Interprocedural Binary Analysis

For Automation and Computer-Human Collaboration

Dr. Matt Revelle
HACR Lab
Montana State University
Bozeman, MT USA

2024 April 01

Outline

- Vulnerability Research Example
- Programs as Graphs
- Simplifying Graphs with Constraints
- Type Inference for Binary Analysis

Interprocedural Binary Analysis

Example


```
cgiFormString(0x1aaa4, &var_910, 0x200);  // {"user"}
cgiFormString(0x1aaac, &var_710, 0x200);  // {"passwd"}
cgiFormString(0x1accc, &var_510, 0x200);  // {"start"}
cgiFormString(0x1acd4, &var_310, 0x200);  // {"count"}
if (data_2c2e4 != 0)
{
    sub_1194c("name --> [%s][%s]\n", &var_910);
    sub_1194c("count -->[%s][%s]\n", &var_510);
}
int32_t r0_9;
if (sub_160a8(&var_910, &var_710) == 0)
{
    r0_9 = sub_119a4();
}
else
{
    sprintf(&var_110, "sqlsearch -t video -o %s -r %s,%…"  , "/tmp/video_list.xml", &var_510, &var_310);
    if (data_2c2e4 != 0)
    {
        sub_1194c("cmd[%s]\n", &var_110);
    }
    system(&var_110);
```

Authentication check

Command injection

https://www.exploit-db.com/exploits/43435

Interprocedural Binary Analysis

Example

```
if (((uint32_t)*(int8_t*)arg2) != 0)
{
    __b64_pton(arg2, &var_1090, strlen(arg2));
    if (data_2c2e4 != 0)
    {
        sub_1194c("pwd [%s]\n", arg2);
        sub_1194c("pwd decode[%s]\n", &var_1090);
    }
}
int32_t r0_4 = strcmp(arg1, "mydlinkBRionyg");
int32_t r0_6;
void* r0_7;
if (r0_4 == 0)
{
    r0_6 = strcmp(&var_1090, "abc12345cba");
    if (r0_6 == 0)
    {
        r0_7 = 1;
    }
}
```

Decode Base64 encoded password

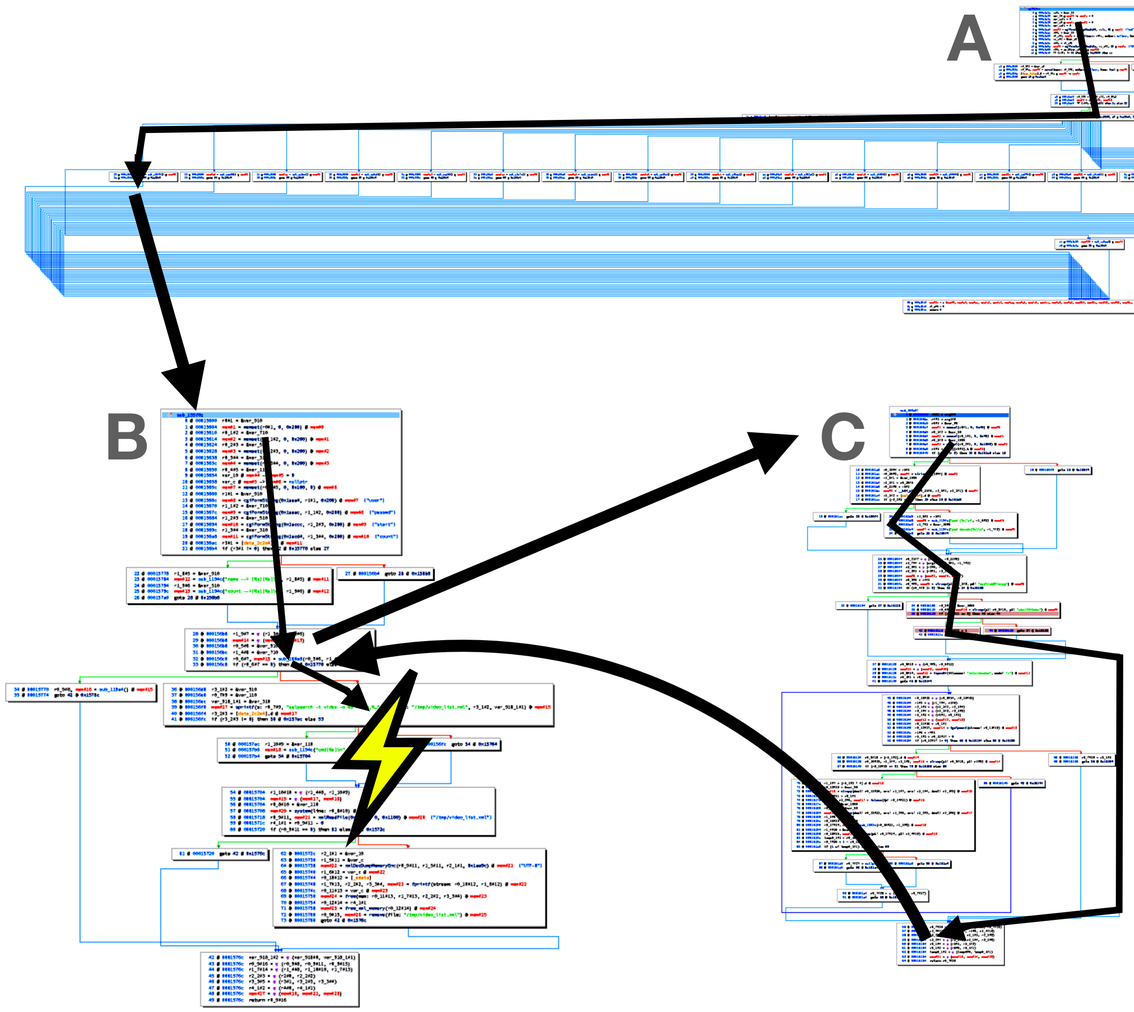
Hard-coded login credentials

`is_authenticated` variable

https://www.exploit-db.com/exploits/43435

Interprocedural Binary Analysis

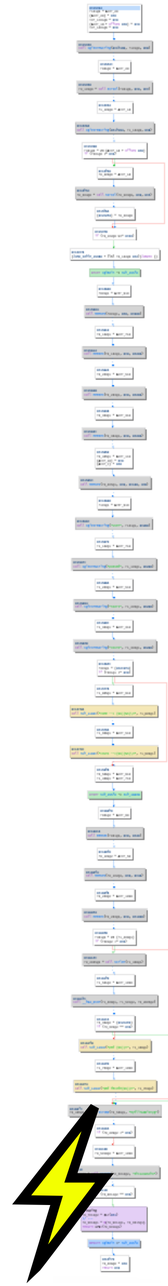
Example



- Task: Find a feasible path that uses the hard-coded credentials and reaches the vulnerability
- Manual tracking of feasible paths and constraints over multiple function control-flow graph (CFGs)

Interprocedural Binary Analysis
Example

- Number of paths in a function can be very large, but often many are infeasible
- Automated removal of these paths can have a big impact
- Can use automated analyses to automatically simplify an interprocedural CFG as it is constructed



