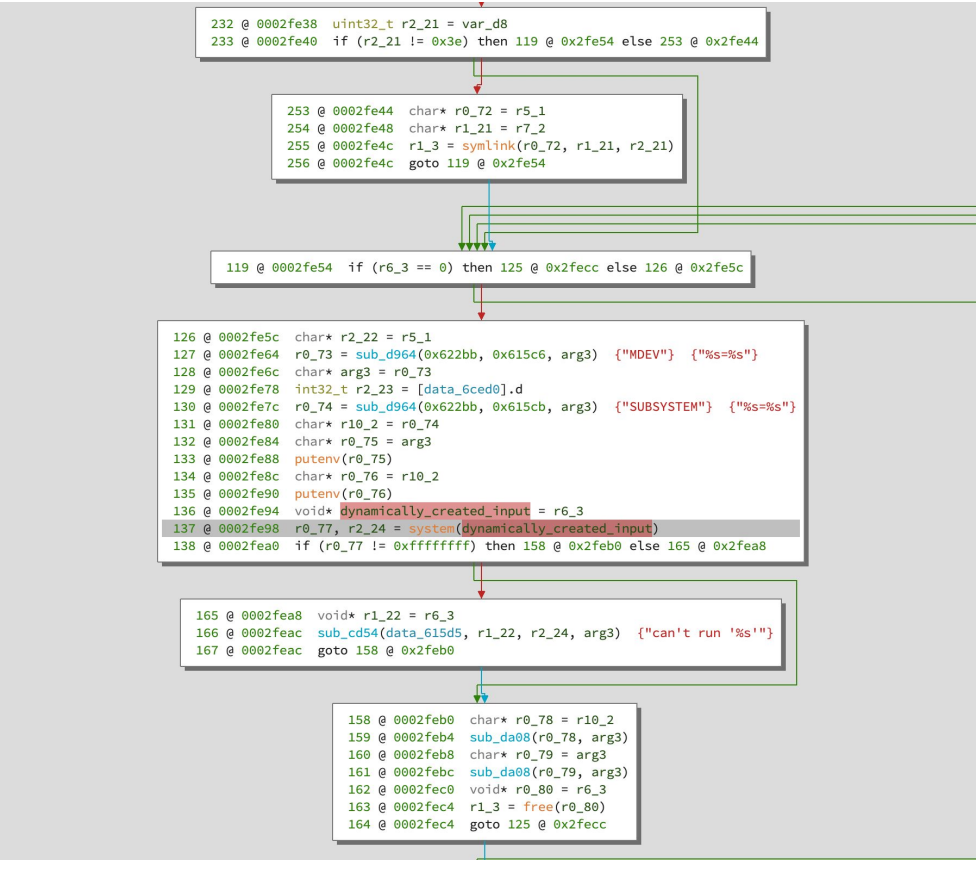
- Library functions with known behaviors
  - system
  - printf
- Rule out non-vulnerable calls
- Check for constants or not

```
system(data_080808);
printf(data_090909);
```

Example Busybox

- Simple program which rules out all safe printf/system calls
  - Goal: find vulnerable or potentially vulnerable places to look
  - Helps narrow down attack surface
- Returns unprovable calls which are false positives
- Analysis encoded with understanding of what is “incorrect”
  - system(dynamically_created_input)
- Find all call sites and “Ask” if the function use is provably safe
  - over-approximation

Binary Ninja Analysis: Bugs in system()



Binary Ninja Analysis: printf()

Stack Clash Vulnerability

- Uncontrolled memory allocation vulnerability
- User controlled size value is allocated in the stack
- If the size provided by the guest is more than the total size of the stack, it is possible to shift the Stack Pointer (RSP) into other regions of process memory

- Qualys published this bug class in 2017
- https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt

Stack Clash Vulnerability

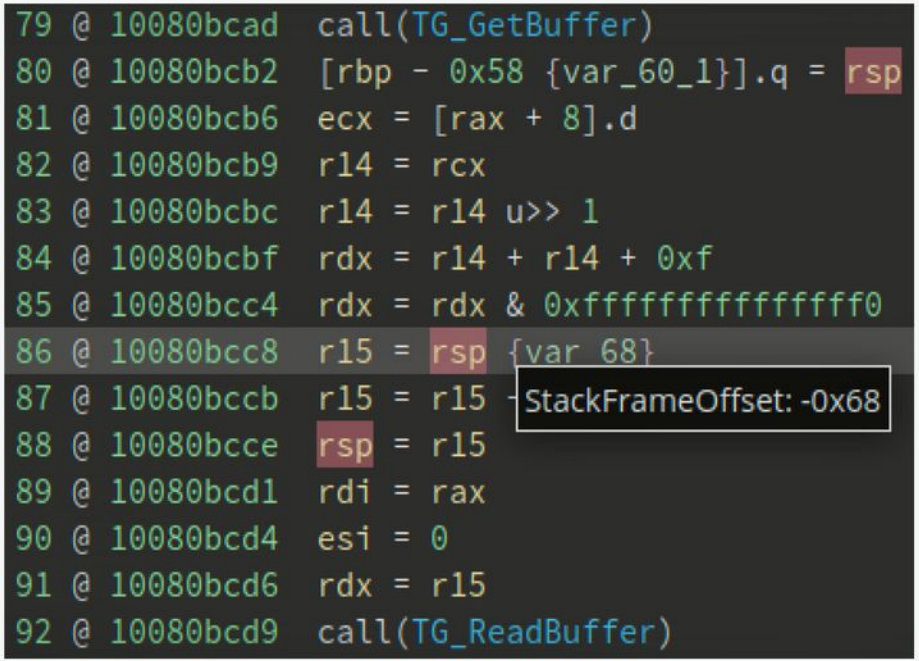
But… Compilers!

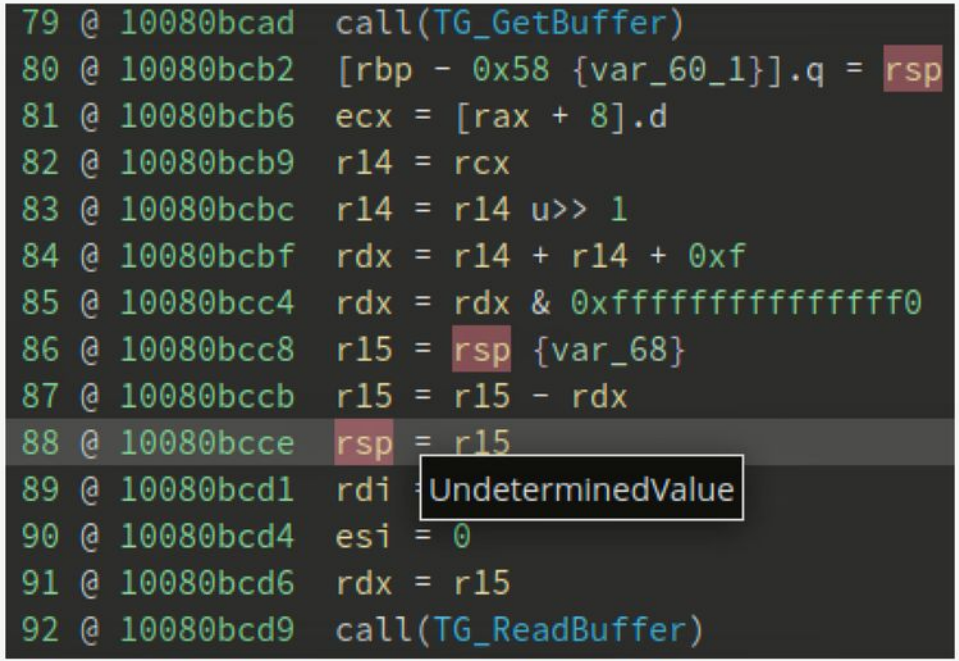
- Stack Clash mitigation in Apple's Clang
- If alloc() is used, Clang adds "`___chkstk_darwin()`"
- If too much stack space is requested, safe crash

So either

- explicitly disabled using -fno-stack-check
- setting -mmacosx-version-min=10.1X

Stack Clash Vulnerability





Stack Clash Vulnerability

```
>>> tg = bv.get_symbol_by_raw_name("TG_ReadBuffer")
>>> for ref in bv.get_code_refs(tg.address):
...     function = ref.function
...     il = function.get_low_level_il_at(ref.address)
...     if il is not None:
...         rsp = il.get_possible_reg_values("rsp")
...         if rsp.type == RegisterValueType.UndeterminedValue:
...             print(hex(ref.address))
```

Undecidability & Program Correctness

Why?

- What does undecidability mean for problems real people (not just CS theorists) care about?
- What do you do when your application's requirements require "solving" an undecidable problem?

Reaping the Benefits

- You want a compiler that finds the fastest possible machine code for a given program.
- You have JavaScript, with some high and low security variables. You want to make sure high security information isn't reachable.
- You have a parser for your programming language. You change it, but want to verify parses all the programs it used to.
- You have an anti-virus program, and wonder if it can execute a malicious instruction.
- We have vulnerability primitives. We want to prove it does not exist in the program.

Actually the halting problem.

Parsing C++ is literally undecidable

August 24, 2013

Many programmers are aware that C++ templates are Turing-complete, and this was proved in the 2003 paper C++ Templates are Turing Complete.

Dijkstra

Program testing
can be used to show the presence of bugs,
but never to show their absence

Existence of Bugs: Undecidable?

- Rice's Theorem 1951
  - *"All interesting questions about the behavior of programs are undecidable"*
  - So we can never programmatically find bugs?
- But wait… **approximative** answers still possible
- ~~Decide a property for any analyzed program~~
- Approximate answers for most realistic programs
- Continuously build towards more precise approximations

Exercise 1: C Pointers

- Theory: Approximation is useful for finding bugs in programs
- C Language Pointer Questions
  - NULL Dereferences
  - Dangling Pointers
  - Leaking Memory
  - Unintended Aliases
- Ordinary compilers fail to find or notify the program with answers to these questions

```
int main(int argc, char *argv[]) {
    if (argc == 42) {
        char *p,*q;
        p = NULL;
        printf("%s",p);
        q = (char *)malloc(100);
        p = q;
        free(q);
        *p = ’x’;
        free(p);
        p = (char *)malloc(100);
        p = (char *)malloc(100);
        q = p;
        strcat(p,q);
        assert(argc > 87);
} }
```

Approximation Constraints

- Conservative means the analysis is safe
- Errors in analysis of same class, depending on intention
- Incorporate real world restraints into analysis
- Example:

  Integers on a 32 bit machine should not be greater than 32 Bytes

Approximation Constraints

- Related to the concept of *soundness* of program analyzers
- Sound means we never get incorrect results
- Sometimes over approximation is better than under approximation
  - Example: Analysis can not prove a free-d pointer is not used again.
- Verbiage opposite between verification tools and testing tools
  - False positives better for bug hunting

Model Checking

Definition:

- Checking finite state machine properties
- No actual execution
- Prove program meets specification (formal verification)

Popular Uses:

- Automatically verifying correctness
- Prove a run bug free
- Symbolic model checking (bounds the state explosion problem)

