30 Years into Scientific Binary Decompilation

What We Have Achieved and What We Need to Do Next

Fish Wang

Arizona State University

4/24/2022

The crown of binary analysis

Why binary decompilation?

- Code reuse (even vendors or IP holders lose access to source!)
- Vulnerability discovery
- Source code analysis
  - Especially when analyzing binaries generated from non-traditional languages or compilers
- Program understanding
- “Software democracy”
  - Right to repair
  - Less blind trust on the entire supply chain

What do we decompile?

- Well-formed executables, library files, object files, or even machine code snippets
  - *Binaries* is not a well-defined term!
  - Usually from C code, but not always

- With binaries on Windows, Linux, MacOS, and other platforms
- With calls to external functions
- With compiler optimizations

What do we decompile? (Cont.)

- Well-formed executables, library files, object files, or even machine code snippets
  - *Binaries* is not a well-defined term!
  - Usually from C code, but not always

- No obfuscation
- No packing
- No JIT

Binary decompilation is difficult

- Compiling is a lossy process
  - Control-flow structures, variable names, types, function boundaries may all be lost!
- Compiling is sometimes seen as a protection and anti-reverse-engineering technique
- Optimizations do not care about decompilation

What have we achieved?

Impressive achievements in the past 30 years!

The first scientific attempt

Dr. Cristina Cifuentes

A Methodology for Decompilation *

Cristina Cifuentes
cifuente@fitmail.qut.edu.au

K.John Gough
gough@fitmail.qut.edu.au

School of Computing Science
Queensland University of Technology
GPO Box 2434, Brisbane, QLD 4001, Australia

A Structuring Algorithm for Decompilation *

Cristina Cifuentes
cifuente@fitmail.qut.edu.au

School of Computing Science
Queensland University of Technology
GPO Box 2434, Brisbane, QLD 4001, Australia

Reverse Compilation Techniques

by

Cristina Cifuentes

Bc.App.Sc – Computing Honours, QUT (1990)
Bc.AppSc – Computing, QUT (1989)

Submitted to the School of Computing Science
in partial fulfilment of the requirements for the degree of