Interprocedural Binary Analysis

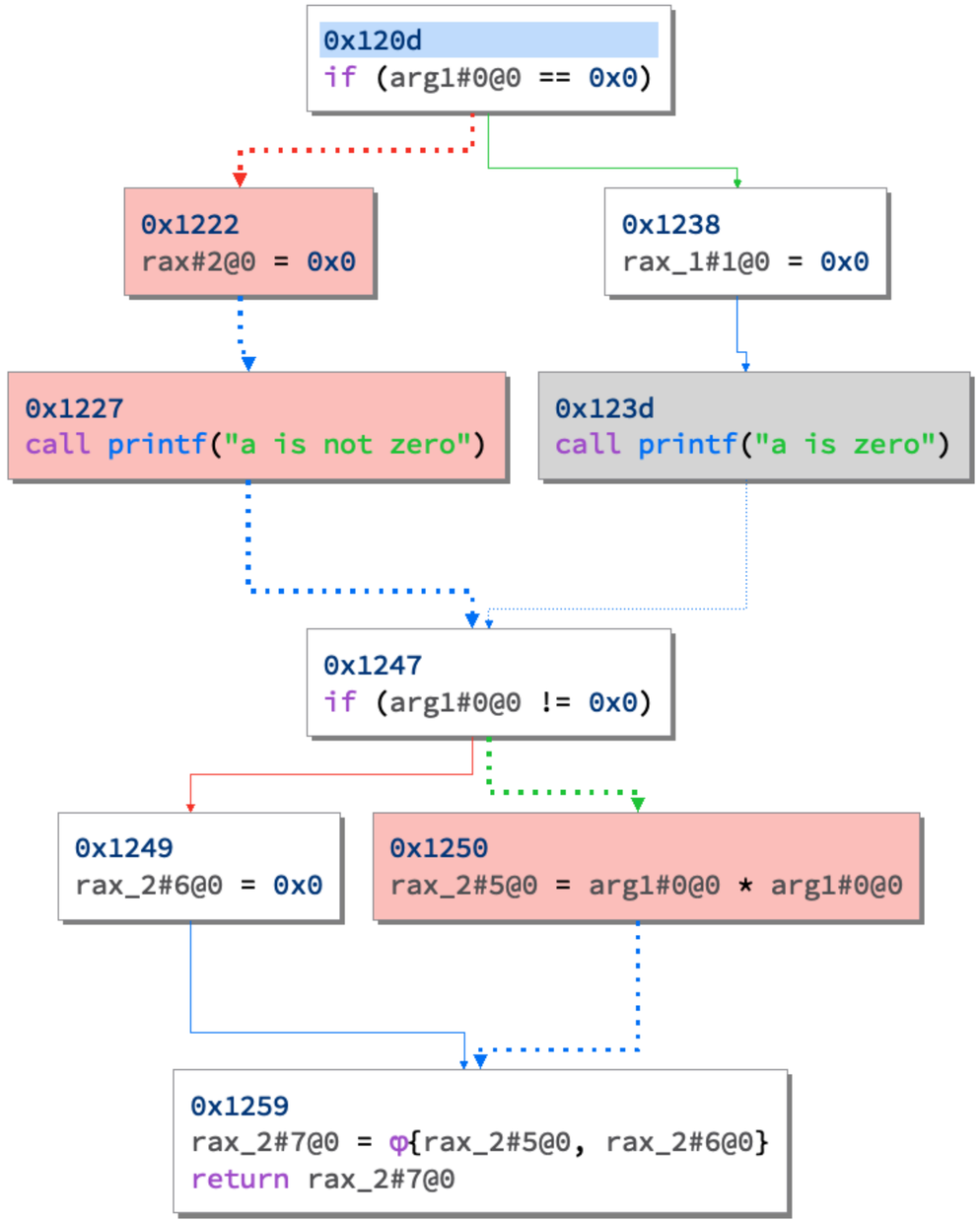
Problem Statement

Use automated analyses to interactively help vulnerability researchers manage the complexity of analyzing program binaries for vulnerabilities.

Blaze

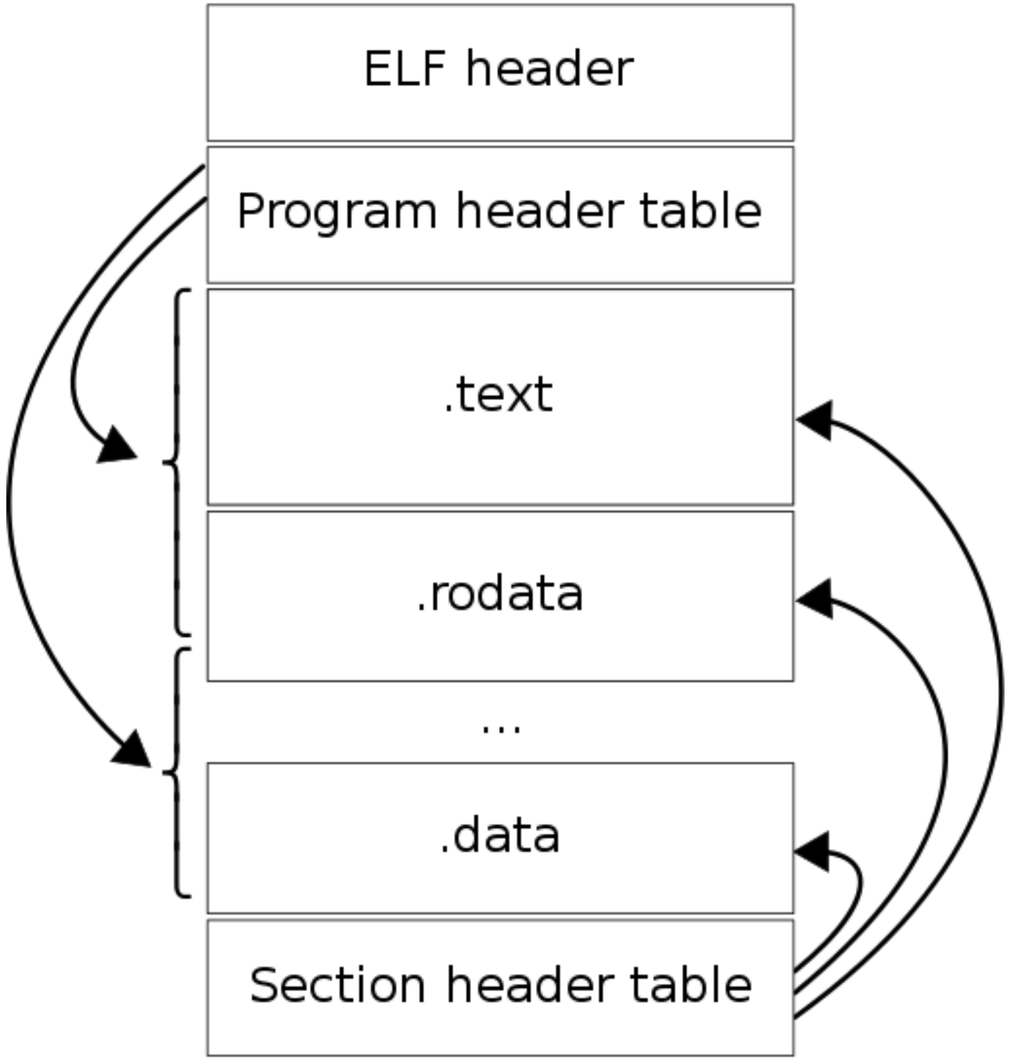
Static Analysis Framework

- Built around ***interprocedural control-flow graphs (ICFGs)*** and a typed intermediate language ***(PIL)***
- Supports symbolic analysis through satisfiability modulo theories (SMT) solvers
- Open source, written in Haskell
- Support for many executable formats and architectures via BINARYNINJA and GHIDRA



"Haskell logo." https://www.haskell.org/img/haskell-logo.svg
"Binary Ninja logo." https://www.cyberus-technology.de/assets/images/products/tycho/logo_binary_ninja.png
"Ghidra logo." https://ghidra-sre.org/images/GHIDRA_1.png

What is a Program Executable?



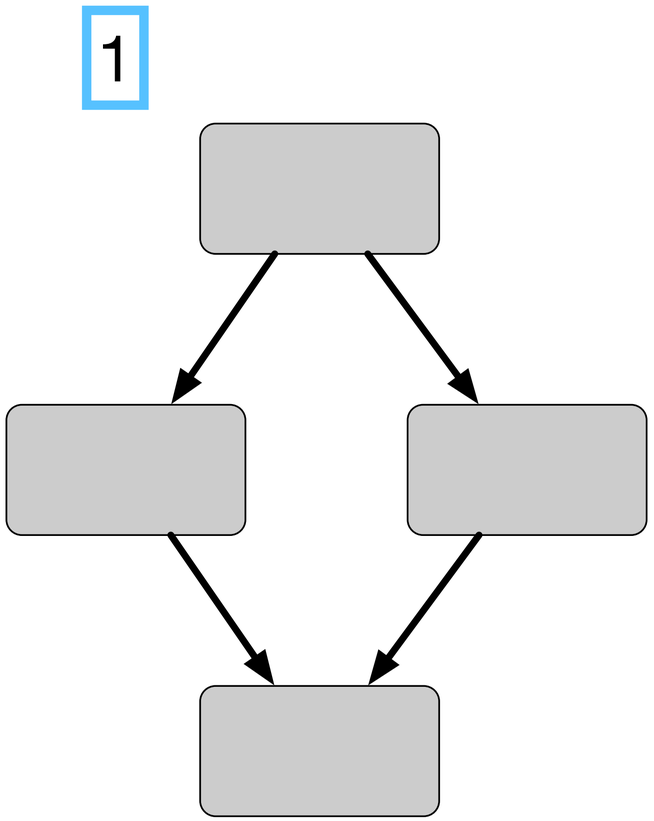
Surueña, CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, via Wikimedia Commons