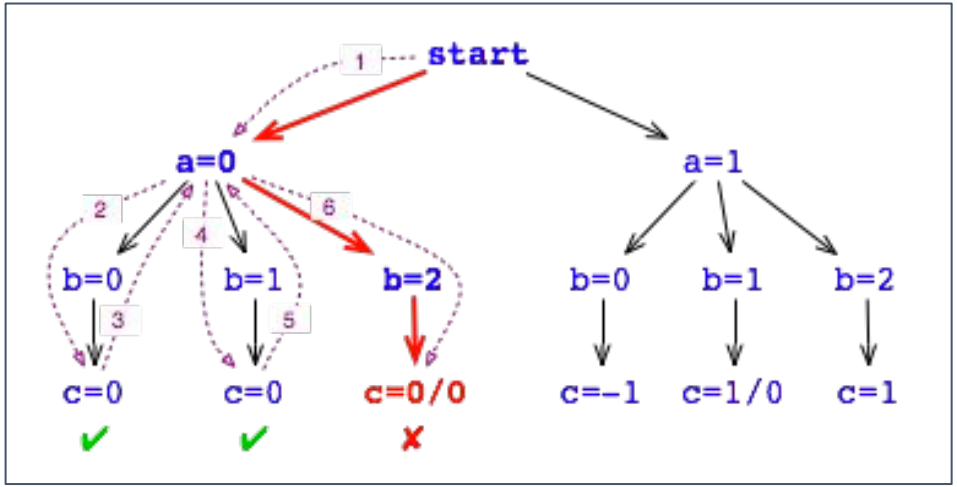
Model Checking: Example

Common tools:

- Isabelle & CSP (Haskell)
- PyModel
- Javapathfinder

```
[ASSERT]
All possible value combinations
result
in c = real #

b = [0,2]
a = [0,1]
c = a / ( a + b - 2 )
```



Single Static Assignment

- Structures the intermediate representation so that every variable is assigned exactly once
- Formally equivalent to continuation-passing style (CPS) IR

```
> map ($ 2) [(2*), (4*), (8*)]
[4,8,16]
```

- Developed at IBM

Single Static Assignment

- SSA form makes use-def chains explicit in the IR, which in turn helps to simplify some optimizations
- Def-use

```
Function *F = ...;

for (User *U : F->users()) {
  if (Instruction *Inst = dyn_cast<Instruction>(U)) {
    errs() << "F is used in instruction:\n";
    errs() << *Inst << "\n";
  }
```

- Use-def

```
Instruction *pi = ...;

for (Use &U : pi->operands()) {
  Value *v = U.get();
  // ...
}
```

Single Static Assignment

Motivation: Redundancy Elimination

Redundancy elimination optimizations attempt to remove redundant computations

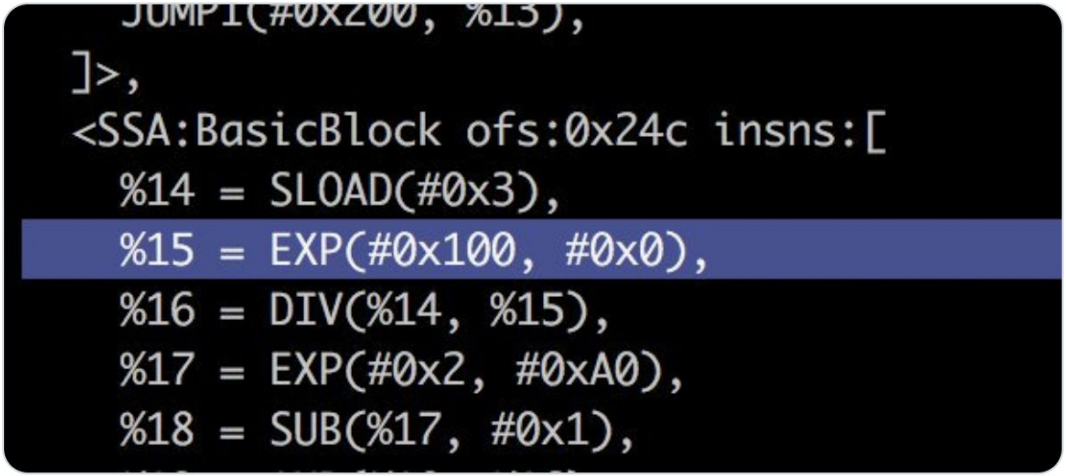
- value numbering
- conditional constant propagation
- common-subexpression elimination (CSE)
- partial-redundancy elimination

```
<SSABasicBlock offset:0x24c num_insns:30 in: [0x24b] insns:[
    <0x24c: %14 = SLOAD(#3)>
    <0x24d: %15 = EXP(#100, #0)>
    <0x24e: %16 = DIV(%14, %15)>
    <0x24f: %17 = EXP(#2, #a0)>
    <0x250: %18 = SUB(%17, #1)>
```

```
<SSABasicBlock offset:0x24c num_insns:30 in: [0x24b] insns:[
    <0x24c: %14 = SLOAD(#3)>
    <0x251: %19 = AND(#ffffffffffffffffffffffffffffffffffffffff, %14)>
```

Ryan Stortz
@withzombies

There are contracts on the blockchain that calculate 1 with exponentiation. This actually costs people money...



10:39 PM · Mar 6, 2018 · Twitter Web Client

Value Numbering

Associate a symbolic value to each computation, in a way that any two computations with the same symbolic value always compute the same value

SSA Form Congruency

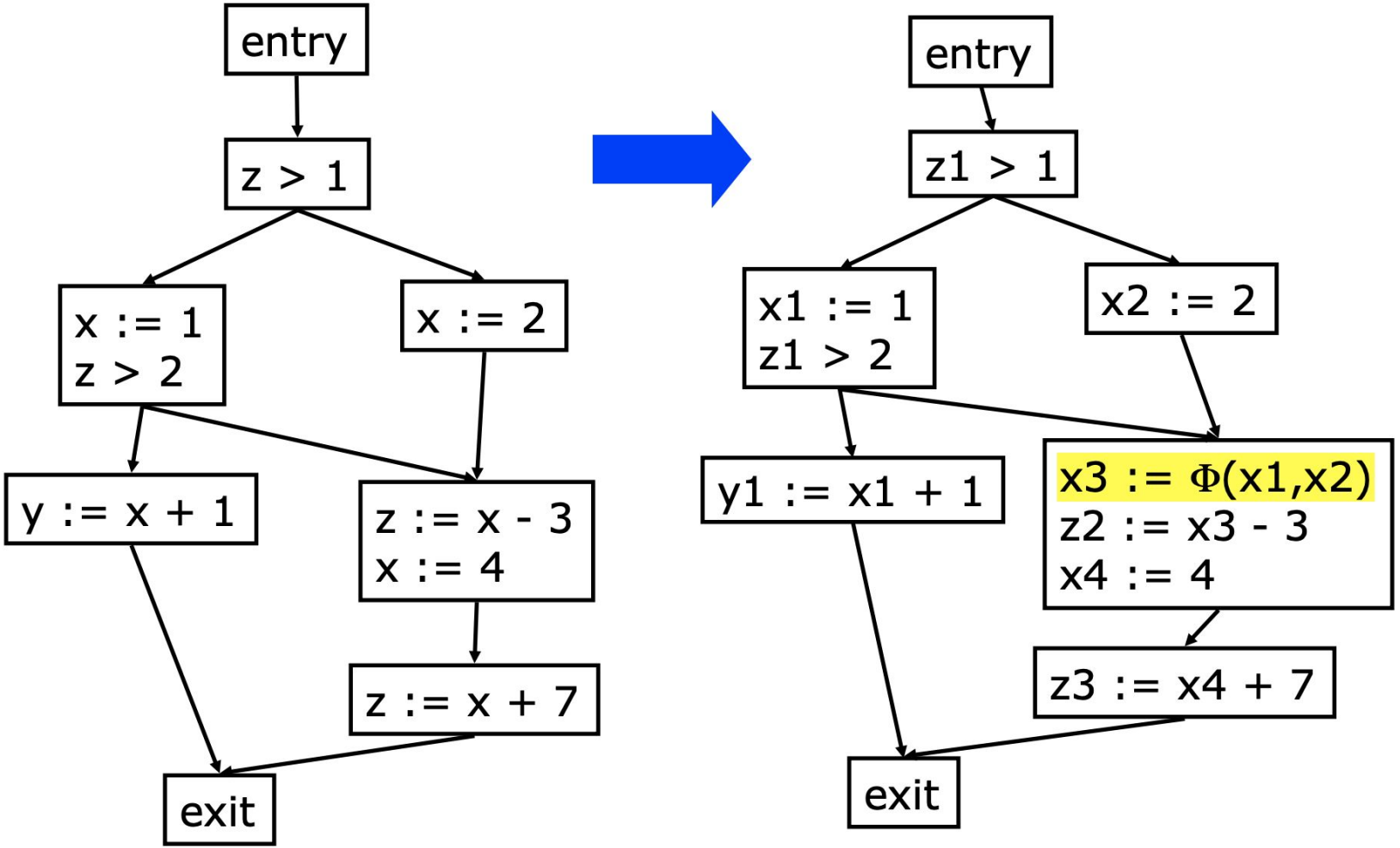
- In SSA form, if x and a are variables, they are congruent only if they are both alive and they are the same variable
- If they are provably the same value (by constant or copy propagation)

