Call for Contribution: A New White-Box Analytic Tool

Junwei Wang

WhibOx 2019
May 19, 2019, Darmstadt

Overview

1. Why do we need a (new) tool?
2. What does it look like?
3. How to build it?

Why do we need a white-box analytic tool?

- **designer**: to do an in-depth assessment in the design stage
- **security analyst**: to evaluate the client's solution
- To participate a CTF (such as *WhibOx* contest)

Basic requirements (for generic attacks)

- tracing registers / accessed memory
  - used in DCA, LDA, collision attack, ...
  - many libraries for visualizing/analyzing traces already exist
- injection faults
  - *e.g.* manipulating data, instructions or control flows

Advanced demands (to understand the design)

- codes transformation
  - *e.g.* single static assignment (SSA) transformation
- control-flow & data-flow analyses
  - *e.g.* data dependency analysis
- ...

Many Tools Already Exist!

Mainly based on

- **debuggers**: *GDB*, *vtrace*
- or **dynamic binary instrumentation** (DBI): *IntelPIN*, *Valgrind*
  - `SideChannelMarvels/Tracer`
- or **CPU emulators**: *Qemu*, *Unicorn*, *Ghidra*
  - `RolfRolles/GhidraPAL`

Advantages

- Efficient!
- Very little development efforts to do!

But have limited capabilities

Basic requirements

- ✓ tracing memory / registers
- ✓ injection faults

Advanced demands

- ✗ codes transformation
- ✗ control-flow & data-flow analyses
- ✗ ...

Besides, these tools only deal with binaries

$\Rightarrow$ requiring knowledges on the binary and its architecture

$\Rightarrow$ some tools are a bit hard to deploy

$\Rightarrow$ the attack might be affected by physical behavior

In fact, the architecture is not very important

- because the attack are also only software-based
- no hardware property is exploited.
- we don't want to build one tool for each architecture

Illustration: RolfRolles/GhidraPAL

```
class EmulatorTraceGenerator {
    AddressSpace defaultSpace;
    LoggingMemorizingMemoryBank defaultMemoryBank;
    MemoryState ms;
    Emulate Emulator;
    Program CurrentProgram;
    HashSet<Address> PrintfAddrs;

    // Why do I need those numbers? How can I get them?
    public static final long[] PrintfLocations = {0x04010a5l,0x04011931l,0x04011b91l
        ,0x0401deel,0x04023721,0x04023881};
    // What are those target-dependent registers initialized with those values?
    public static final String[] Reg32Names  = {"EAX","ECX","EDX","EBX","ESP","EBP
        ","ESI","EDI"};
    public static final long[]   Reg32Values = {0x28abbcl,0x611856c0l,0x0l,0x0l,0
        x28ab50l,0x28ac08l,0x200283f0l,0x6119fe9fl};
    public static final long ProgramBegin = 0x004000001;
    public static final long ProgramEnd   = 0x005201ff1;
    public static final long StackBegin   = 0x0028ab501;
    public static final long StackEnd     = 0x0028ABFC1;
    public static final long InputBegin   = 0x0028ABE81;
    public static final long ExecBegin    = 0x004011C51;
    public static final long ExecEnd      = 0x004023811;
```

```
public EmulatorTraceGenerator(Program currentProgram)
{
    CurrentProgram = currentProgram;
    // I don't care about the programming languages and architectures
    SleighLanguage l = (SleighLanguage)currentProgram.getLanguage();

    // Initialize AddressSpace objects
    defaultSpace  = currentProgram.getAddressFactory().getDefaultAddressSpace↩
        ();
    AddressSpace registerSpace = currentProgram.getAddressFactory().↩
        getRegisterSpace();
    AddressSpace uniqueSpace   = currentProgram.getAddressFactory().↩
        getUniqueSpace();

    // Create MemoryPageBank objects for the address spaces
    boolean isBigEndian = l.isBigEndian();

    // memory tracing are hooked here (see later)
    defaultMemoryBank  = new LoggingMemorizingMemoryBank(defaultSpace,  ↩
        isBigEndian, 4096, acc, StackBegin, StackEnd);
    MemoryPageBank registerMemoryBank = ...;
    MemoryPageBank uniqueMemoryBank   = ...;
```

```
        // Create and initialize the MemoryState
        ms = new MemoryState(l); ms.setMemoryBank(registerMemoryBank); ms.↩
             setMemoryBank(defaultMemoryBank);

        // Initialize the BreakTable
        BreakTableCallBack bt = new BreakTableCallBack(l);

        // Create the emulator object
        Emulator = new Emulate(l, ms, bt);

        PrintfAddrs = new HashSet<Address>();
        for(long printfRef : PrintfLocations)
            PrintfAddrs.add(defaultSpace.getAddress(printfRef));
    }


    void Init() {
        defaultMemoryBank.Accesses = new ArrayList<Byte>();
        SleighLanguage l = (SleighLanguage)CurrentProgram.getLanguage();
        VarnodeTranslator vt = new VarnodeTranslator(CurrentProgram);

        for(int i = 0; i < Reg32Names.length; i++)
            ms.setValue(l.getRegister(Reg32Names[i]), Reg32Values[i]);
    }
```

```
    ArrayList<Byte> execute(long desInput) {
        Address eaBeg = defaultSpace.getAddress(ExecBegin);
        Address eaEnd = defaultSpace.getAddress(ExecEnd);
        Init();
        byte[] desArr = new byte[8];
        for(int i = 0; i < 8; i++)
            desArr[i] = (byte)((desInput >> (8*i)) & 0xFF1);
        defaultMemoryBank.setChunk(InputBegin, 8, desArr);
        Emulator.setExecuteAddress(eaBeg);
        while(!eaEnd.equals(Emulator.getExecuteAddress())) {
            // why I need to do this?
            if(PrintfAddrs.contains(Emulator.getExecuteAddress()))
                Emulator.setExecuteAddress(Emulator.getExecuteAddress().add(5L));
            Emulator.executeInstruction(true);
        }
        return defaultMemoryBank.Accesses;
    }

}
```