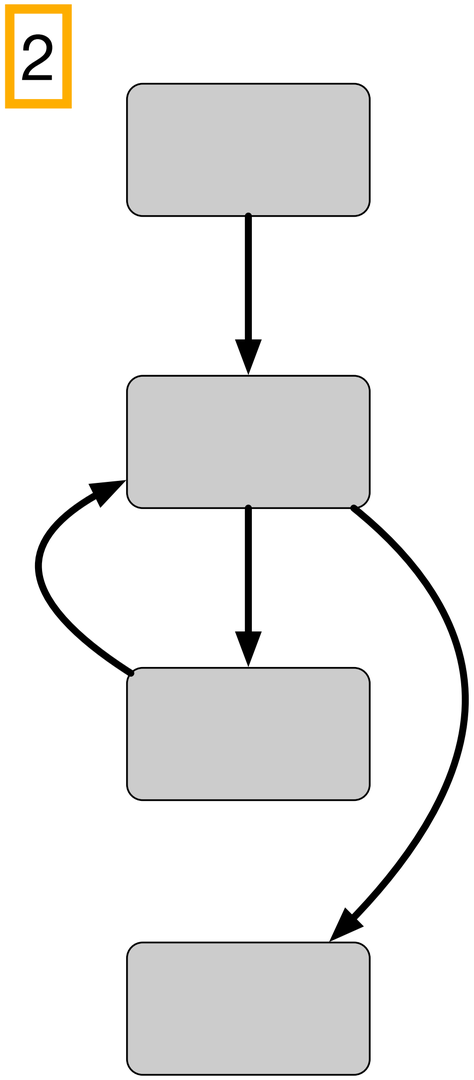
```
001011c0 f3 0f 1e fa     ENDBR64
001011c4 55              PUSH        RBP
001011c5 48 89 e5        MOV         RBP,RSP
001011c8 48 83 ec 28     SUB         RSP,0x28
001011cc 48 89 7d d8     MOV         qword ptr [RBP + local_30],inputKey
001011d0 c7 45 ec        MOV         dword ptr [RBP + lastSectionSum],0x0
         00 00 00 00
001011d7 c7 45 f0        MOV         dword ptr [RBP + j],0x0
         00 00 00 00
001011de eb 78           JMP         LAB_00101258

                     LAB_001011e0                                    XREF[1
001011e0 8b 55 f0        MOV         EDX,dword ptr [RBP + j]
001011e3 89 d0           MOV         EAX,EDX
001011e5 c1 e0 02        SHL         EAX,0x2
001011e8 01 d0           ADD         EAX,EDX
001011ea 89 45 fc        MOV         dword ptr [RBP + local_c],EAX
001011ed c7 45 f4        MOV         dword ptr [RBP + sectionSum],0x0
         00 00 00 00
001011f4 c7 45 f8        MOV         dword ptr [RBP + i],0x0
         00 00 00 00
001011fb eb 1f           JMP         LAB_0010121c

                     LAB_001011fd                                    XREF[1
001011fd 8b 55 fc        MOV         EDX,dword ptr [RBP + local_c]
00101200 8b 45 f8        MOV         EAX,dword ptr [RBP + i]
00101203 01 d0           ADD         EAX,EDX
00101205 48 63 d0        MOVSXD      RDX,EAX
00101208 48 8b 45 d8     MOV         RAX,qword ptr [RBP + local_30]
0010120c 48 01 d0        ADD         RAX,RDX
0010120f 0f b6 00        MOVZX       EAX,byte ptr [RAX]
00101212 0f be c0        MOVSX       EAX,AL
00101215 01 45 f4        ADD         dword ptr [RBP + sectionSum],EAX
00101218 83 45 f8 01     ADD         dword ptr [RBP + i],0x1

                     LAB_0010121c                                    XREF[1
0010121c 8b 05 f2        MOV         EAX,dword ptr [DAT_00104014]
         2d 00 00
```

Programs as Graphs

```
def funcA(x, y):
  result = None
  if x == y:
    result = 2 * x
  else:
    result = x + y
  return result
```

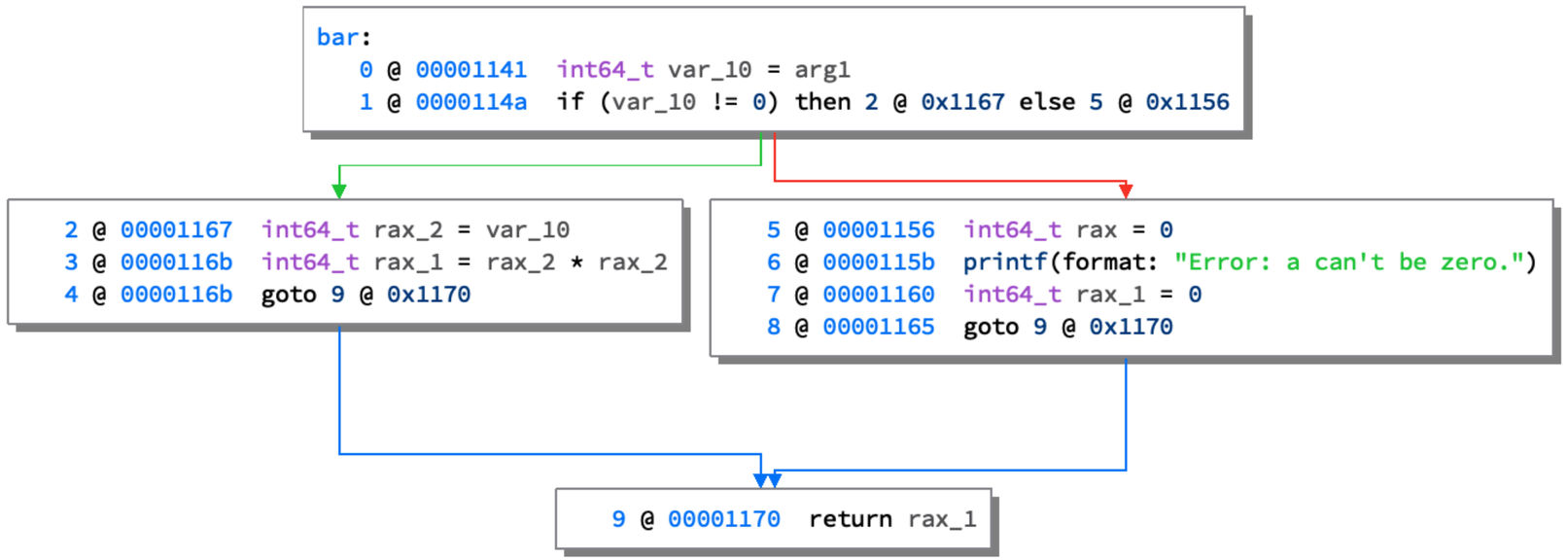
```
def funcB(xs):
  result = 0
  for x in xs:
    result += x
  return result
```





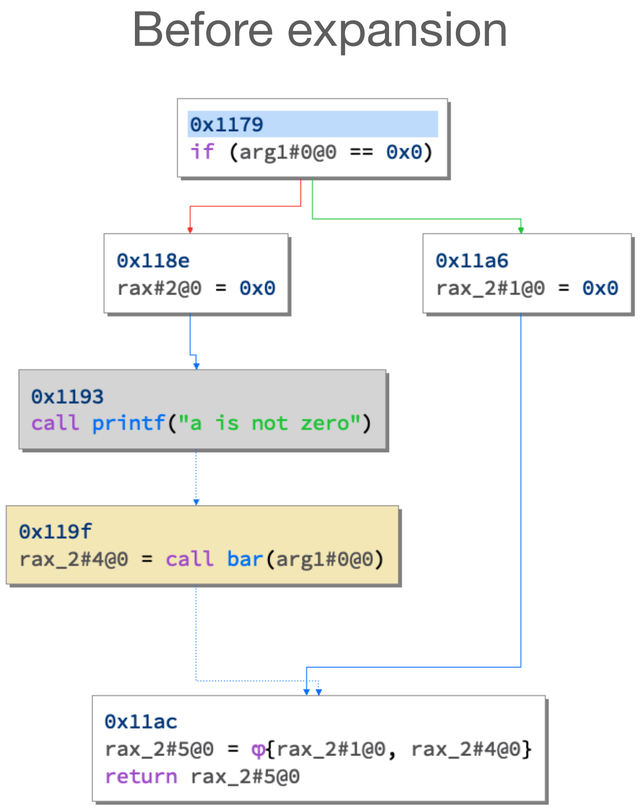
Control-Flow Graphs
(CFGs)

```c
long bar(long a) {
  if (a == 0) {
    printf("Error: a can't be zero.");
    return 0;
  }
  else {
    return a * a;
  }
```