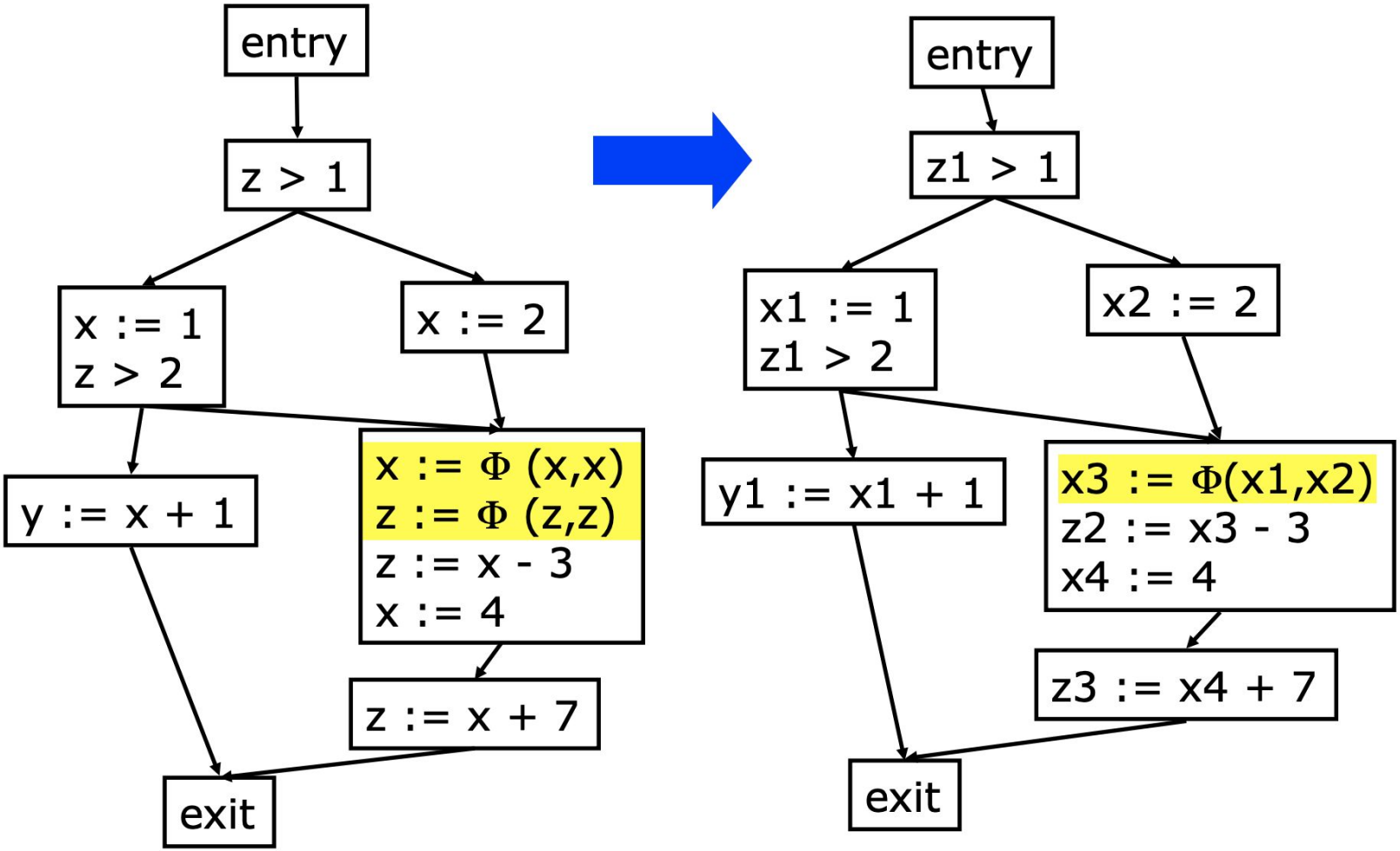
SSA Form

To translate into SSA form:

- Insert trivial Φ functions at join points for each live variable
- Φ(t,t,…,t), where the number of t's is the number of incoming flow edges
- Globally analyze and rename definitions and uses of variables to establish SSA property

After we are done with our optimizations, we can throw away all of the statements involving Φ functions (ie, "unSSA")

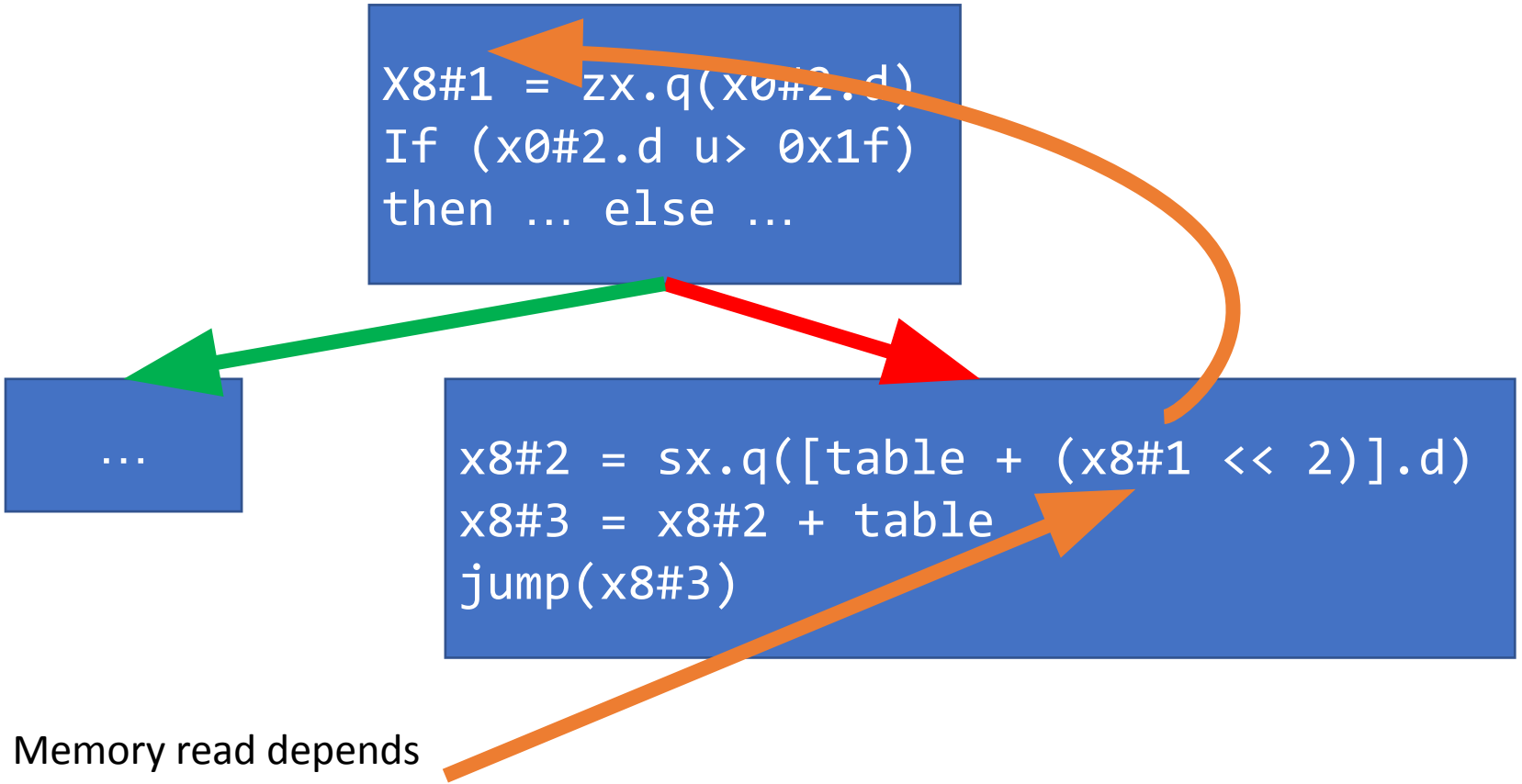

entry
z > 1
x := 1
z > 2
x := 2
y := x + 1
z := x - 3
x := 4
z := x + 7
exit
entry
z1 > 1
x1 := 1
z1 > 2
x2 := 2
y1 := x1 + 1
x3 := Φ(x1,x2)
z2 := x3 - 3
x4 := 4
z3 := x4 + 7
exit


entry
z > 1
x := 1
z > 2
x := 2
y := x + 1
x := Φ (x,x)
z := Φ (z,z)
z := x - 3
x := 4
z := x + 7
exit
entry
z1 > 1
x1 := 1
z1 > 2
x2 := 2
y1 := x1 + 1
x3 := Φ(x1,x2)
z2 := x3 - 3
x4 := 4
z3 := x4 + 7
exit

Exercise Three

Jump Tables

- Jump table resolution based on path-sensitive data flow
- SSA conversion process also tracks control flow dependence for every block
- Data flow computations allow disjoint sets of possible values
- Reads from memory are simulated
- At jump site, possible values are the possible jump targets



Medium Level IL SSA form

Exercise Two: find_bad_memcpy

- System call which performs and read and a write
- Three arguments: src, dst, size
- We can reason about all of them to some extent
  - Ideally:
    - Controlled, unbounded source (src)
    - Controlled, unbounded size (n)
    - Constant sized or controlled sized destination (dst)

```c
#include <string.h>

// clang -O0 -fno-stack-protector test.c -o test

int main(int argc, char** argv)
{
        char buf[64];
        char otherbuf[64];
        int8_t numBytes = argc;

        if (numBytes > (int8_t)100)
                return -1;

        memcpy(otherbuf, argv[0], numBytes % 64 );
        memcpy(buf, argv[0], numBytes);

        if (buf[63] == 0x42)
                return 0;
        else
                return -1;
}
```

Avoiding PHI

An SSA form with the minimum number of Φ functions can be created by using dominance frontiers

Definitions:

- In a flowgraph, node a dominates node b (“a dom b”) if every possible execution path from entry to b includes a
- If a and b are different nodes, we say that a strictly dominates b (“a sdom b”)
- If a sdom b, and there is no c such that a sdom c and c sdom b, we say that a is the immediate dominator of b (“a idom b”)

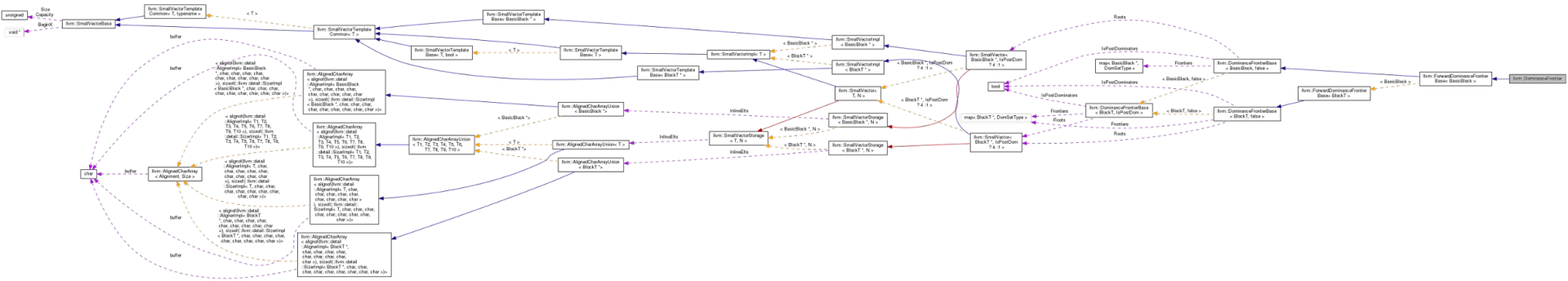
Dominance Frontier

For a node a, the dominance frontier of a, DF[a], is the set of all nodes b such that a strictly dominates an immediate predecessor of b but not b itself