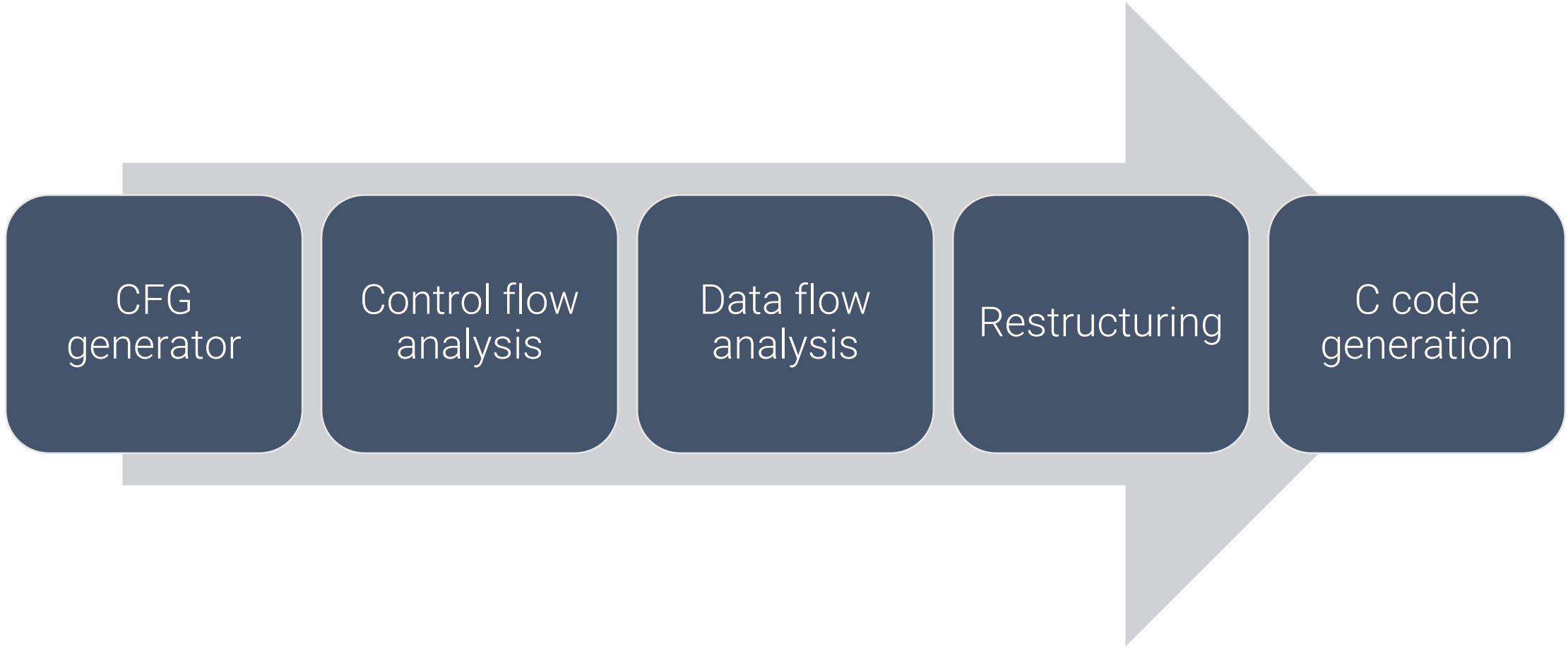
Doctor of Philosophy

at the

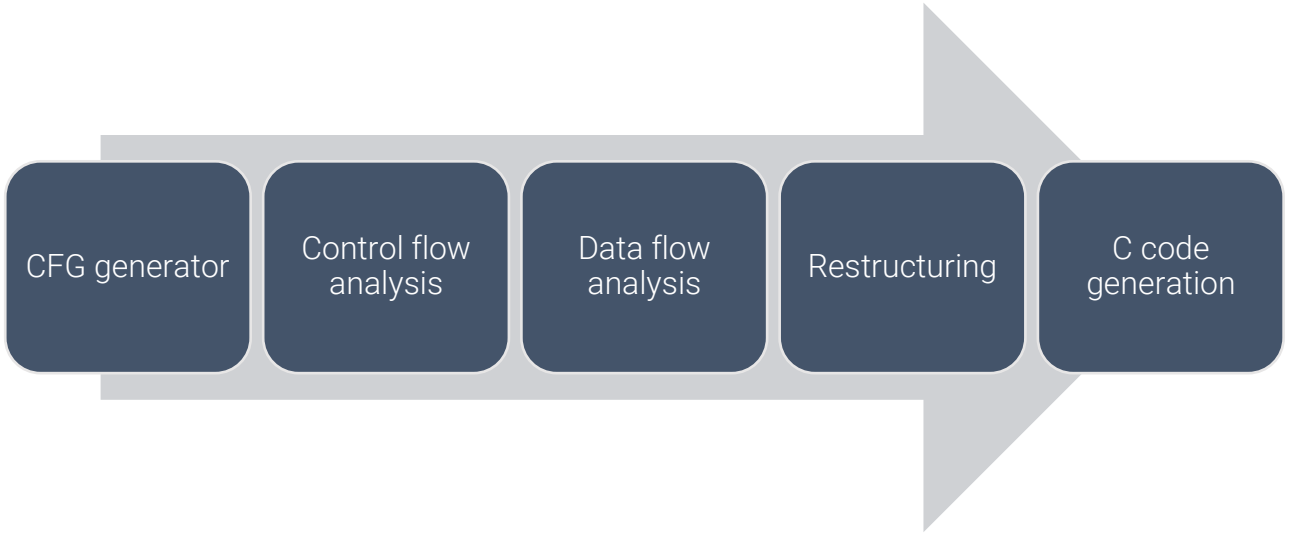
QUEENSLAND UNIVERSITY OF TECHNOLOGY

July 1994

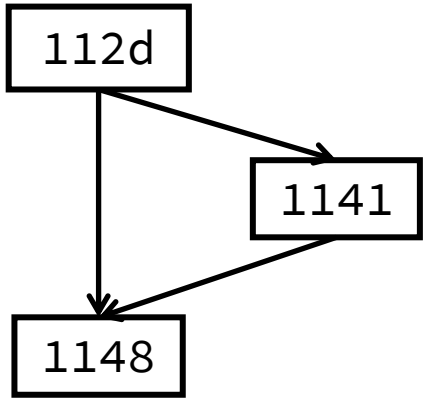
A Decompilation Pipeline



Do It Yourself



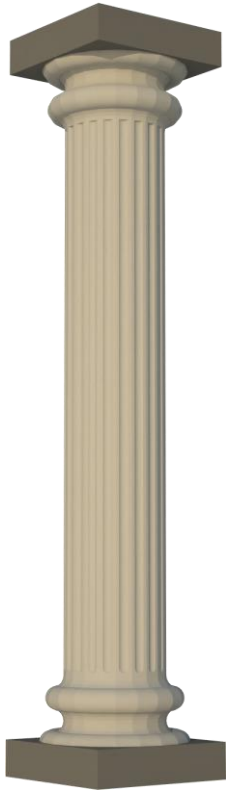

```
112d:   push rbp
112e:   mov  rbp,rsp
1131:   mov  DWORD PTR [rbp-0x14],edi
1134:   mov  DWORD PTR [rbp-0x4],0x0
113b:   cmp  DWORD PTR [rbp-0x14],0x1
113f:   jne  1148
1141:   add  DWORD PTR [rbp-0x4],0x1337
1148:   mov  eax,0x0
114d:   pop  rbp
114e:   ret
114f:   nop
```



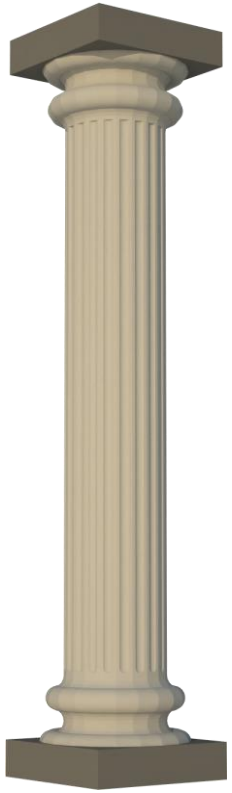

```
rbp-0x14: arg0
rbp-0x4:  var0
```

```
var0 = 0;  // 1134
if (arg0 == 1) {  // 113b
    // 1141
    var0 = var0 + 0x1337;
}
return 0;  // 1148
```

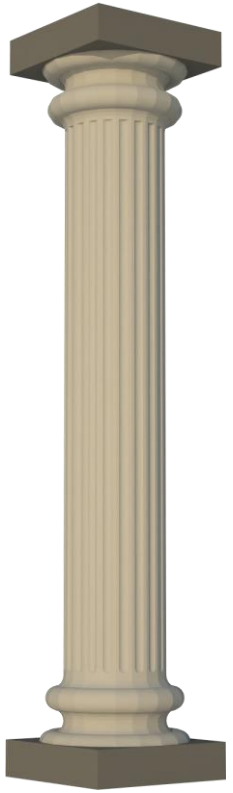
Three pillars of binary decompilation



Control flow graph recovery



Type inference



Control flow structuring

CFG rec

- Generates
  transitions

execution

TABLE III: The specifics of existing algorithms (numbered with rings like ①) and heuristics (numbered with discs like ❶).