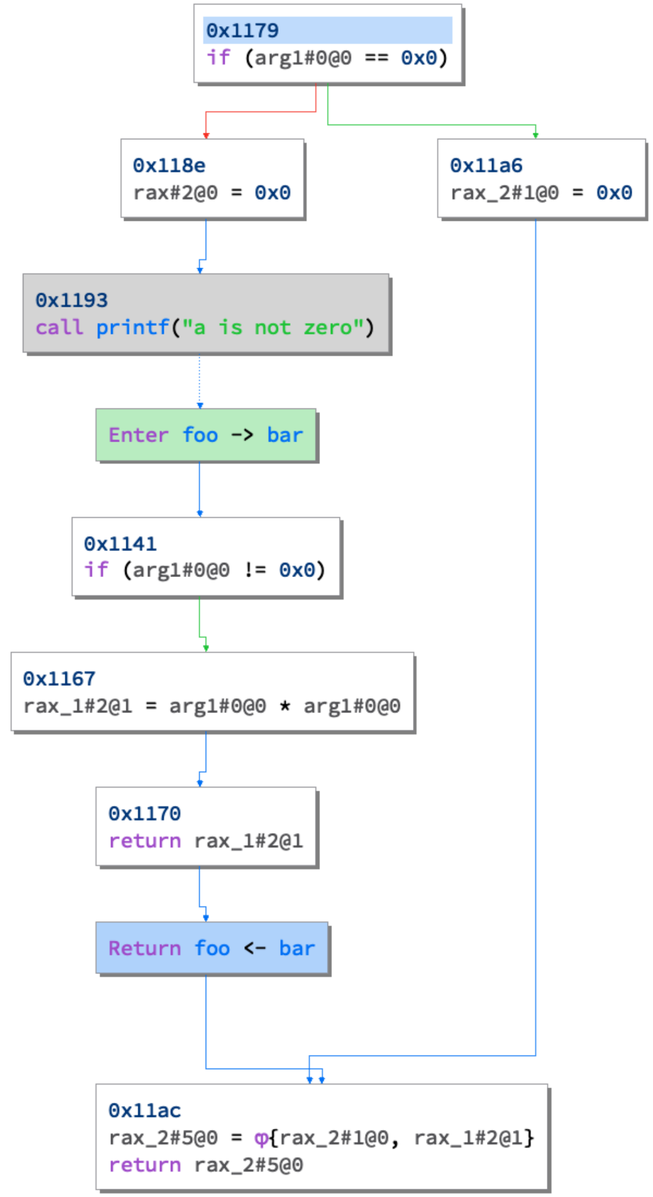


Interprocedural Control-Flow Graphs (ICFGs)

- Control-flow graphs (CFGs) that may span across function calls
- In ICFGs, function calls are expandable ***call nodes***
- ICFGs can be constructed programmatically or by user interaction