Illustration: RolfRolles/GhidraPAL

```
class LoggingMemorizingMemoryBank extends MemoryPageBank {
    ArrayList<Byte> Accesses = new ArrayList<Byte>();

    // Log the low byte of all addresses targeted by 1-byte reads
    public int getChunk(long addrOffset, int size, byte[] res, boolean stop) {
        int iRes = super.getChunk(addrOffset, size, res, stop);
        if(size == 1) {
            Accesses.add((byte)(addrOffset&0xFF1));
        }
        return iRes;
    }

    // Log the low byte of all addresses targeted by 1-byte writes
    public void setChunk(long offset, int size, byte[] val) {
        super.setChunk(offset, size, val);
        if(size == 1)
            Accesses.add((byte)(offset&0xFF1));
    }
}
```

gōng yù shàn qí shì bì xiān lì qí qì

工欲善其事，必先利其器

To do a good job, an artisan needs the best tools.

—— Chinese idiom

In Dream

we work with

- ✓ a Swiss army knife (basic/advanced features all-in-one)
- ✓ independent with programming languages (PL) and architectures

- ✓ open source: get community involved & contributed
- ✓ cross-platform (working with Windows, Linux and MacOS)
- ✓ usable, extendable, and maintainable

Windows

Mac

Linux

qián rén zāi shù hòu rén chéng liáng

前人栽树，后人乘凉

To enjoy the benefits of the hard work of one's predecessors.

—— Chinese idiom

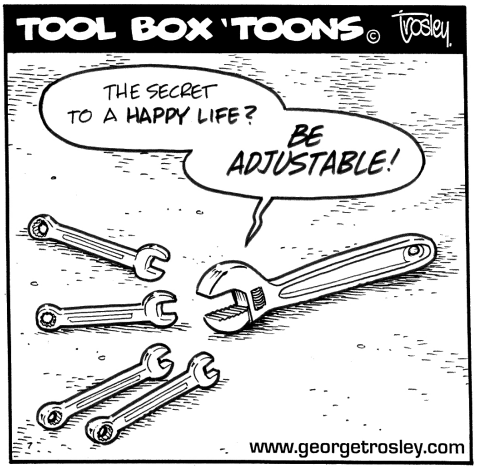
In the Beginning, a Compiler

duplicates $N \times M$ times of (a three-phrase design)

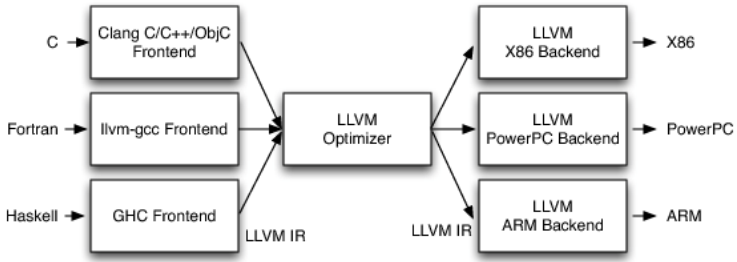
Source: `https://www.aosabook.org/en/llvm.html`

for N front-ends (*i.e.* PL) and M target back-ends (*i.e.* CPU architectures).