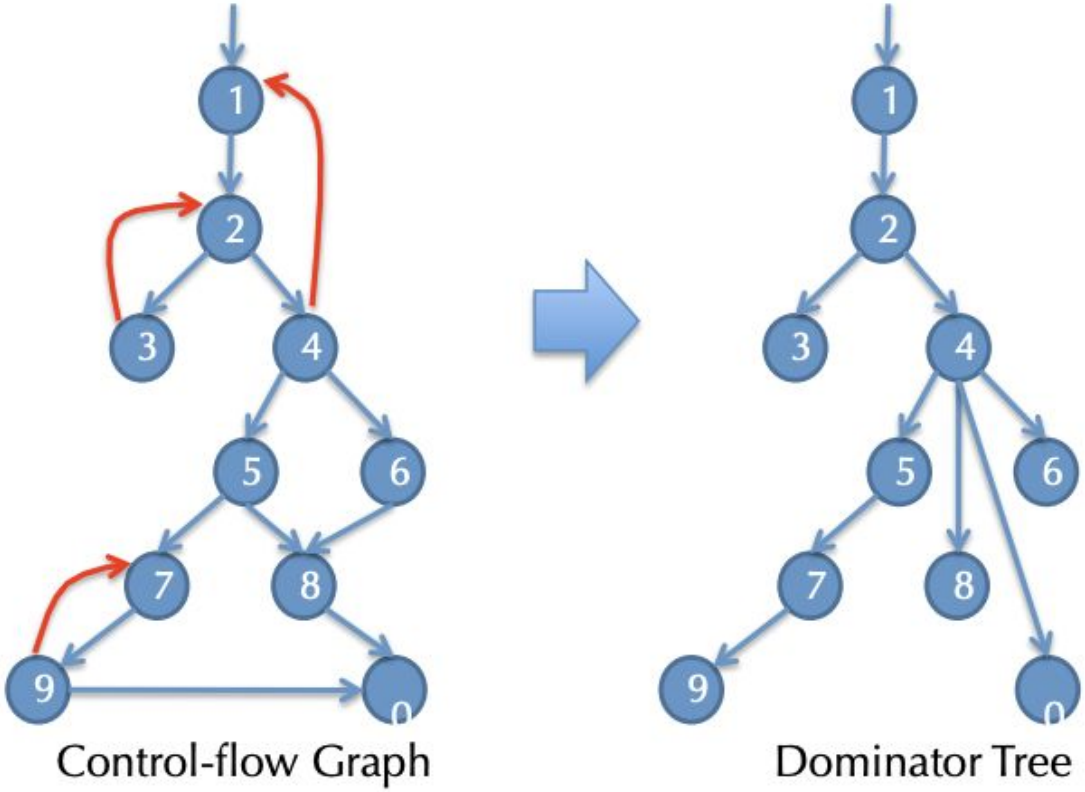
More formally:

DF[a] = {b | (∃c∈Pred(b) such that a dom c but not a sdom b}

Dominance Frontier



Dominance Frontier

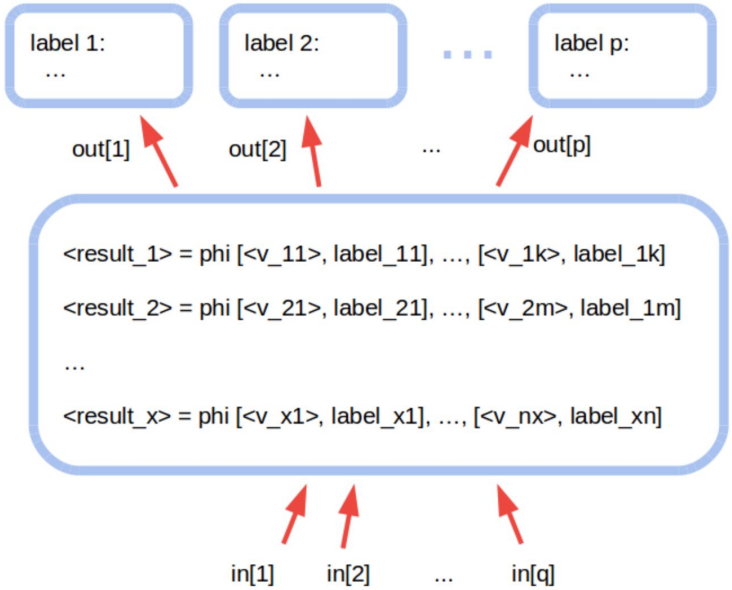


Control-flow Graph

Dominator Tree

PHI Nodes in Practice

- Path Explosion
- Incrementally harder to reason about



Uninitialized Variables

```
NTSTATUS TriggerUninitializedStackVariable(IN PVOID UserBuffer) {
    ULONG UserValue = 0;
    ULONG MagicValue = 0xBAD0B0B0;
    NTSTATUS Status = STATUS_SUCCESS;
    UNINITIALIZED_STACK_VARIABLE UninitializedStackVariable;

    PAGED_CODE();

    __try {
        // Verify if the buffer resides in user mode
        ProbeForRead(UserBuffer,
                     sizeof(UNINITIALIZED_STACK_VARIABLE),
                     (ULONG)__alignof(UNINITIALIZED_STACK_VARIABLE));

        // Get the value from user mode
        UserValue = *(PULONG)UserBuffer;

        // Validate the magic value
        if (UserValue == MagicValue) {
            UninitializedStackVariable.Value = UserValue;
            UninitializedStackVariable.Callback = &UninitializedStackVariableObjectCallback;
        }

        // Call the callback function
        if (UninitializedStackVariable.Callback) {
            UninitializedStackVariable.Callback();
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        Status = GetExceptionCode();
    }
    return Status;
}
```

Type Analysis

- Various operations are intended to be applied only to certain arguments
  - arithmetic operations and comparisons apply only to integers
  - conditions in control structures must be integers
  - only integers can be input and output of the main function
  - only functions can be called, and with correct number of arguments
  - the unary * operator only applies to heap pointers (or null)
  - field lookups are only performed on records, not on other types of values

Type Analysis

Assumption: that their violation results in runtime errors

Analysis: checks that these requirements hold during execution

This is an nontrivial question, it is undecidable.

Resort to a conservative approximation: ***typability***

$$\begin{aligned} \tau \rightarrow\ & \mathsf{int} \\ \mid\ & \&\tau \\ \mid\ & (\tau,\ldots,\tau) \rightarrow \tau \end{aligned}$$

Type Constraints

```
short() {
    var x, y, z;  // [[short]] = ()→[[z]]
    x = input;    // [[input]] = int
    y = alloc x;  // [[alloc x]] = &[[x]] , [[y]] = [[alloc x]]
    *y = x;       // [[y]] = &[[x]]
    z = *y;       // [[z]] = [[*y]]
    return z;     // [[z]] = [[*y]]
}
```

Type Constraints

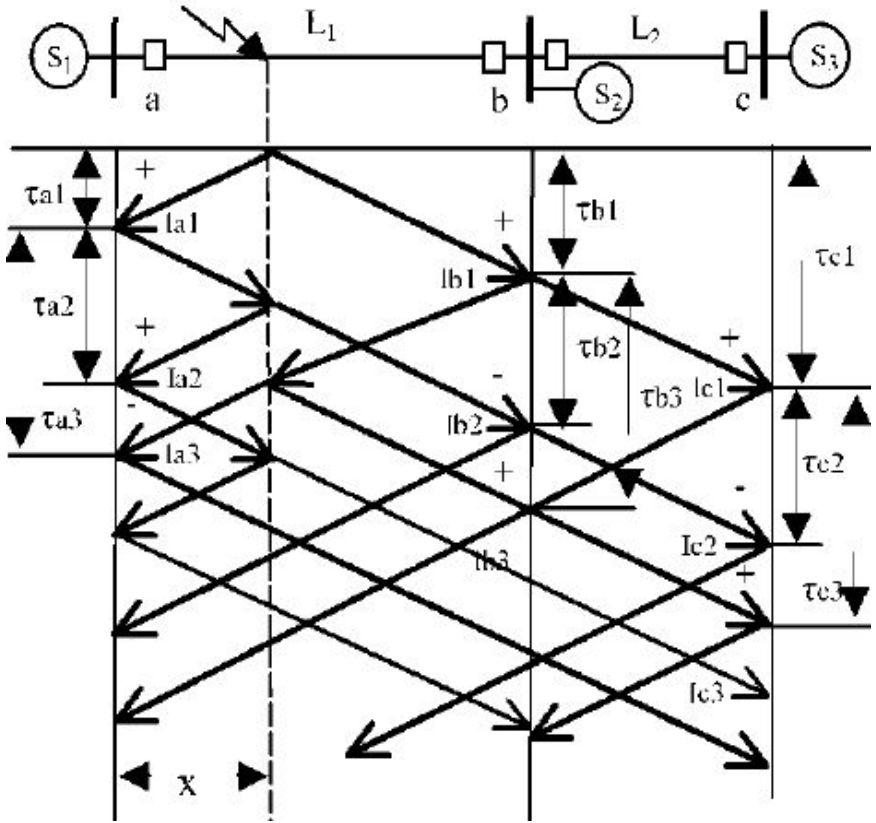
- A given program (or fragment of a program) gives rise to a collection of equality constraints on type terms with variables
- Collection of constraints can be built by traversing the CFG
- The order by which the CFG is traversed is irrelevant!

Type Constraints

- A “solution“ assigns a type to each type variable, such that all equality constraints are satisfied
- Correctness
  - Specific runtime errors cannot occur
- *Identifiers solution*

```
[[short]] = ()→int
[[x]] = int
[[y]] = &int
[[z]] = int
```

Lattice Theory




The spirit of the 90s is alive in Brooklyn
Dumbo
Brooklyn, NY

Lattice Overview

- A lattice is a partial order where every subset has a least upper bound and a greatest lower bound
- “Least upper bound” = Join
- “Greatest lower bound” = Meet
- Ordered from least to most precise information

S_{12} = {1, 2, 3, 4, 6, 12}

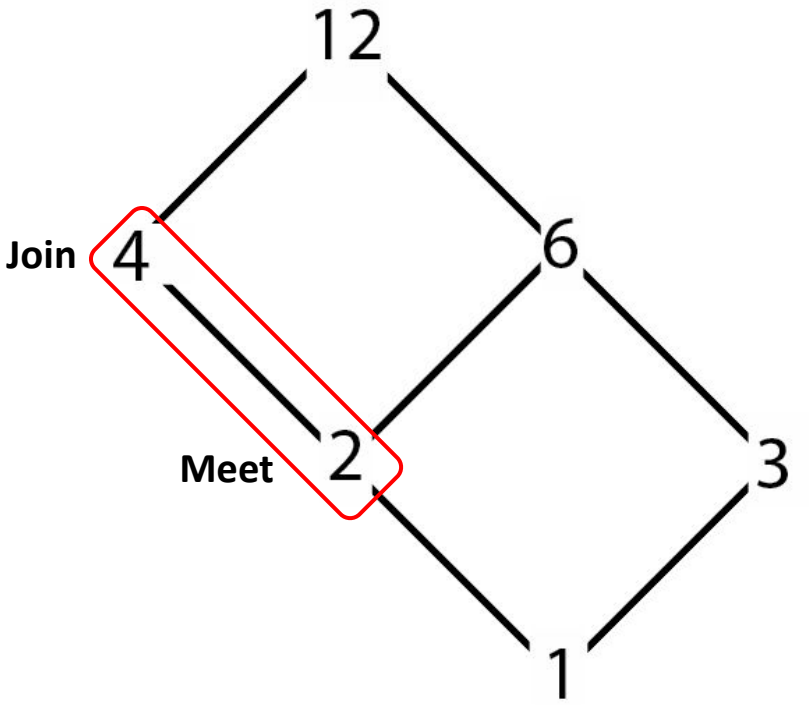
Is this a lattice?

Lattice Overview

S_{12} = {1, 2, 3, 4, 6, 12}

Is this a lattice?