Alg.		*Algorithms & Heuristics*	*Goals*	*Tools*
Disassembly	Linear Sweep	① Start from code addresses with symbols	Code accuracy	OBJDUMP, PSI, UROBOROS
		❶ Continuous scanning for instructions	Code coverage	OBJDUMP, PSI, UROBOROS
		❷ Skip bad opcodes	Code accuracy	OBJDUMP
		❸ Replace padding and re-disassembly	Code accuracy	PSI
		❹ Exclude code around errors	Code accuracy	UROBOROS
	Recursive Descent	② Follow control flow to do disassembly	Code accuracy	DYNINST, GHIDRA, ANGR, BAP, RADARE2
		③ Start from program entry, `main`, and symbols	Code accuracy	DYNINST, GHIDRA, ANGR, BAP, RADARE2
		❺ Function entry matching	Code coverage	DYNINST, GHIDRA, ANGR, BAP, RADARE2
		❻ Linear sweep code gaps	Code coverage	ANGR
		❼ Disassembly from targets of xrefs	Code coverage	GHIDRA, RADARE2
Symbolization	Xrefs	④ Exclude data units that are floating points	Xref accuracy	ANGR
		❽ Brute force operands and data units	Xref coverage	UROBOROS, MCSEMA, GHIDRA, ANGR
		❾ Pointers in data have machine size	Xref accuracy	UROBOROS, MCSEMA, GHIDRA, ANGR
		❿ Alignment of pointers in data	Xref accuracy	UROBOROS, MCSEMA, GHIDRA
		⓫ Pointers in data or referenced by other xrefs can be non-aligned	Xref coverage	GHIDRA, ANGR
		⓬ References to code can only point to function entries	Xref accuracy	GHIDRA
		⓭ Enlarge boundaries of data regions	Xref coverage	GHIDRA, ANGR
		⓮ Address tables have minimal size of 2	Xref accuracy	GHIDRA
		⓯ Exclude pointers that may overlap with a string	Xref accuracy	MCSEMA, GHIDRA
		⓰ While scanning data regions, use step-length based on type inference	Xref accuracy	ANGR
Function Entry	MAIN Function	⑤ Identify `main` based on arguments to `__libc_start_main`	Func coverage	ANGR, BAP
		⓱ Identify `main` using patterns in `_start/__scrt_common_main_seh`	Func coverage	DYNINST, RADARE2
	General Function	⑥ Identify function entries based on symbols	Func coverage	DYNINST, GHIDRA, ANGR, BAP, RADARE2
		⑦ Identify function entries based on exception information	Func coverage	GHIDRA
		⑧ Identify function entries based on targets of direct calls	Func coverage	DYNINST, GHIDRA, ANGR, BAP, RADARE2
		⑨ Identify function entries by resolving indirect calls	Func coverage	GHIDRA, ANGR
		⓲ Identify function entries based on prologues/decision-tree	Func coverage	DYNINST, GHIDRA, ANGR, BAP, RADARE2
		⓳ Consider begins of code discovered by linear scan as function entries	Func coverage	ANGR
CFG	Indirect Jump	⑩ Use VSA to resolve jump table targets	CFG accuracy	DYNINST, GHIDRA, ANGR
		⓴ Follow patterns to determine jump tables	CFG accuracy	DYNINST, GHIDRA, RADARE2
		㉑ Discard jump tables with index bound larger than a threshold	CFG accuracy	GHIDRA, ANGR, RADARE2
		㉒ Restrict the depth of slice for VSA	Efficiency	DYNINST, ANGR
	Indirect Call	⑪ Identify targets based on constant propagation	CFG coverage	GHIDRA, ANGR
	Tail Call	⑫ Consider a jump to the start of another function as a tail call	CFG accuracy	DYNINST, ANGR
		㉓ Determine tail call based on distance between the jump and its target	CFG accuracy	RADARE2
		㉔ A tail call and its target cross multiple functions	CFG accuracy	GHIDRA
		㉕ Tail calls cannot be conditional jumps	CFG accuracy	GHIDRA, ANGR
		㉖ A tail call tears down its stack	CFG accuracy	DYNINST, ANGR
		㉗ A tail call does not jump to the middle of a function	CFG accuracy	ANGR
		㉘ Target of a tail call cannot be target of any conditional jumps	CFG accuracy	ANGR
	Non-returning Function	⑬ Identify system calls or library functions that are known non-returning	CFG accuracy	DYNINST, GHIDRA, ANGR, BAP, RADARE2
		⑭ Identify functions with no `ret` and no tail calls that return	CFG accuracy	DYNINST, ANGR, RADARE2
		⑮ Identify functions that always call non-returning functions	CFG accuracy	BAP
		㉙ Detect non-returning functions based on fall-through after the call-sites	CFG accuracy	GHIDRA

Peng, Chenbin et.al, SoK: All You Ev

Perfect CFG Recovery

- To generate perfect decompilation, a decompiler must know at least everything that the compiler knows
  - Different compilers know different things…

- E.g., Non-returning functions

Actual Characteristics	CFG Recovery Result	Consequence
Not returning	Returning	Introducing an extra edge in the CFG
Returning	Not returning	Missing an edge in the CFG

Perfect CFG Recovery (Cont.)

- What does a compiler know?

```
NAME
       error, error_at_line, error_message_count, error_one_per_line, error_print_progname - glibc
error reporting functions

DESCRIPTION
       ... snip ...
       If status has a nonzero value, then error() calls exit(3) to terminate the program using the
given value as the exit status.
       ... snip ...
```

Type inference: What to infer

Based on locations

- Local variable types
- Global variable types
- Function (call) types

Based on types

- Pointer vs. non-pointer
- Signed vs. unsigned
- Standalone variables vs. structs
- Struct members
- Nested structs

Type inference: What to infer

Based on locations

- Local variable types
- Global variable types
- Function (call) types

Based on types

- Pointer vs. non-pointer
- Signed vs. unsigned
- Standalone variables vs. structs
- Struct members
- Nested structs

Some techniques can only tell pointers from non-pointers (without recovering the type that the pointers point to). They are insufficient for decompilation!

Type inference: How to infer

Inferred by…

- Dynamic analysis
  - Howard (NDSS'11), etc.
- Static analysis
  - Solving collected type constraints: TIE (NDSS'11), Retypd (PLDI'16)
  - Probabilistic: OSPREY (Oakland'21)
- Propagation from function argument types
  - Using external function signatures
  - Fast library function recognition (often in static binaries)

Probabilistic analysis

Structs are essentially variables that follow an allocation pattern.