Before expansion

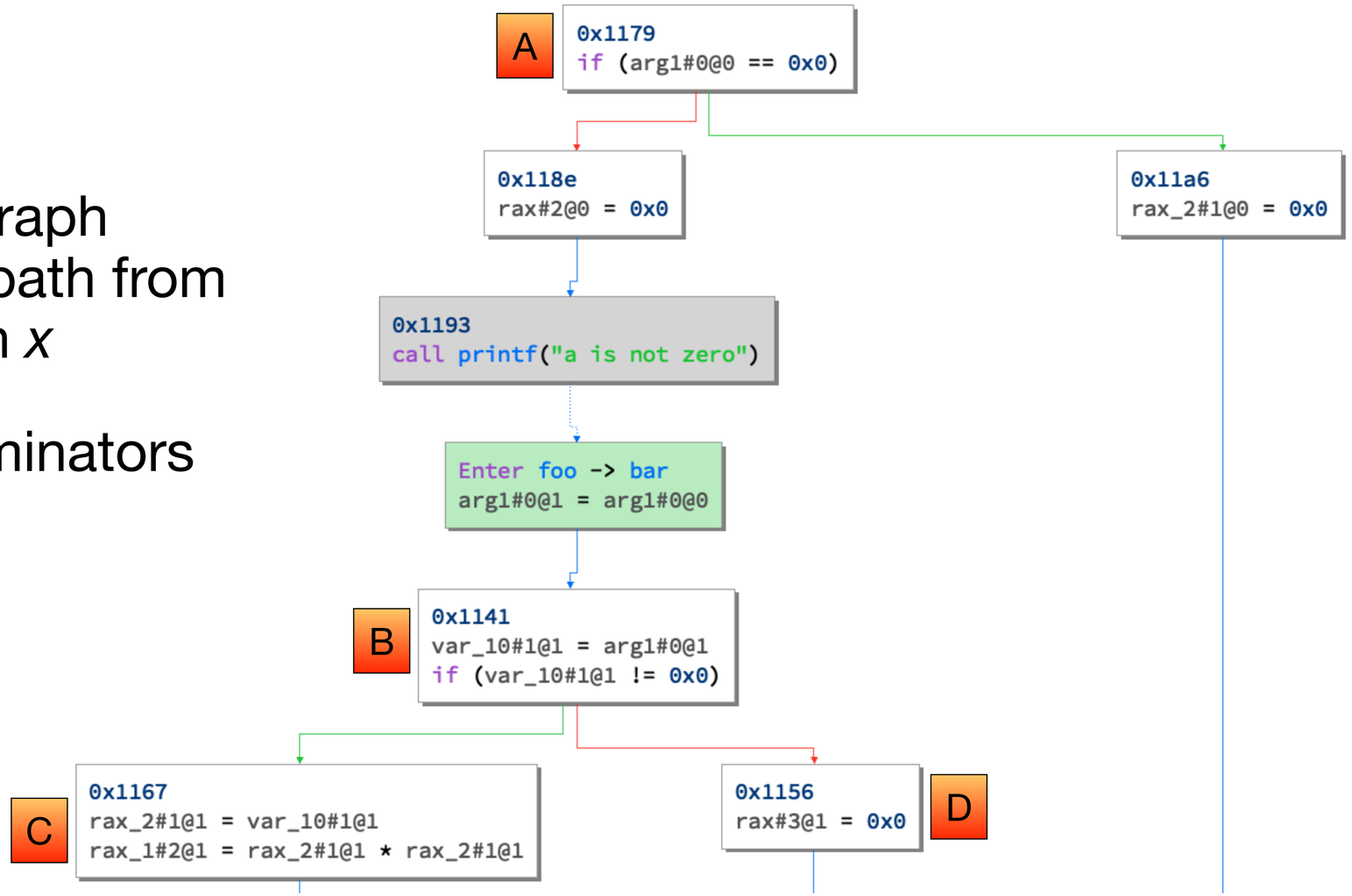


Call to `bar` expanded

Dominators

Influence of a Node

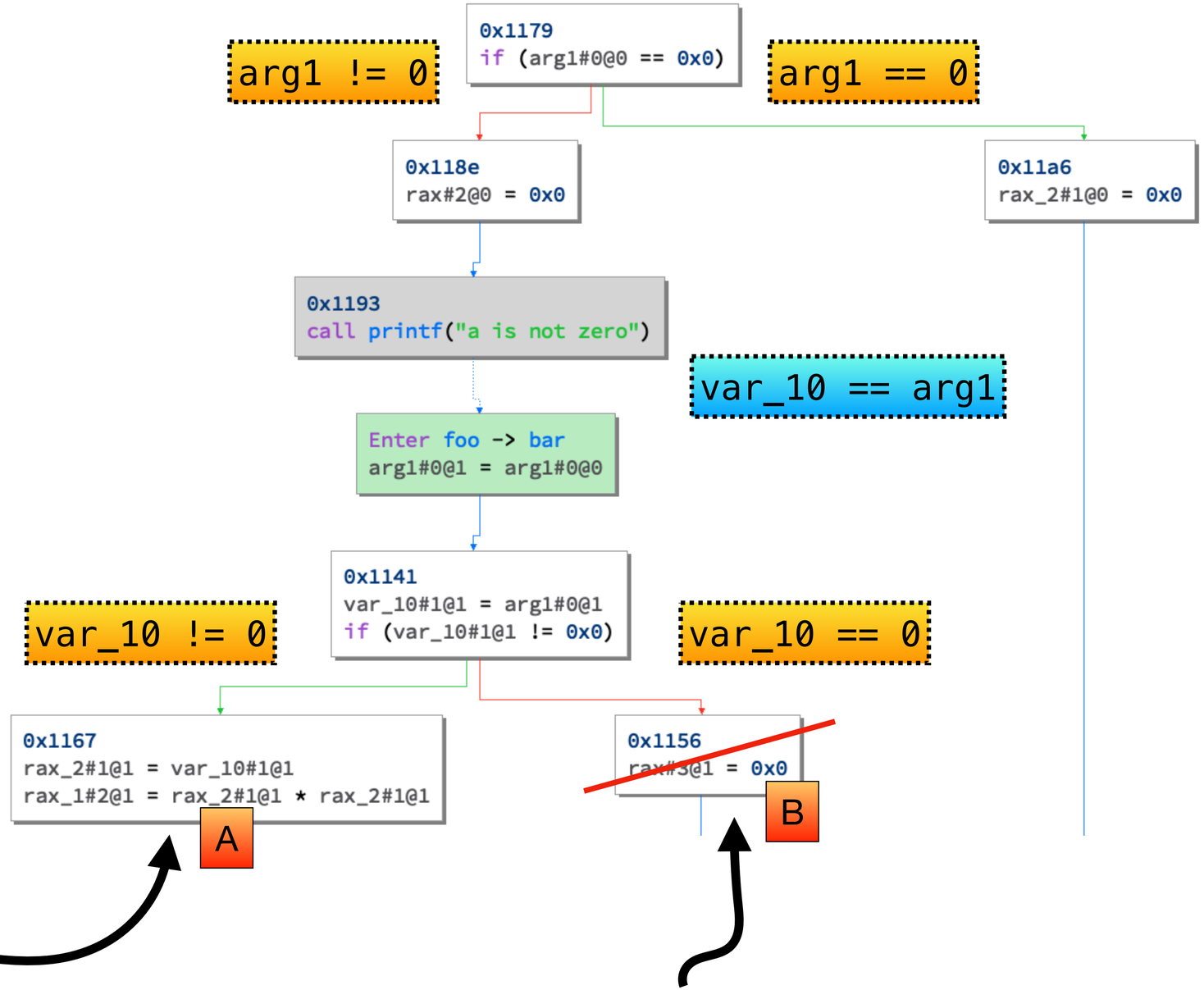
- A node *x* in a control-flow graph ***dominates*** node *y* if every path from the root to *y* passes through *x*
- A node may have many dominators



Branch Contexts

Dominating Constraints

- Nodes dominated by a conditional branch are in a ***branch context***
- Every branch context is associated with a constraint
- Branch contexts can be nested
- Use branch contexts to determine if a node is reachable