(2, 4)

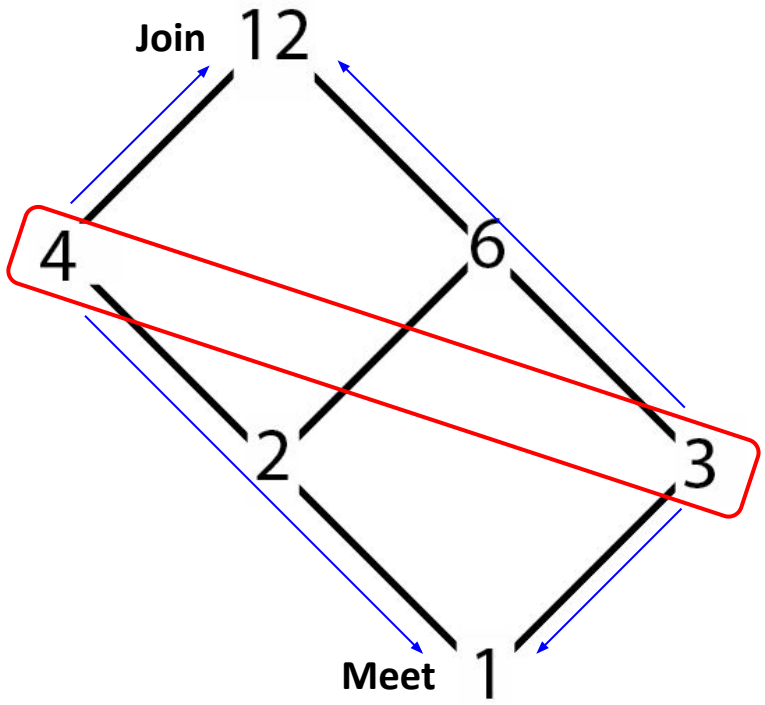


Lattice Overview

S_{12} = {1, 2, 3, 4, 6, 12}

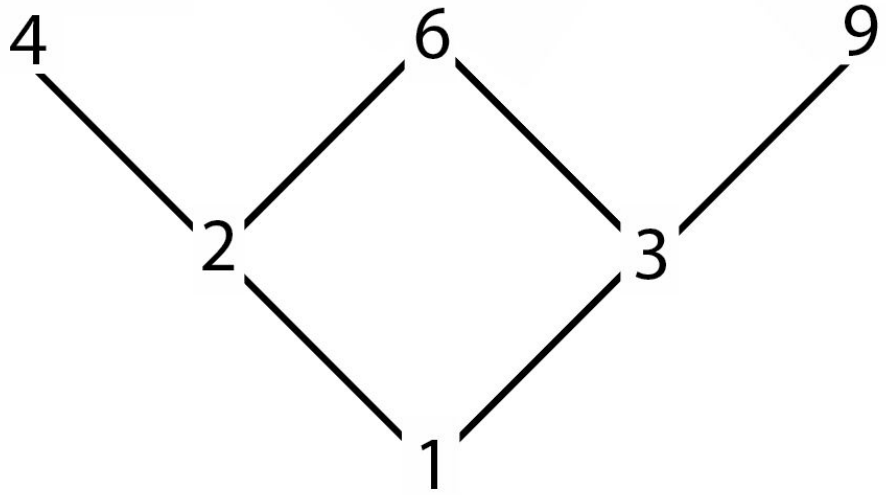
Is this a lattice?

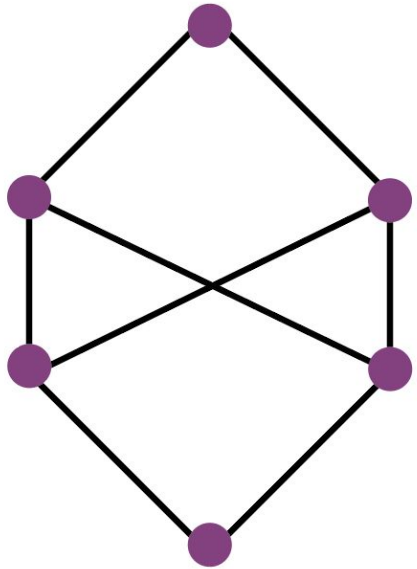
(3, 4)



Lattice Overview

(Things that are not lattices)





Sign Analysis

- Analysis returns possible signs of the integer values of variables and expressions in a given program
  - One degree of detail past type analysis discussed previously
- Concrete executions: values can be arbitrary integers
- Circumvent undecidability by introducing approximation

Sign Analysis

- An abstraction of integer values
  - Three abstract values to describe a known integer
  - Positive, Negative, Zero
  - Other values include pointers and integers of unknown sign
- Not analysis for more specific integer types!
  - UINT64, INT16, etc…

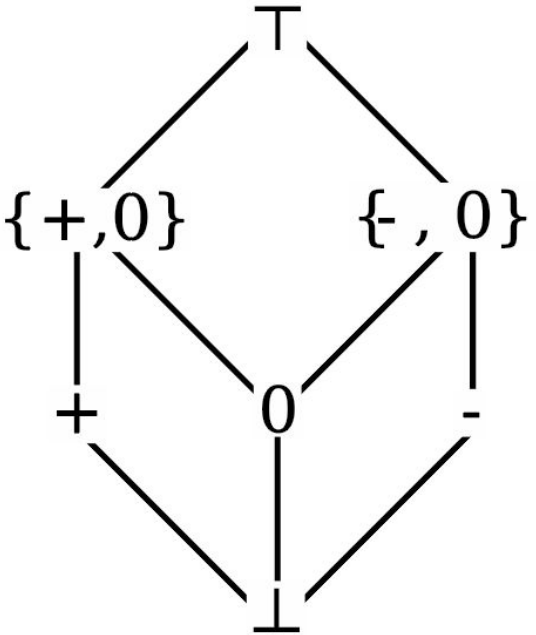
Sign Analysis

- Assume interest in definite information, statement true for every execution path
  - Only report + if expression will certainly evaluate to a positive number in every execution of that expression
- '$\perp$' for values that are not operated on as numbers (i.e., pointers), have no value in any execution, or unreachable
- '$\top$' for values that have an unknown sign. Could be any number of sign {0, +, -}

Signedness Example

```
int main(int argc,  char **argv){
     auto a = 0, b = 0, c = 0;
     a = argc + 42;
     b = argc + 87;

     if(argc < 1){
          c = a - b;
     } else {
          c = a + b;
     }
     return c;
}
```



Signedness Example (Abstract Operators)

+	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	-	+	⊤
-	⊥	-	-	⊤	⊤
+	⊥	+	⊤	+	⊤
⊤	⊥	⊤	⊤	⊤	⊤

-	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	-	⊤
-	⊥	-	⊤	-	⊤
+	⊥	+	+	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊤

Signedness Example

```
x0 int main(int argc,  char **argv) {
x1    auto a = 0, b = 0, c = 0;
x2    a = argc + 42;
x3    b = argc + 87;

      if(argc < 1) { // Test if argc is +
      [c STATE ONE]
x4         c = a - b;
      } else {
      [c STATE TWO]
x5         c = a + b;
      }
x6    return c;  // JOIN
   }
```

$x_0 = [argc \rightarrow \{0,+\},\ argv \rightarrow \bot]$

$x_1 = [a \rightarrow 0,\ b \rightarrow 0,\ c \rightarrow 0]$

$x_2 = [a \rightarrow x_0(argc)+(+) \rightarrow +]$

$x_3 = [b \rightarrow x_0(argc)+(+) \rightarrow +]$

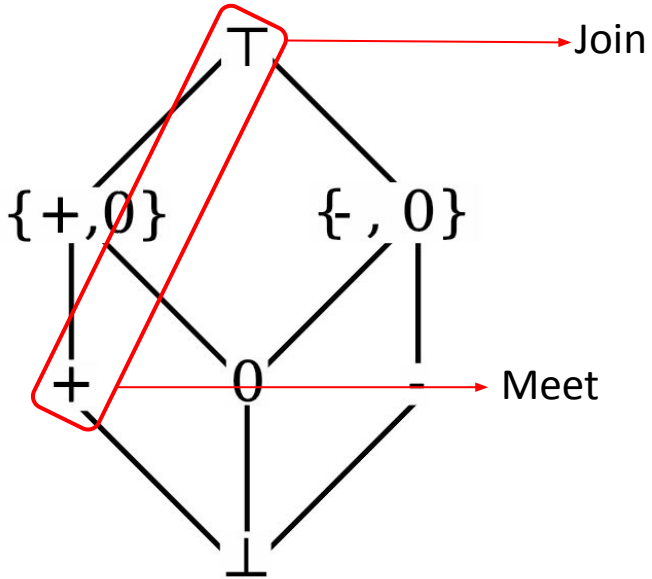
$x_4 = [c \rightarrow x_2(a)-x_3(b) \rightarrow (+)-(+) \rightarrow \top]$

$x_5 = [c \rightarrow x_2(a)+x_3(b) \rightarrow (+)+(+) \rightarrow +]$

$x_6 = [c \rightarrow \{x_4(c), x_5(c)\} \rightarrow \{\top,+\} \rightarrow \top]$

Signedness Example

x_6 `return c;` `// JOIN`



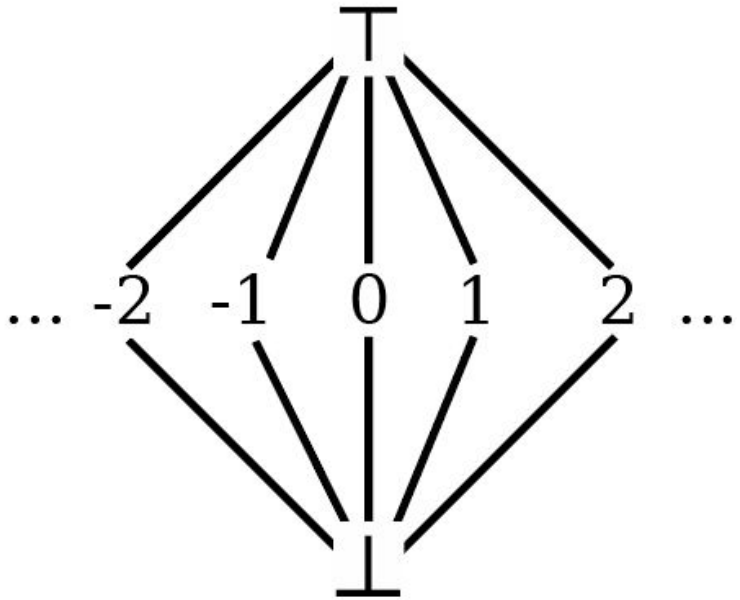
$x_4(c) \rightarrow \top$

$x_5(c) \rightarrow +$

$x_6 = [c \rightarrow \{x_4(c), x_5(c)\} \rightarrow \{\top, +\} \rightarrow \top]$

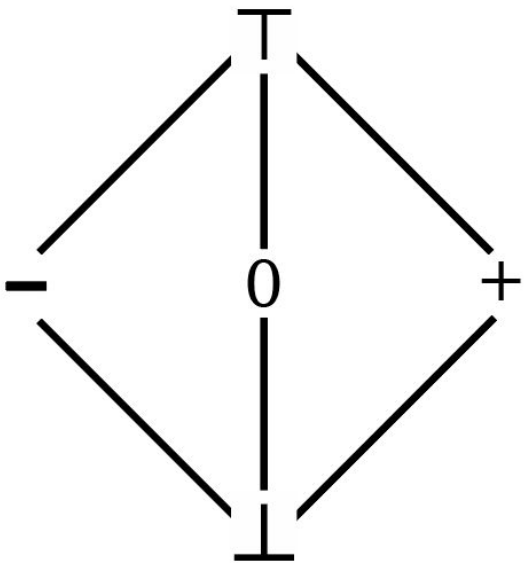
Constant Propagation Example

```
x0  var a, b, c
x1  a = 0
x2  b = a + 12
x3  c = a + b
x4  a = foo(a)
x5  c = foo(c)
x6  return a+c
```