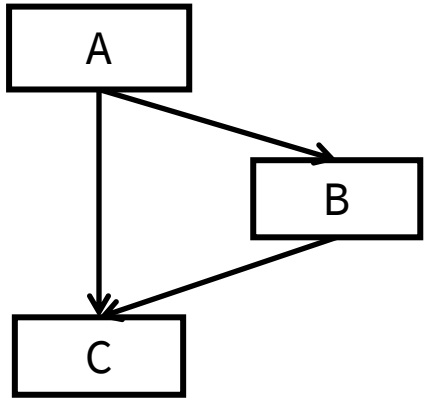
How to tell the difference between a struct on the stack and a few variables?

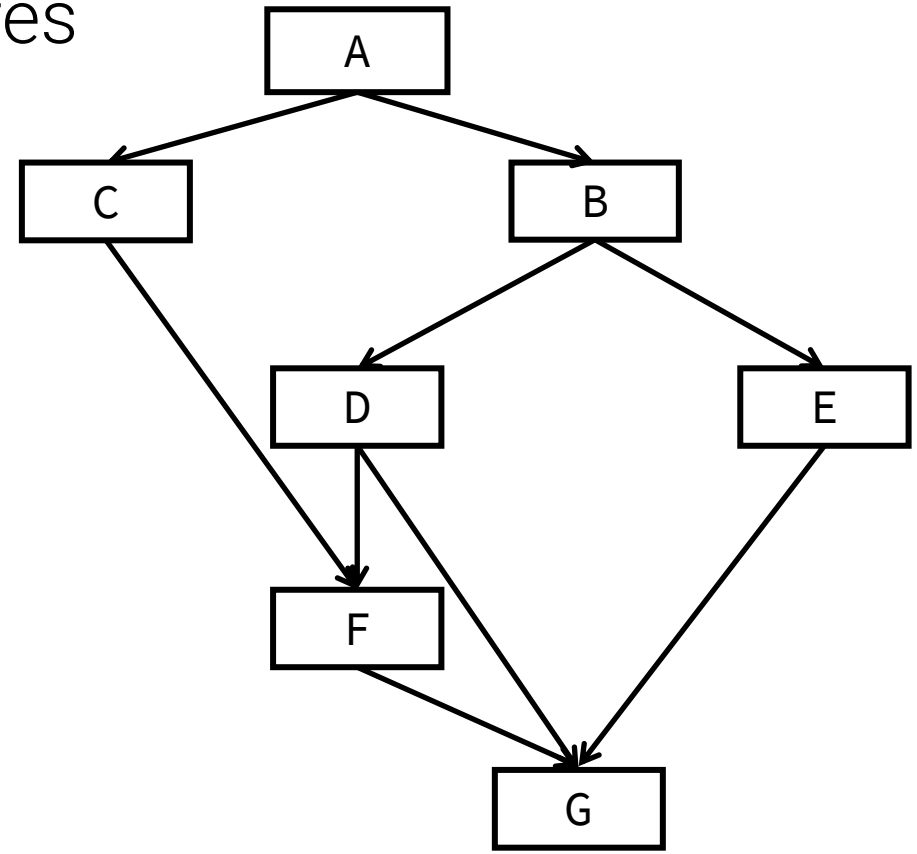
```
// stack of func_bar
int var_0;
int var_1;
int var_2;
```

```
struct foo {
  int var_0;
  int var_1;
  int var_2;
}
```