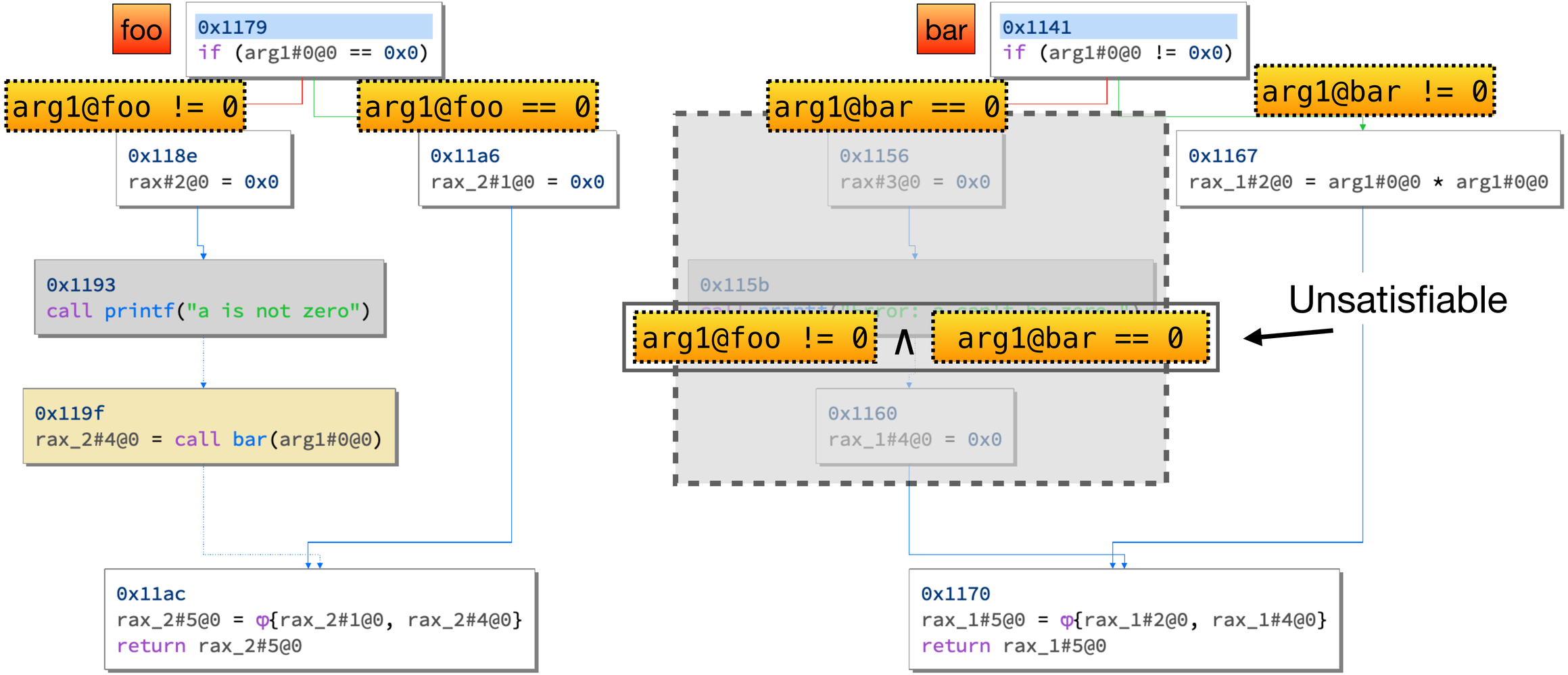


arg1 != 0 ∧ var_10 != 0 ∧ var_10 == arg1

arg1 != 0 ∧ var_10 == 0 ∧ var_10 == arg1

Constraint-Driven Transformations

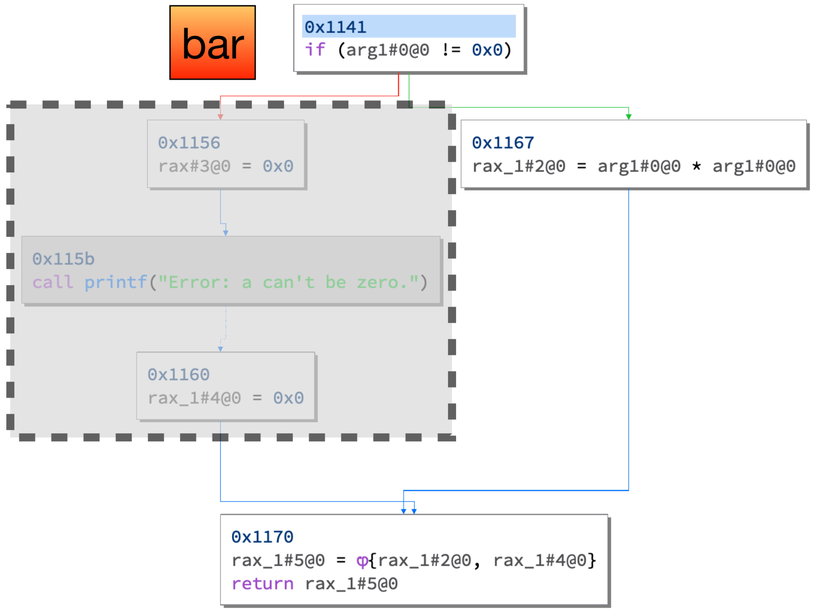
Call Expansion

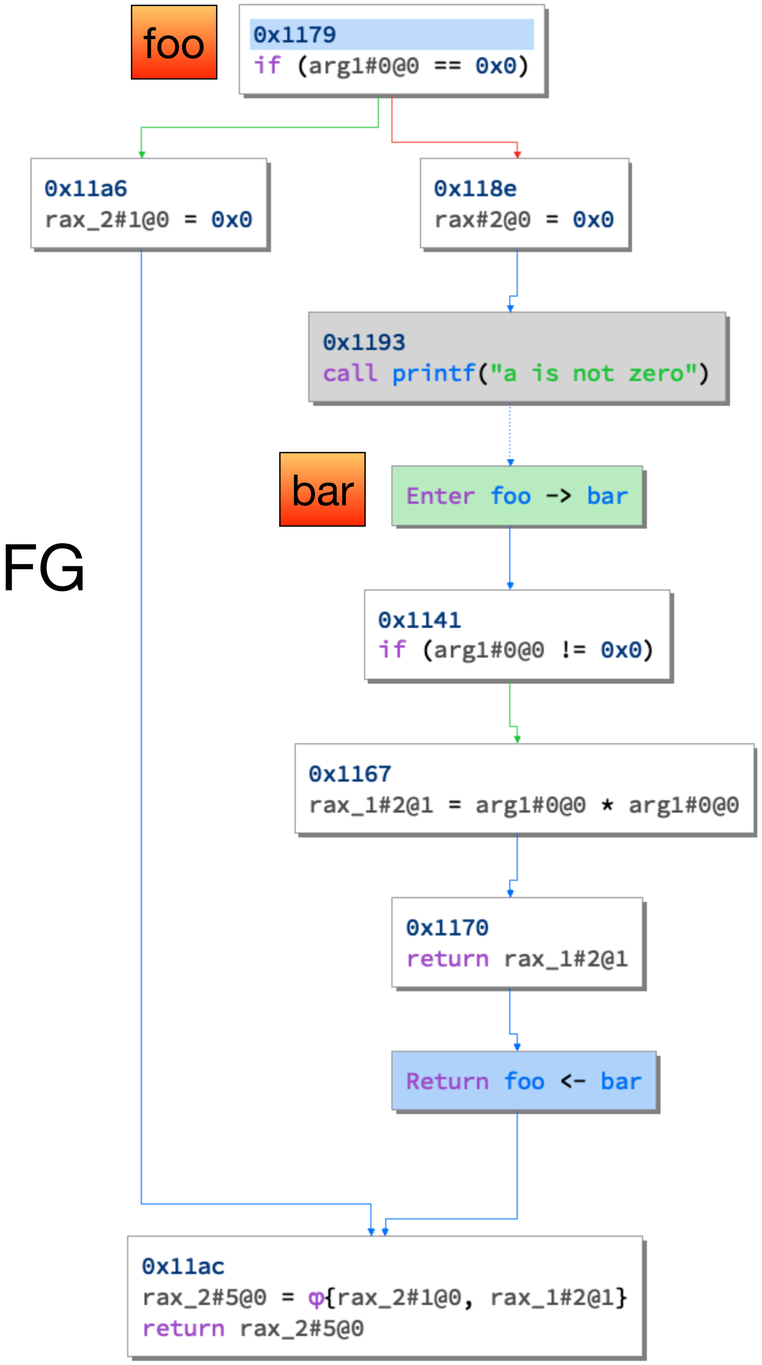


Constraint-Driven Transformations

Call Expansion

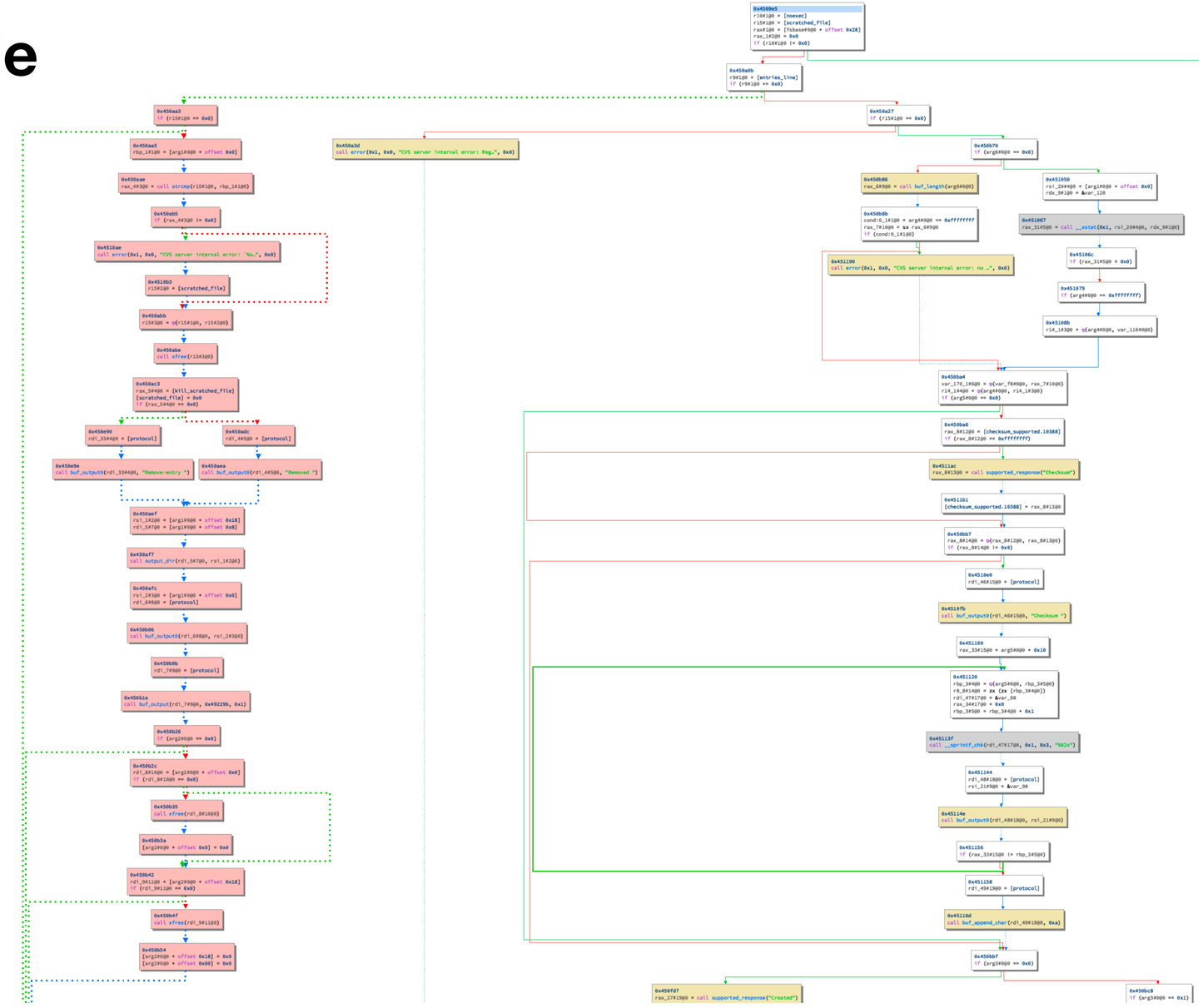
- The call to `bar` is expanded
- Infeasible path is automatically removed from the ICFG





Constraint-Driven Transformations

CVS Example



Satisfiability Modulo Theories (SMT)

- ***SMT solvers*** can check if a formula is satisfiable
- Support for integers, floats, bit vectors, arrays, and more through theories
- Describe program constraints as a mathematical formula
- Behind the scenes in Blaze, typed PIL statements are used to generate SMT formulas

```
1   (declare-const x Int)
2   (declare-const y Int)
3   (assert (< x 10))
4   (assert (> y 0))
5   (assert (= x (* y 2)))
6   (check-sat)
7   (get-model)
```

```
sat
(
  (define-fun y () Int
    1)
  (define-fun x () Int
    2)
)
```