Abstract Interpretation

- An abstraction is a property
- Abstract Domain
  - Less complex domain of properties
- Abstract Interpretation
  - Mapping from a complex domain to less complex domain



ABSTRACT INTERPRETATION : A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

This implies that A-Cont is in fact a complete lattice, but we need only one of the two join and meet operations. The set of context vectors is defined by $\widetilde{\text{A-Cont}} = \text{Arcs}^0 \rightarrow \text{A-Cont}$.
Whatever $(Cv', Cv'') \in \widetilde{\text{A-Cont}}^2$ may be, we define :

$$Cv' \mathbin{\tilde{\circ}} Cv'' = \lambda r \,.\, Cv'(r) \circ Cv''(r)$$

$$Cv' \mathbin{\tilde{\leq}} Cv'' = \{\forall r \in \text{Arcs}^0,\ Cv'(r) \leq Cv''(r)\}$$

$$\tilde{\top} = \lambda r \,.\, \top \quad \text{and} \quad \tilde{\bot} = \lambda r \,.\, \bot$$

$\langle \widetilde{\text{A-Cont}}, \tilde{\circ}, \tilde{\leq}, \tilde{\top}, \tilde{\bot} \rangle$ can be shown to be a complete lattice. The function :

$$\underline{\text{Int}} : \text{Arcs}^0 \times \widetilde{\text{A-Cont}} \rightarrow \text{A-Cont}$$

defines the interpretation of basic instructions. If $\{C(q) \mid q \in \underline{\text{a-pred}}(n)\}$ is the set of input contexts of node n, then the output context on exit arc r of n ($r \in \underline{\text{a-succ}}(n)$) is equal to $\underline{\text{Int}}(r, C)$. $\underline{\text{Int}}$ is supposed to be order-preserving :

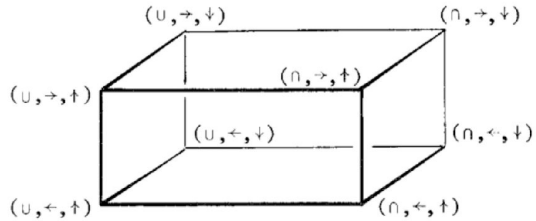
$\forall a \in \text{Arcs}, \forall (\underline{Cv'}, \underline{Cv''}) \in \widetilde{\text{A-Cont}}^2,$

$$\{\underline{Cv'} \mathbin{\tilde{\leq}} \underline{Cv''}\} \Longrightarrow \{\underline{\text{Int}}(a, \underline{Cv'}) \leq \underline{\text{Int}}(a, \underline{Cv''})\}$$

The local interpretation of elementary program constructs which is defined by $\underline{\text{Int}}$ is used to associate a system of equations with the program. We define

$$\widetilde{\underline{\text{Int}}} : \widetilde{\text{A-Cont}} \rightarrow \widetilde{\text{A-Cont}} \mid \widetilde{\underline{\text{Int}}}(\underline{Cv}) = \lambda r \,.\, \underline{\text{Int}}(r, \underline{Cv})$$

It is easy to show that $\widetilde{\underline{\text{Int}}}$ is order-preserving. Hence it has fixpoints, Tarski[55]. Therefore the context vector resulting from the abstract interpretation I of program P, which defines the global properties of P, may be chosen to be one of the extreme solutions to the system of equations $\underline{Cv} = \widetilde{\underline{\text{Int}}}(\underline{Cv})$.



Examples :

Kildall[73] uses $(\cap, \rightarrow, \uparrow)$, Wegbreit[75] uses $(\cup, \rightarrow, \downarrow)$. Tenenbaum[74] uses both $(\cup, \rightarrow, \downarrow)$ and $(\cap, \leftarrow, \uparrow)$.

5.3 Examples

5.3.1 Static Semantics of Programs

The static semanticsof programs we defined in section 4 is an abstract interpretation :

$$I_{SS} = \langle \text{Contexts}, \cup, \subseteq, \text{Env}, \emptyset, \underline{\text{n-context}} \rangle$$

where Contexts, $\cup$, $\subseteq$, Env, $\emptyset$, $\underline{\text{n-context}}$, Context-Vectors, $\tilde{\cup}$, $\tilde{\subseteq}$, $\widetilde{\text{F-Cont}}$ respectively correspond to A-Cont, $\circ$, $\leq$, $\top$, $\bot$, $\underline{\text{Int}}$, $\widetilde{\text{A-Cont}}$, $\tilde{\circ}$, $\tilde{\leq}$, $\widetilde{\underline{\text{Int}}}$.

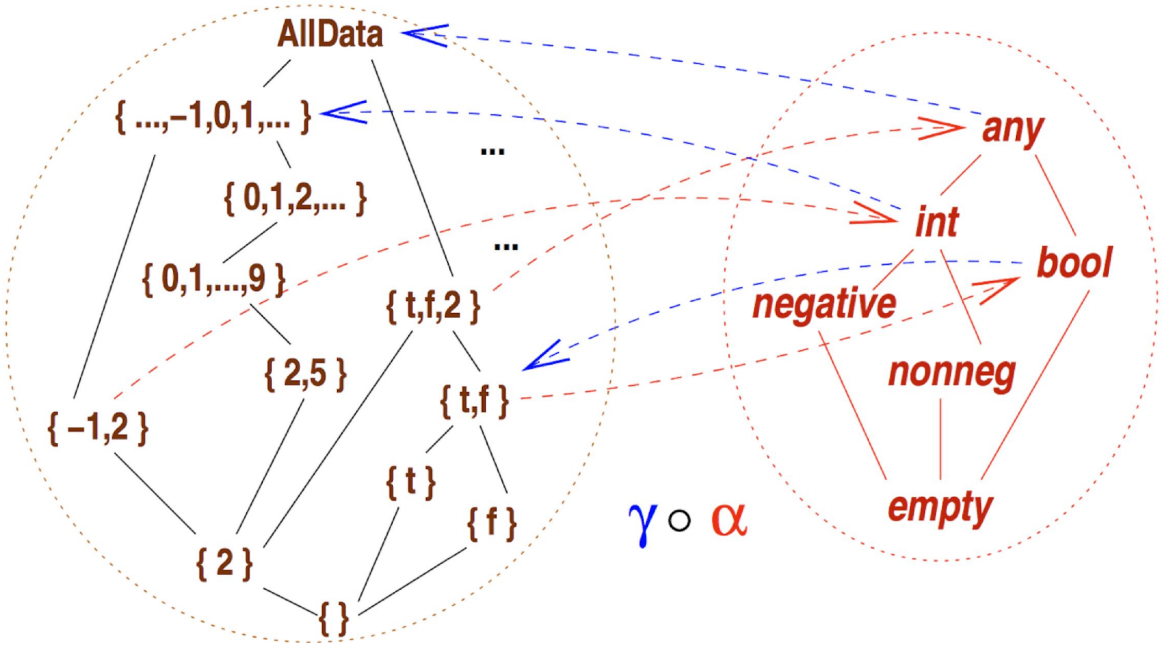
5.3.2 Data Flow Analysis

Data flow analysis problems (see references in Ullman[75]) may be formalized as abstract interpretations of programs.

brown (color)

heavy (weight)

Abstract Interpretation



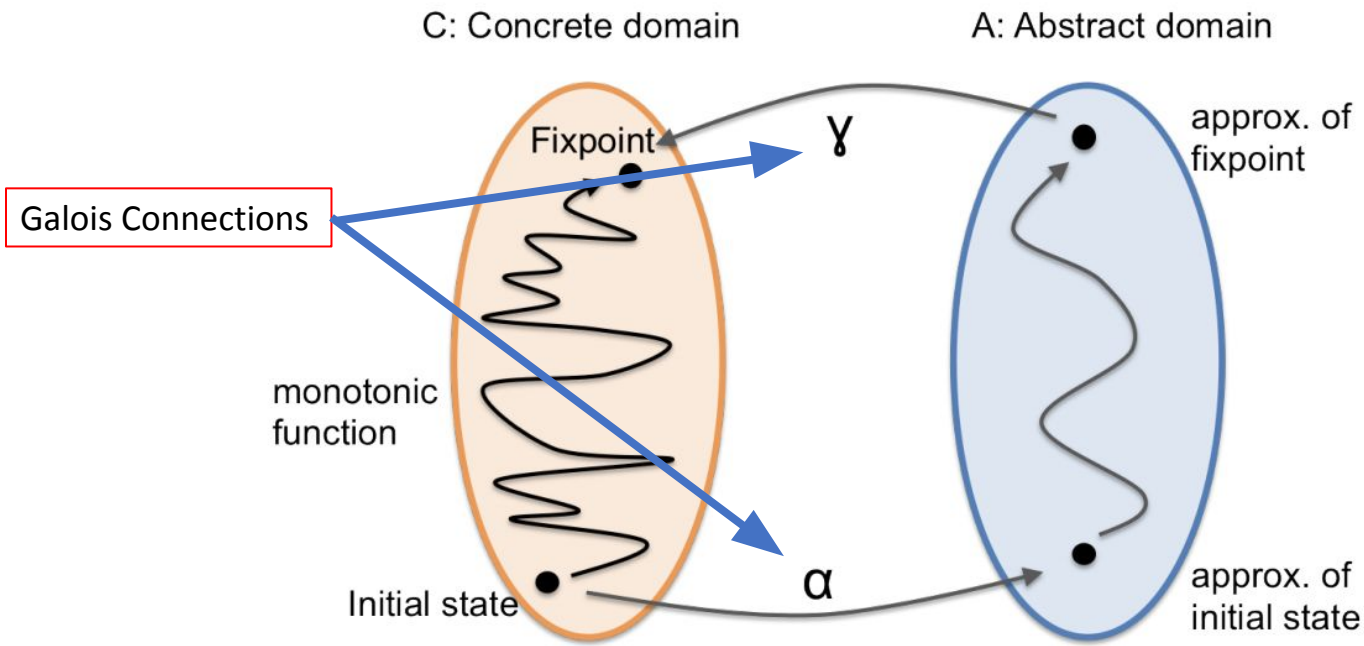
Abstract Semantics

- Problem: Compute a sound approximation $S^{\#}$ of S

$S \subseteq S^{\#}$

This is a Galois Connection (order theory)