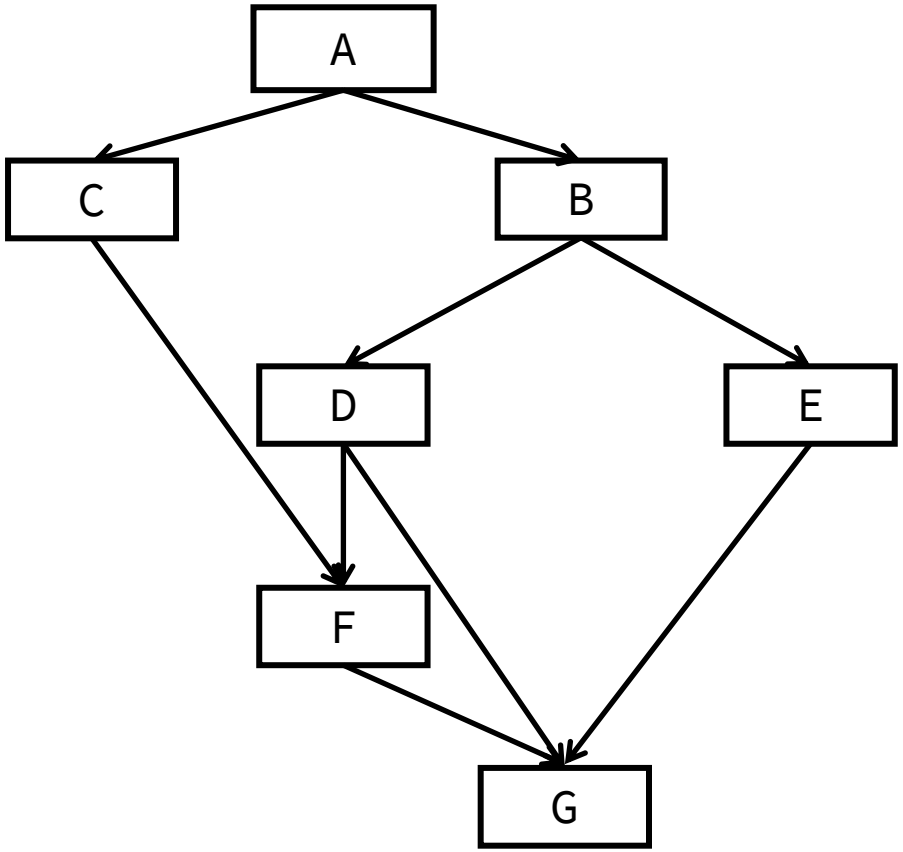
Control flow structuring

Proper structures vs improper structures

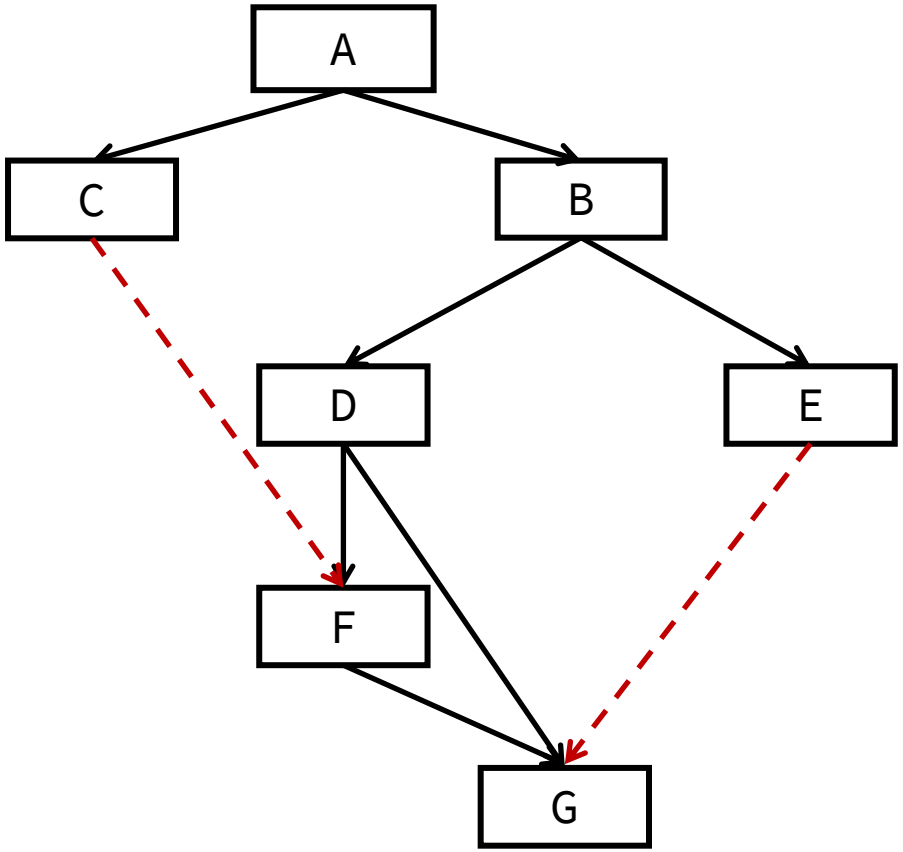




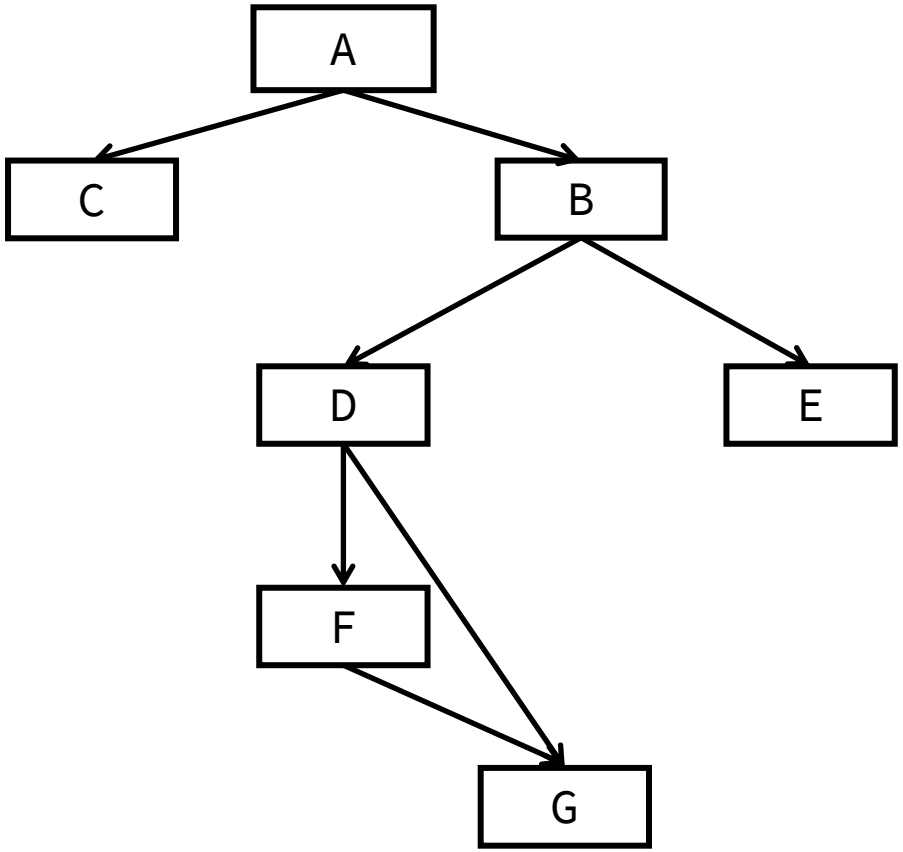
Control flow structuring (Cont.)



Control flow structuring (Cont.)