Type System

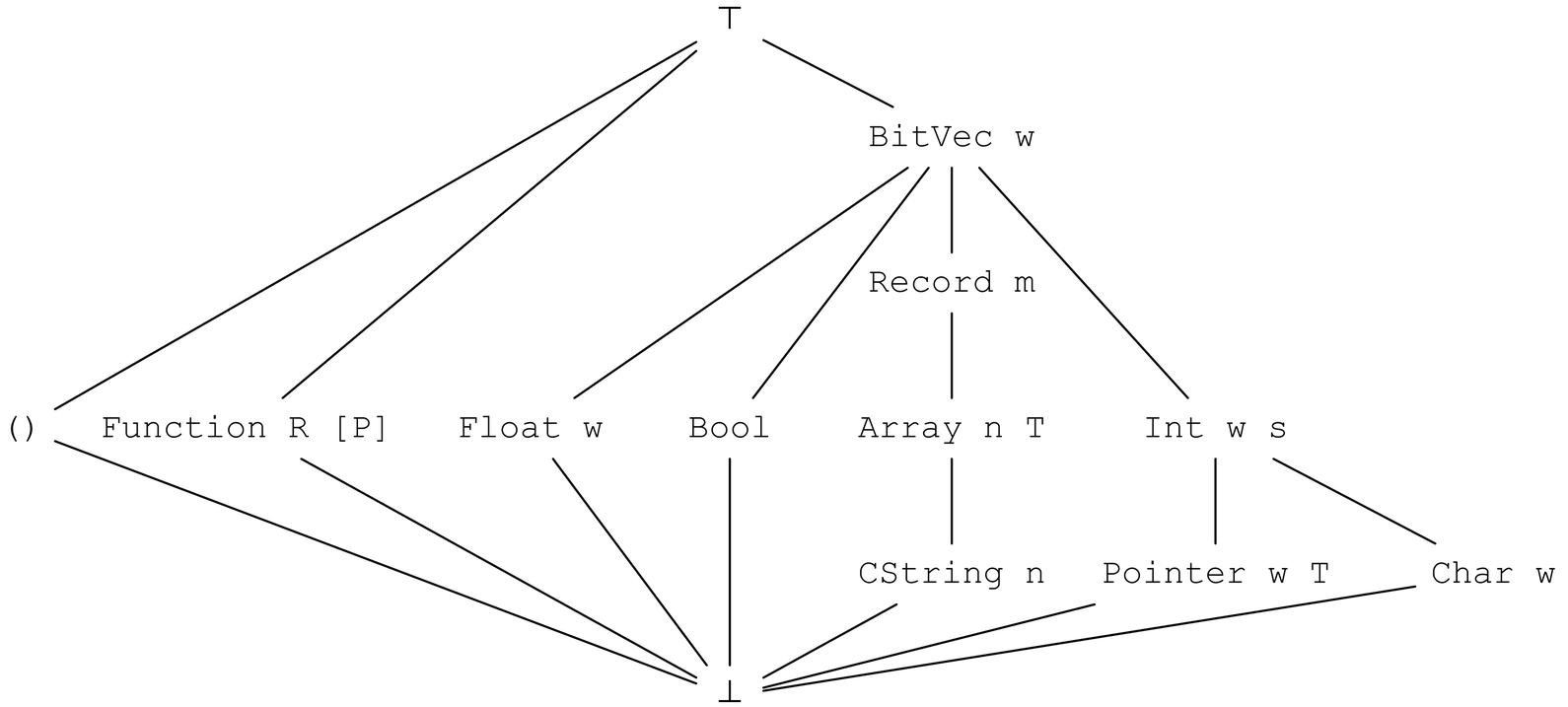
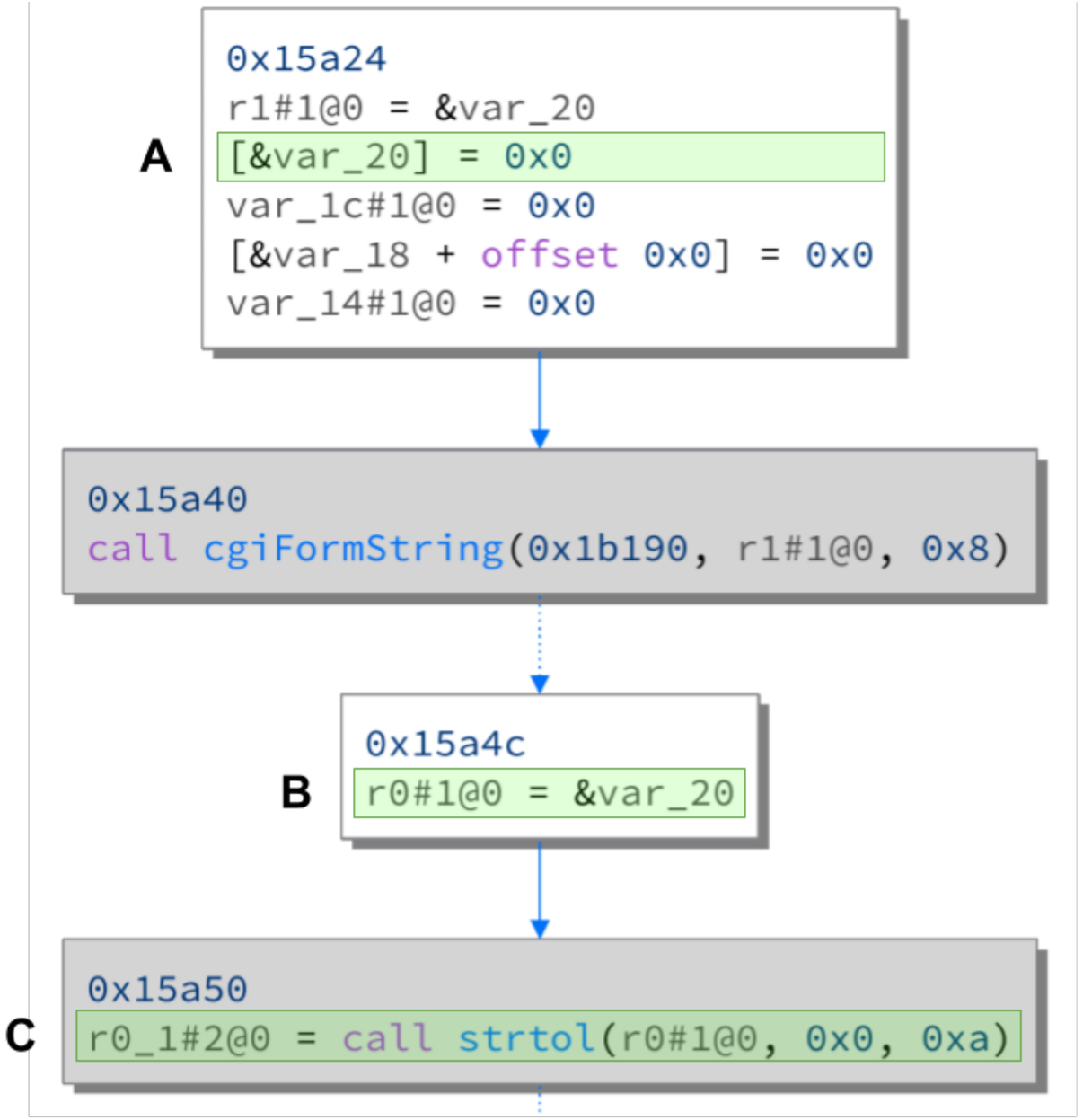


TABLE I. THE TYPES AND TYPE CONSTRUCTORS AVAILABLE IN THE PIL TYPE SYSTEM. SINGLE-LETTER METAVARIABLES ARE USED AS PLACEHOLDERS FOR UNSPECIFIED VALUES (LOWERCASE) AND TYPES (UPPERCASE).

Name	Description
`BitVec w`	Bit vector types specified by bit-width `w`.
`Int w s`	Integer types specified by bit-width `w` and signedness flag `s`.
`Pointer w T`	Pointer types specified by the pointer bit-width `w` and pointee type `T`.
`Float w`	Float types specified by bit-width `w`.
`Char w`	Character types specified by bit-width `w`.
`Bool`	The Boolean type.
`Record m`	Record types specified by a map `m` of field offsets to field types.
`Array n T`	Array types specified by a length `n` and element type `T`.
`CString n`	Null-terminated sequence of `Char 8` values specified by length `n`.
`Function R [P]`	Function types specified by return type `R` and variable number of parameter types `[P]`.
`()`	The Unit type inhabited by the single value `()`.
$\top, \bot$	The Top and Bottom types.

Type Inference

```
     &var_20   : T1
          T1 <: Pointer 32 T2
A     var_20   : T3
          T2  = T3
          T3 <: BitVec 32


B       r0#1   : T4
          T1  = T4


     strtol    : T5
          T5 <: Function T6 [T7 T8 T9]
C         T4  = T7
          T7 <: Pointer 32 T10
         T10 <: CString ?
```



T_2 and T_3 are the same type.
T_1, T_4, and T_7 are all the same type.
Simplify the rules to be expressed in terms of T_2 and T_1.