Abstract Semantics



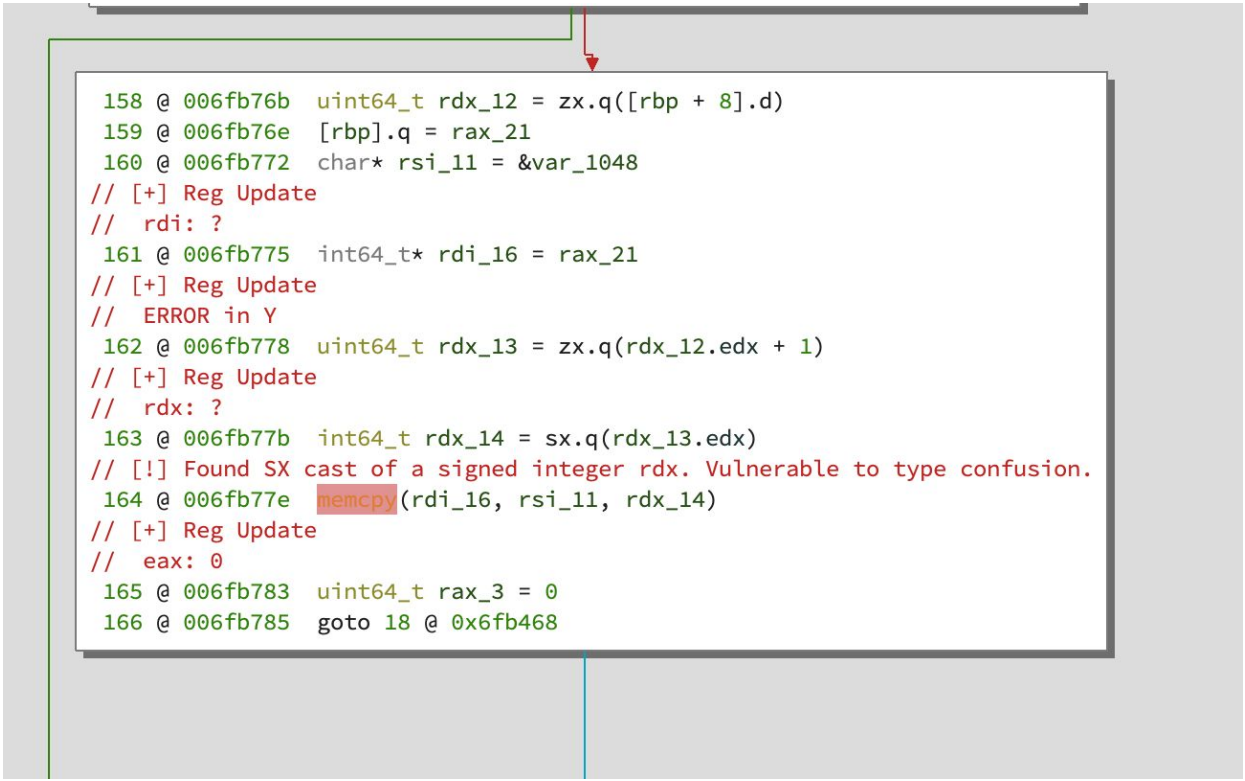
Exercise: PHP

- PHP CVE-2016-6297
  - “zip://” stream handler
  - **php_stream_zip_opener** fails to check the **path_len**
  - Integer overflow
  - This value will be directly used to calculate the length passed to the memcpy function, which then leads to buffer overflow and memory corruption
- Even if safe calculations are used for the memcpy length, the value is already ‘corrupted’
- Analysis must decompose all components of the expression for n
- Similar to *Time of use, time of check*

Exercise: PHP

```
<snippet ext/zip/zip_stream.c:289>
    fragment_len = strlen(fragment);
    if (fragment_len < 1) {
        return NULL;
    }
    path_len = strlen(path); //path_len can be negative
    if (path_len >= MAXPATHLEN || mode[0] != ‘r’){
        return NULL;
    }
     //(path_len - fragment_len) can be controlled
    memcpy(file_dirname, path, path_len - fragment_len);
</snippet>
```

Exercise: PHP



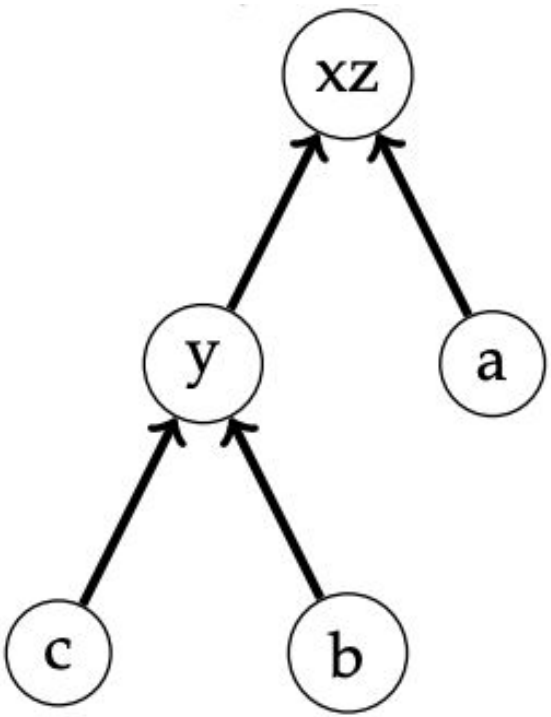
Heap Analysis

Steensgaard's Points-To Analysis

```
1 : p : &x
2 : r : &p
3 : q : &y
4 : s : &q
5 : r : s
```

Steensgaard's Points-To Analysis

```
1 : a = &x
2 : b = &y
3 : if p then
4 :     y = &z
5 : else
6 :     y = &x
7 : c = &y
```



Points-To Exercise

```
switch(num)
{
    case 1:
        name = malloc(255*sizeof(char));
        printf("Insert Username: ");
        scanf("%254s", name);
        if(strcmp(name,"root") == 0)
        {
            printf("root not allowed.\n");
            strcpy(name,"");
        }
        break;
    case 2:
        pass = malloc(255*sizeof(char));
        printf("Insert Password: ");
        scanf("%254s", pass);
        break;
    case 3:
        free(name);
        free(pass);
        break;
    case 4:
        if(strcmp(name,"root") == 0)
        {
            printf("You just used after free!\n");
            exit(0);
        }
        break;
    case 5:
        exit(0);
}
```

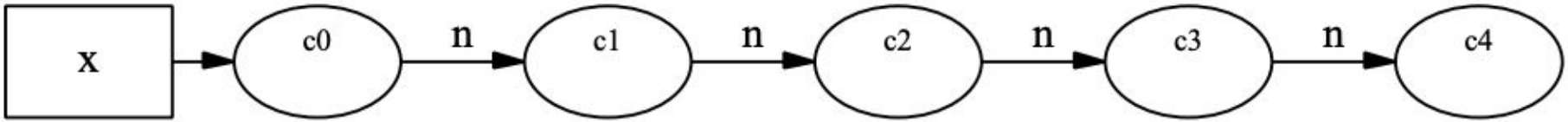
Shape Analysis

Goal: Determine, at each state, shape graphs which represent heap structures

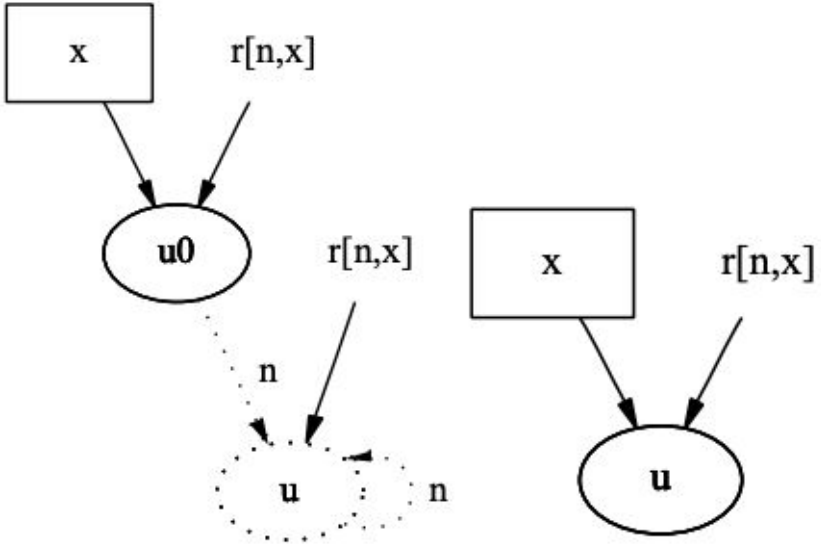
- Forward analysis
- External value: shape graphs for possible initial values of Var

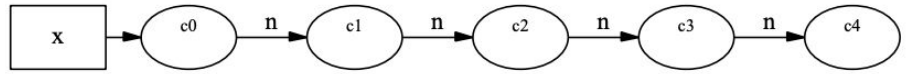
⇒ Structure recovery!