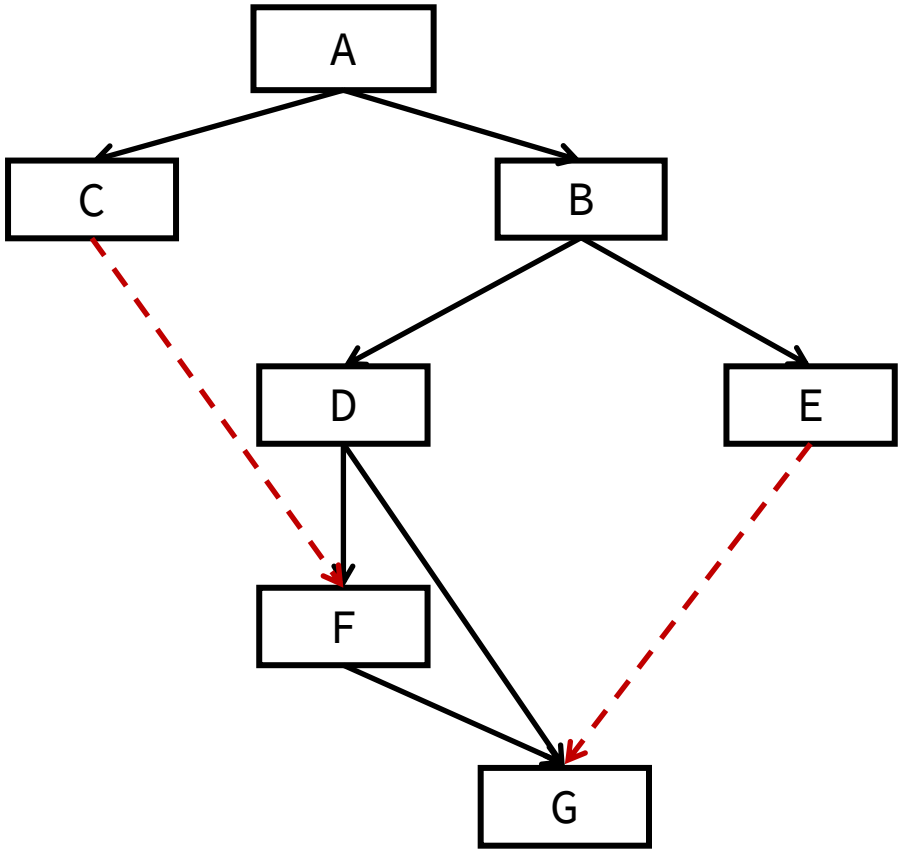


Control flow structuring (Cont.)




```
if (A) {
  C
  goto F
} else {
  if (B) {
    if (D) {
      F
    }
    G
  } else {
    E
    goto G
}
```

Control flow structuring (Cont.)




```
if (A) {
  C
  goto F
} else {
  if (B) {
    if (D) {
      F
    }
    G
  } else {
    E
    goto G
}
```

Structuring and expressibility

C is not expressible enough

Although decompilation mostly focuses on C programs, not all (control-flow) graphs are expressible in C

E.g., Ada and VB.Net support multi-level exits

```
do {
  ...
  while {
    ...
    if (...)
      break do;
  }
} while (...);
```

Innovative structuring solutions

No-more gotos (NDSS'15)

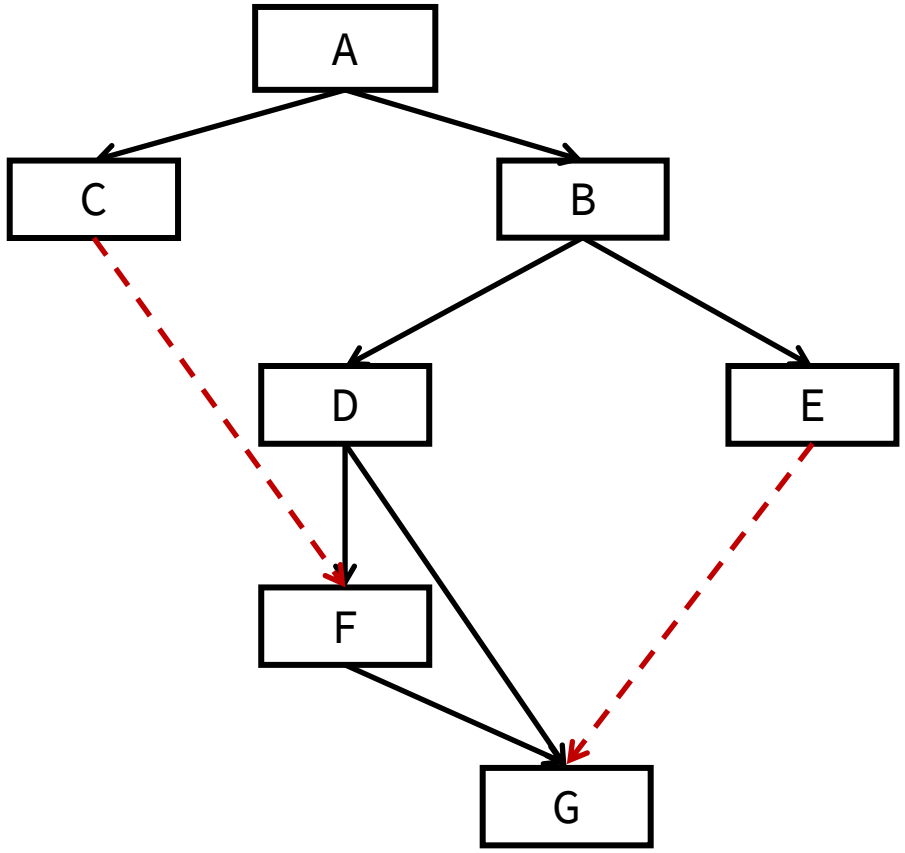
Guarding blocks with Boolean expressions

Introducing extra Boolean variables

A comb for decompiled C code (AsiaCCS'20)

Selectively duplicating nodes to create proper structures

No more gotos: Pattern-independent structuring



- Recover *reaching conditions* for each block
- Collapse and simplify conditions

Problems:

- Some gotos improve readability
- Long Boolean expressions
- Boolean simplification is expensive
  Boolean minimization is NP-hard

Research questions

Q1: Can we make C more expressible?

Why do we have to target the C language?

E.g., Binary Ninja's HLIL

Q2: What differentiates between good gotos and bad gotos?

Q3: How are these improper subgraphs created in the first place?

Can we invert the transformations causing these subgraphs?