1

$$\frac{T_2 = T_3 \quad T_1 = T_4 \quad T_4 = T_7}{\begin{array}{l} T_1 <: \text{Pointer } 32\ T_2 \\ T_2 <: \text{BitVec } 32 \\ T_1 <: \text{Pointer } 32\ T_{10} \\ T_5 <: \text{Function } T_6\ [T_1\ T_8\ T_9] \\ T_{10} <: \text{CString } ? \end{array}}$$

T_2 and T_{10} must have the same type.

2

$$\frac{T_1 <: \text{Pointer } 32\ T_2 \quad T_1 <: \text{Pointer } 32\ T_{10}}{T_2 = T_{10}}$$

So then T_2 must be a subtype of `CString`.

3

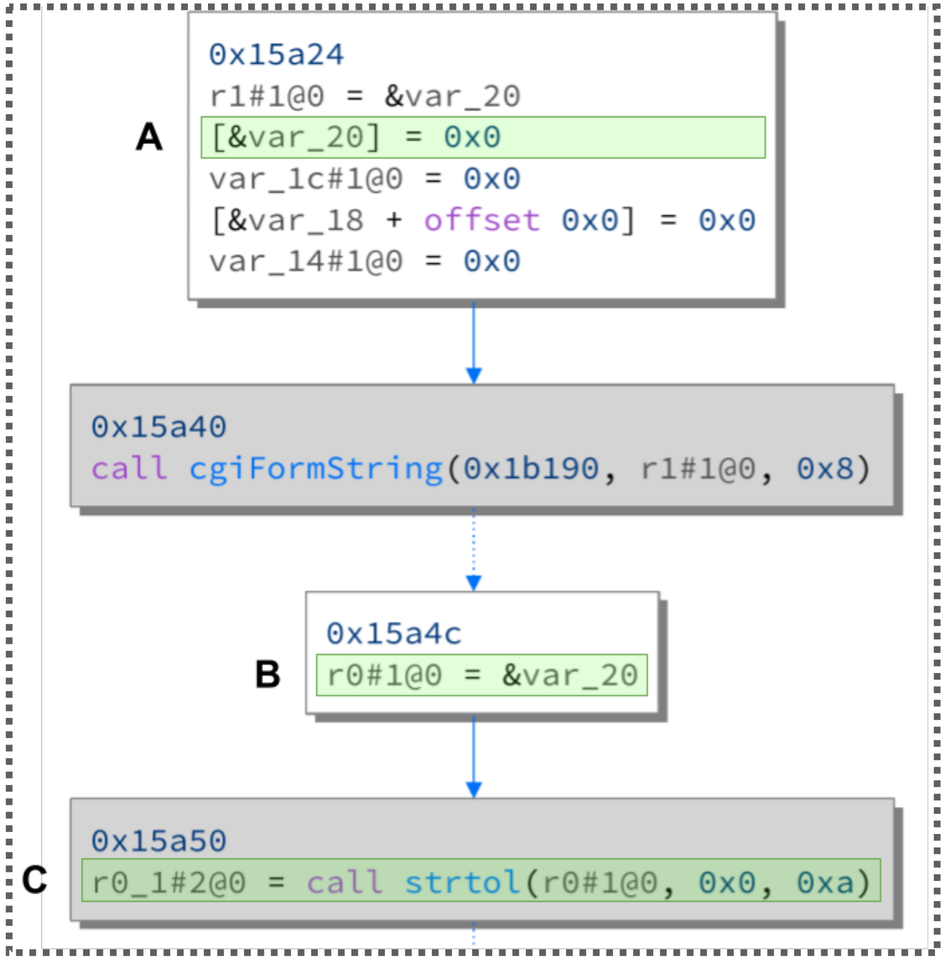
$$\frac{T_2 = T_{10}}{T_2 <: \text{CString } ?}$$

We saw before that T_2 is a 32-bit bit vector. Unify that type with `CString`.

4

$$\frac{T_2 <: \text{BitVec } 32 \quad T_2 <: \text{CString } ?}{T_2 <: \text{CString } 4}$$

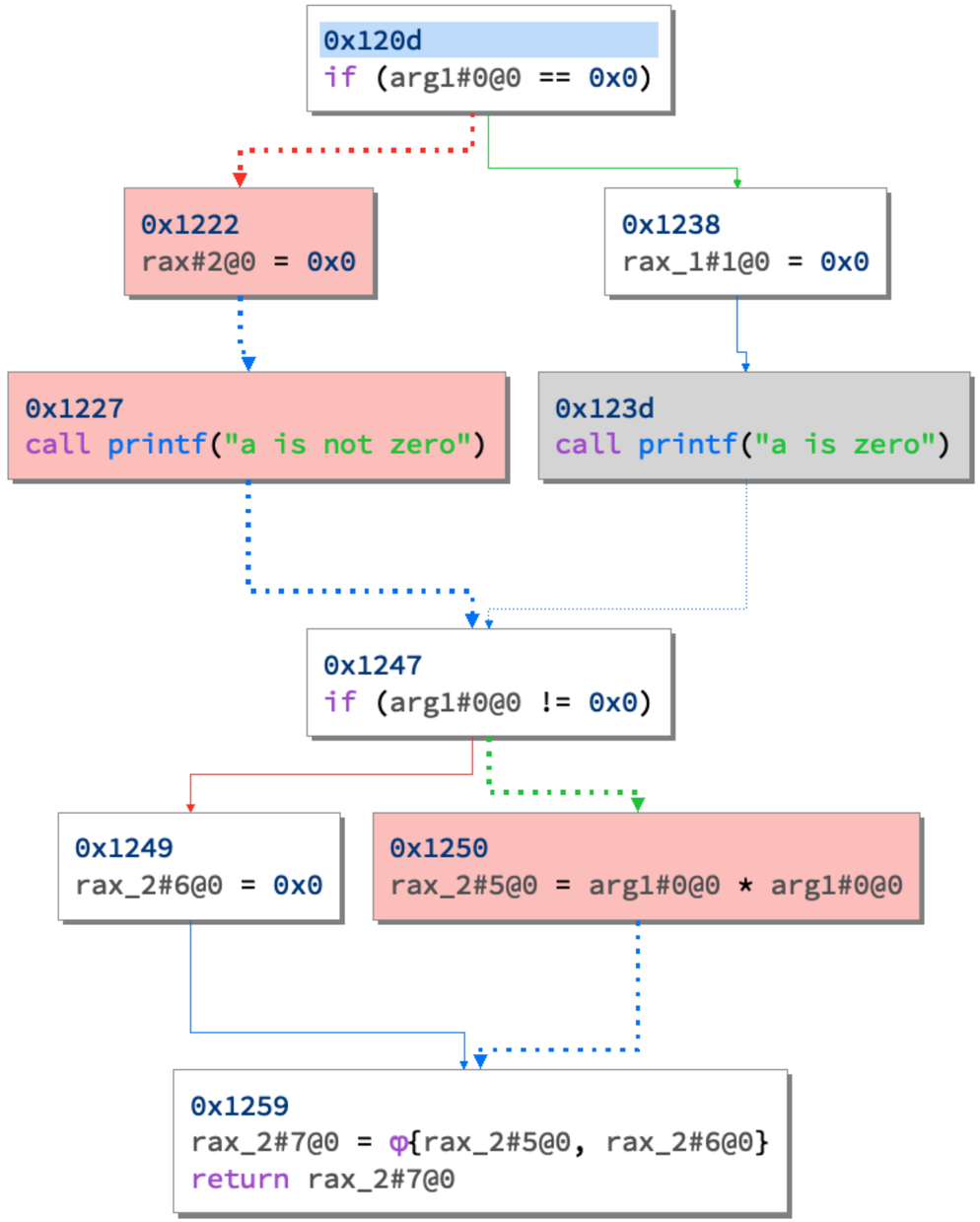
Type inference uncovers that `var_20` has type `CString 4`



Blaze

Static Analysis Framework

- Built around ***interprocedural control-flow graphs (ICFGs)*** and a typed intermediate language ***(PIL)***
- Supports symbolic analysis through satisfiability modulo theories (SMT) solvers
- Open source, written in Haskell
- Support for many executable formats and architectures via BINARYNINJA and GHIDRA



"Haskell logo." https://www.haskell.org/img/haskell-logo.svg
"Binary Ninja logo." https://www.cyberus-technology.de/assets/images/products/tycho/logo_binary_ninja.png
"Ghidra logo." https://ghidra-sre.org/images/GHIDRA_1.png