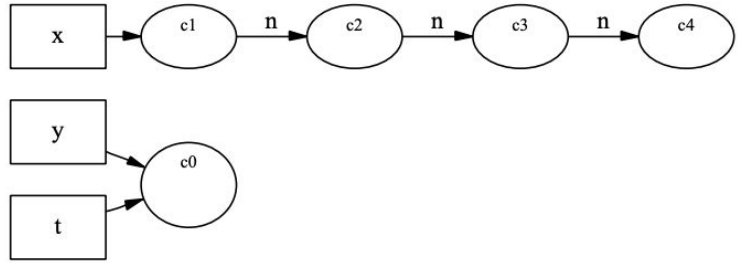
Example: Shape Analysis

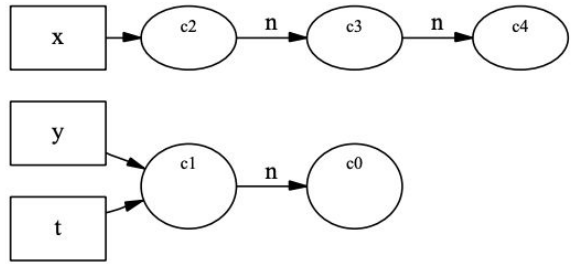


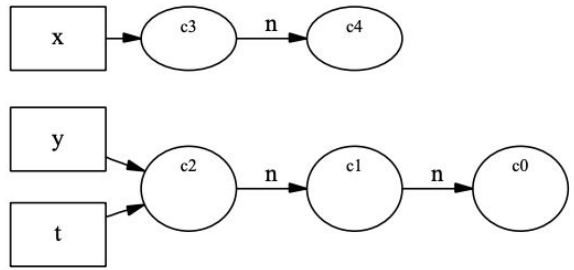
Shape Analysis Answers



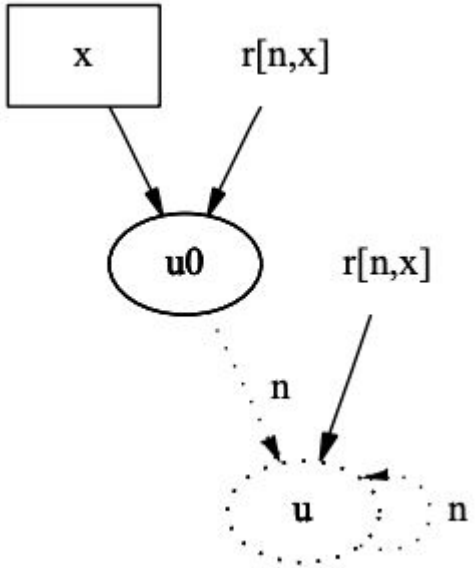

x
c0
n
c1
n
c2
n
c3
n
c4

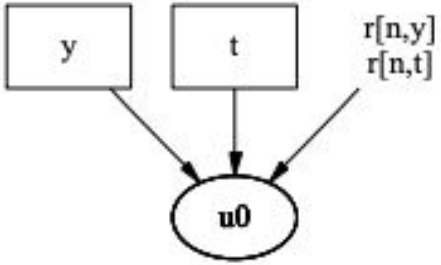

x
c1
n
c2
n
c3
n
c4
y
t
c0

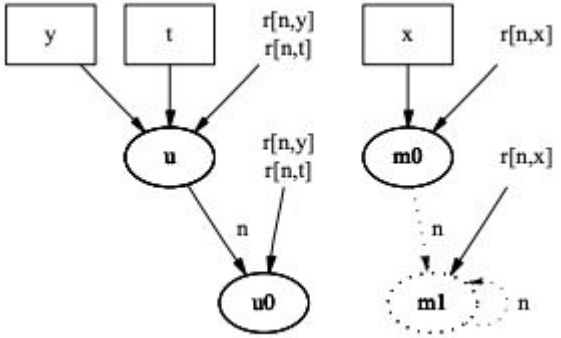

x
c2
n
c3
n
c4
y
t
c1
n
c0

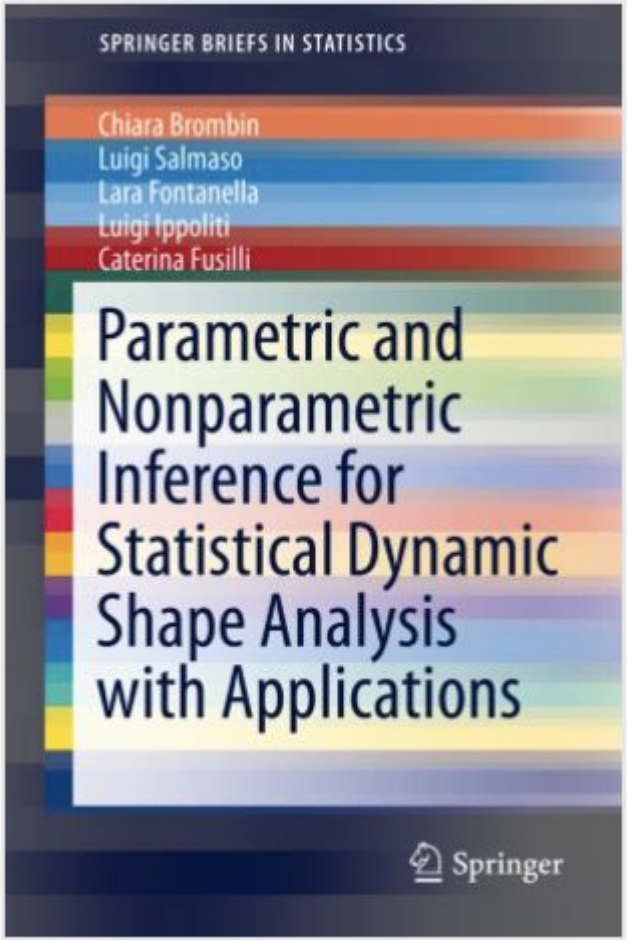

x
c3
n
c4
y
t
c2
n
c1
n
c0

Shape Analysis: All execution states









- Core properties
  - POINTS-TO!
  - pointed-to-by-x
- Instrumentation properties
  - Shared properties between shapes
  - is-heap-cell-shared
  - reachable-from-x-through-n

THOR: A Tool for Reasoning About Shape and Arithmetic*

(Tool Paper)

Stephen Magill[1], Ming-Hsien Tsai[2], Peter Lee[1], and Yih-Kuen Tsay[2]

[1] Carnegie Mellon University
[2] National Taiwan University

Separation Logic

- Shape analysis maps shapes to heap structures
- Issue: Global nature leads to updates altering entire heap state
- Solution: Separation Logic!
- Result: Updates to heap state don’t need to ‘check’ all cells

THOR: Example

example.c

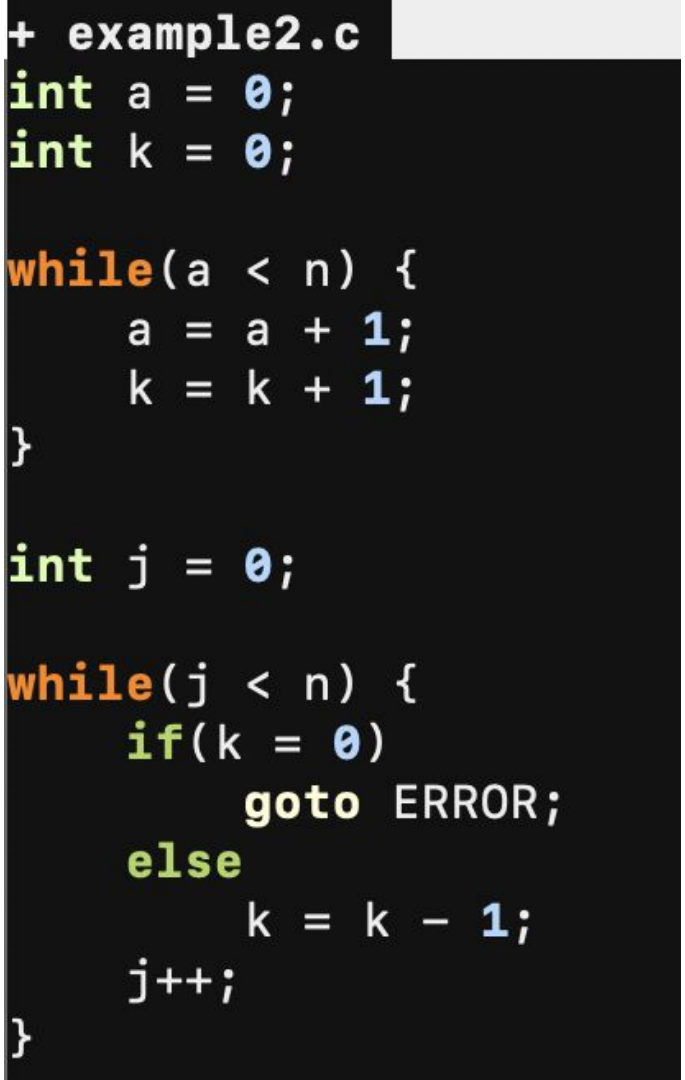
```
int i = malloc(sizeof(int));
List *curr = NULL; *i = 0;

while(*i < n) {
    t = (List*) malloc(sizeof(List));
    t->next = curr;
    curr = t;
    *i = *i + 1;
}

free(i); int j = 0;

while(j < n) {
    t = curr->next;
    free(curr);
    curr = t;
    j++;
}
```

THOR Example



```
+ example2.c
int a = 0;
int k = 0;

while(a < n) {
    a = a + 1;
    k = k + 1;
}

int j = 0;

while(j < n) {
    if(k = 0)
        goto ERROR;
    else
        k = k - 1;
    j++;
}
```

More Material

https://cs.au.dk/~amoeller/spa/spa.pdf
https://mitpress.mit.edu/books/introduction-static-analysis
https://link.springer.com/book/10.1007/978-3-662-03811-6
https://www.amazon.com/Introduction-Theory-Optimizing-Compilers-measurements/dp/1537091123

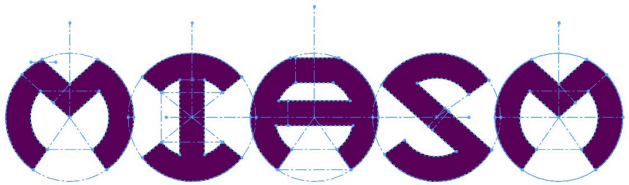
MIASM (2006)

AIRBUS

BinCAT: purrfecting binary static analysis (IDA)

Performs value and taint analysis, type reconstruction, use-after-free and double-free detection

Emulating using JIT, expression simplification, IL's (side effects), variable backtracking

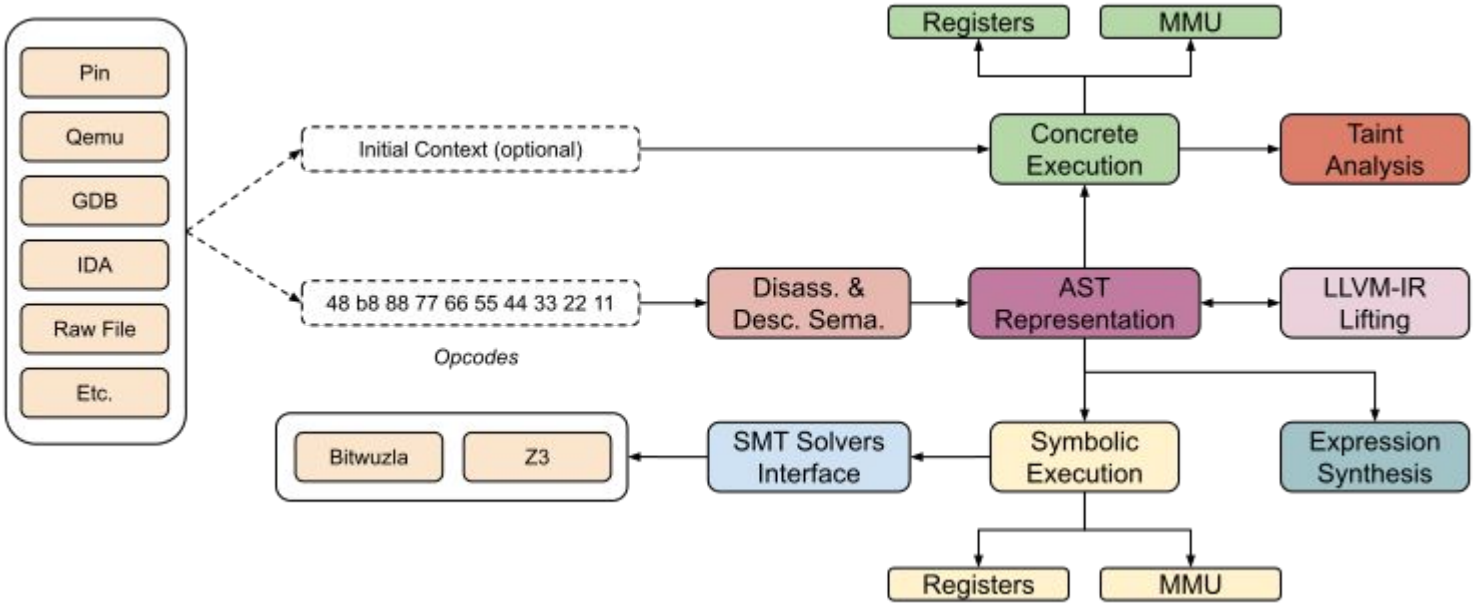


Triton (2015)

Dynamic symbolic execution, taint analysis

SMT simplification passes (de-obfuscation), solver to Z3 and Bitwuzla

Lifts to LLVM



REVEN (2014)

Reversing Platform: timeless debugging, emulation, the whole 9 yards

Determine exploitability and reachability

REVEN allows us to record every single instruction executed in a virtual machine (VM) during a certain period, simulate it and analyze everything statically at any point of execution with a complete view of the machine’s state.

Symbolic CPU

I WANT YOU

FOR MARGIN RESEARCH

Summer Internship 2022

INTERNSHIP@MARGIN.RE