Relevant Techniques: Known Function Recognition

- IDA FLIRT Technology
  - *Fast* Library Identification and Recognition

- AI-based function similarity detection
  - Speed is a major concern

Relevant Techniques: Data sharing and collaborative RE

- Sharing *decompilation artifacts and results* between tools is more challenging than you may think
  - DWARF is generally accepted, but…
    - IDA does not map or display names for register variables
    - IDA and Ghidra have overly strict requirements for DWARF
    - Many tools (including angr) do not even consume variable names or types from DWARF
    - DWARF is unfriendly to Git or any existing versioning tools

Relevant Techniques: Identifying C++ Classes

- C++ class hierarchies no longer exist in their original forms in binary code

- Residual information to use
  - RTTI: Run-Time Type Information
  - Virtual tables

Relevant Techniques: Simplifications and optimizations

Simplifications and optimizations in decompilation share roots with compiler optimizations

But they have different goals!

- Compiler optimizations: Optimizing code for size or speed
- Decompiler optimizations: Optimizing code for readability

Relevant Techniques: AI and evolutionary decompilation

- Using evolutionary algorithms to decompile binaries (BAR'18)
  - Byte-level equivalency
- Training a neural network to transform machine code to C code
  - Only works in a small scale
  - Hard to evaluate correctness (are two functions equivalent?)
- AI-supported decompilation quality improvement
  - Debin (CCS'18), DIRE (ASE'19), DIRTY (USENIX'22)

Commercial and Open-Source Solutions

- Hex-Rays Decompiler
- Ghidra
- Binary Ninja
- angr
- Reko decompiler
- JEB
- RetDec
- Relyze
- Snowman
- r2dec/jsdec

Commercial and Open-Source Solutions

- Hex-Rays Decompiler (Still the best overall decompiler)
- Ghidra
- Binary Ninja
- angr
- Reko decompiler
- JEB
- RetDec
- Relyze
- Snowman
- r2dec/jsdec

What do we need?

No one wants to wait for another 30 years…

Application-guided decompilation

Different decompilation applications require different decompilation capabilities

Capabilities	Automated…				Manual…			
	Vulnerability discovery	Program repair	Binary hardening	Code Reuse	Vulnerability discovery	Binary patching	Code reuse	Program understanding
Readability	✔ ✕	✕	✕	✕	✔	✔	✔	✔
Relocatability	✕	✔	✕	✔	✕	✕	✔	✕
Recompilability	✕	✔	✔ ✕	✔	✕	✕	✕	✕
Correctness	✔	✔	✕	✔	✕	✕	✕	✕
Completeness	✔	✕	✕	✔	✕	✕	✕	✕
Idiom-free	✕	✕	✕	✔ ✕	✕	✕	✕	✕

More AI-powered techniques

- End-to-end AI decompilation is probably not going to work any time soon

- Many problems in decompilation can be solved using AI!
  - Variable type inference
  - Variable name inference
  - Control-flow structuring
  - Simplifications

Exploring more decompilation capabilities

- Flexibility
  - “Why can’t I invert the condition of an if condition?”
  - “Why can’t I convert the cascading if-else into a switch-case?”
  - “Why can’t I inline this function call into its caller?”
- Bi-directional transformation (for patching)
  - Directly applying changes in decompilation on source code
- Customizable decompilation
  - Compiler has *many* flags. Where are decompiler flags?
  - Our attempt: configurable decompilation in angr

Scientific metrics

- What does *readability* mean? How can we quantifiably measure it?
- What does readability mean in decompiled code?
- What does readability mean for reverse engineers?
- Repeat: Replace readability with correctness, completeness, …

Scientific evaluation

- Evaluating the X of decompilation
  - Correctness (first attempt: ISSTA'20)
  - Conciseness
  - Readability
  - Recompilability
  - Completeness

Scientific evaluation (Cont.)

- Decompilation papers are usually not as rigorously evaluated as many other binary analysis papers
  - Coreutils binaries

    High number of duplicated functions
  - Three malware samples (not obfuscated & unpacked)
  - All binaries are compiled using the same compiler (and version)
  - Unclear how well the techniques work on real targets

Scientific evaluation (Cont.)

- Failures are useful, too!
  - Typical mindset: Only 100% successes can be published
  - Another mindset: The future work section is useless

  - We are working on scientific manuscripts, not commercials!
  - We need frank analysis and discussion to truly understand the strengths and pitfalls of any new technique

Scientific data sets

- Cross-dataset generalization
  - Brendan Dolan-Gavitt stressed *Cross-Dataset Generalization* during his keynote speech on BAR 2018
  - How well does a technique, developed and evaluated on one data set, work on a totally different data set?
  - More than one large and real data set is needed
  - Things are getting better for fuzzing, but we are still looking for our first data set for decompilation!

Scientific data sets (Cont.)

- Quality of a data set
  - Does the data set reflect characteristics of real binaries?
  - Does the data set reflect characteristics of modern compilers?
  - Does the data set capture problems that Technique A aims to solve?

- Examples of low-quality data sets
  - Only covering one compiler (or compiler version)
  - Only including binaries from a certain vendor
  - Only including functions that can be decompiled by a specific tool
  - High duplication rate (of functions, binaries, etc.)

Cross-tool collaboration

- Information sharing between tools is difficult
  - Our first attempt: BinSync
  - A unified textual representation format of reverse engineering artifacts + Git-based version management
  - Supports IDA, Binary Ninja, and angr management

https://github.com/angr/binsync

Evaluation against existing solutions

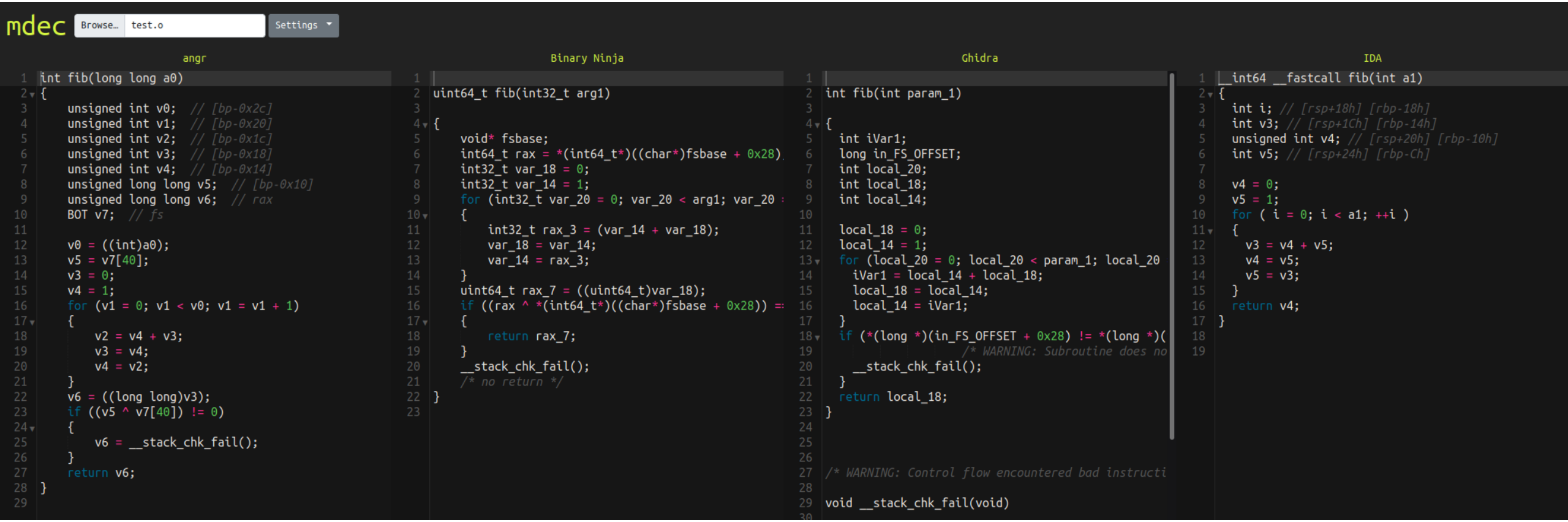
Researchers evaluate binary analysis techniques on or against angr, but …

Is angr bug-free?

How do bugs impact the evaluation result?

How much of improvement is due to a more correct implementation?

mdec: Comparing Decompilers' Output




```
int fib(long long a0)
{
    unsigned int v0;  // [bp-0x2c]
    unsigned int v1;  // [bp-0x20]
    unsigned int v2;  // [bp-0x1c]
    unsigned int v3;  // [bp-0x18]
    unsigned int v4;  // [bp-0x14]
    unsigned long long v5;  // [bp-0x10]
    unsigned long long v6;  // rax
    BOT v7;  // fs

    v0 = ((int)a0);
    v5 = v7[40];
    v3 = 0;
    v4 = 1;
    for (v1 = 0; v1 < v0; v1 = v1 + 1)
    {
        v2 = v4 + v3;
        v3 = v4;
        v4 = v2;
    }
    v6 = ((long long)v3);
    if ((v5 ^ v7[40]) != 0)
    {
        v6 = __stack_chk_fail();
    }
    return v6;
}
```

```
uint64_t fib(int32_t arg1)

{
    void* fsbase;
    int64_t rax = *(int64_t*)((char*)fsbase + 0x28)
    int32_t var_18 = 0;
    int32_t var_14 = 1;
    for (int32_t var_20 = 0; var_20 < arg1; var_20
    {
        int32_t rax_3 = (var_14 + var_18);
        var_18 = var_14;
        var_14 = rax_3;
    }
    uint64_t rax_7 = ((uint64_t)var_18);
    if ((rax ^ *(int64_t*)((char*)fsbase + 0x28)) =
    {
        return rax_7;
    }
    __stack_chk_fail();
    /* no return */
}
```

```
int fib(int param_1)

{
  int iVar1;
  long in_FS_OFFSET;
  int local_20;
  int local_18;
  int local_14;

  local_18 = 0;
  local_14 = 1;
  for (local_20 = 0; local_20 < param_1; local_20
    iVar1 = local_14 + local_18;
    local_18 = local_14;
    local_14 = iVar1;
  }
  if (*(long *)(in_FS_OFFSET + 0x28) != *(long *)(
                    /* WARNING: Subroutine does no
    __stack_chk_fail();
  }
  return local_18;
}



/* WARNING: Control flow encountered bad instructi

void __stack_chk_fail(void)
```

```
__int64 __fastcall fib(int a1)
{
  int i; // [rsp+18h] [rbp-18h]
  int v3; // [rsp+1Ch] [rbp-14h]
  unsigned int v4; // [rsp+20h] [rbp-10h]
  int v5; // [rsp+24h] [rbp-Ch]

  v4 = 0;
  v5 = 1;
  for ( i = 0; i < a1; ++i )
  {
    v3 = v4 + v5;
    v4 = v5;
    v5 = v3;
  }
  return v4;
}
```

https://github.com/mborgerson/mdec

“We haven’t seen such a case yet”

- How many cases are sufficient?
- Where are these cases coming from?
- Which decompiler works best on this binary?

Engineering effort

- Developing a decompiler is difficult. Worse, most decompiler paper authors do not release their solutions!
  - … Or only releasing their solutions after several years
- For researchers

  Re-developing decompilation techniques and research is probably not worth the effort
- We need an open-source, powerful, and flexible decompiler
  - Techniques developed on different platforms (IDA vs Ghidra) are sometimes not comparable
  - Our attempt: angr decompiler
  - Building an open binary decompiler community

Research interest and financial support!

- A more positive attitude towards *real-world-leaning* research
  - “Decompilation has no security applications.”
  - “Decompilation has ethical concerns.”
  - “With the information that decompilation needs, one can already build good vulnerability discovery techniques that work on binary code.”

- Research funding from the government and industry
  - First decompilation-related DARPA programs: CHESS and AMP
  - Decompilation will enable a wide range of new capabilities
  - Decompilation is a science

Email: fishw@asu.edu
angr: https://github.com/angr/angr
BinSync: https://github.com/angr/binsync
mdec: https://github.com/mborgerson/mdec

Questions?