The Ideal is $N + M$



LLVM Architecture



Source: `https://www.aosabook.org/en/llvm.html`

- A complete de-coupling of the front-ends and back-ends
- Thanks to an intermediate representation (IR) independent with PLs and architectures

Our Secret Is Also to Use LLVM IR

```c
int increment(int a) {
  a++;
  return a;
}
```

incremement.c

```
define i32 @increment(i32) {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = add nsw i32 %3, 1
  store i32 %4, i32* %2, align 4
  %5 = load i32, i32* %2, align 4
  ret i32 %5
}
```

clang -emit-llvm -S -c increment.c

```
define i32 @increment(i32) {
  %2 = add nsw i32 %0, 1
  ret i32 %2
}
```

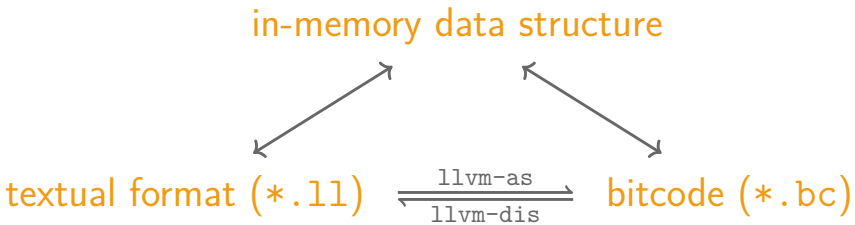
opt -mem2reg -S increment.ll

LLVM IR is Better than Assembly

- LLVM IR is a complete code representation
- with RISC-like instruction sets
- but independent with PLs and architectures
- strong typed variables with simple type system
- infinite virtual registers in SSA form

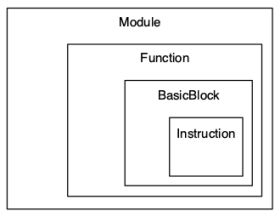
```
define i32 @increment(i32) {
  %2 = add nsw i32 %0, 1
  ret i32 %2
}
```

Three Isomorphic Forms

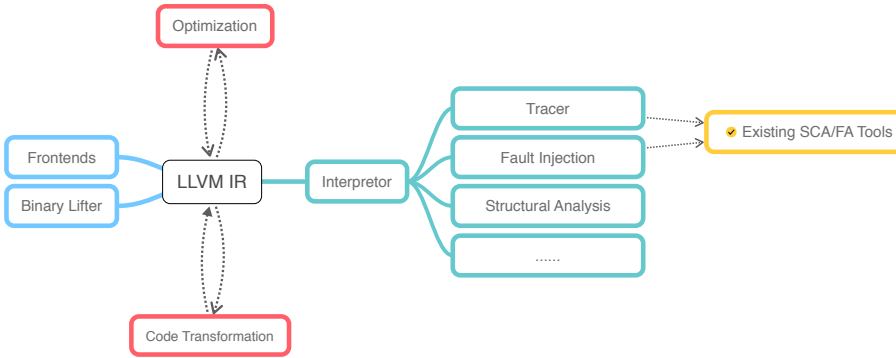


In-memory data structure

- **Module** contains Functions/GlobalVariables
  - Module is the unit of compilation/analysis/optimization
- **Function** contains BasicBlocks/Arguments
  - Functions roughly correspond to functions in C
- **BasicBlock** contains list of instructions
  - Each block ends in a control flow instruction
- **Instruction** is typed opcode + operands



Our Proposal


Optimization
Frontends
Binary Lifter
LLVM IR
Interpretor
Tracer
Fault Injection
Structural Analysis
.....
Existing SCA/FA Tools
Code Transformation

Advantages

- easy to implement the basic requirements
- built-in features to support advanced features (control-flow/data dependency analysis)
- architecture & language independent
  - $\Rightarrow$ we only need to understand one instruction sets
  - $\Rightarrow$ no re-synchronization, free to filter samples in acquisition time, $\cdots$
- big community: many LLVM IR based tools can be directly forked and used

Possible Drawbacks

- Performance (might not be a real issue)
  - it is only a cipher
  - as efficient as emulators
- lifting binary to LLVM IR is not easy
  - difficult to disassemble accurately and recover control flows
  - use existing tools: McSema, $\cdots$

LLVM IR Interpreter

```
void Interpreter::visitBinaryOperator(BinaryOperator &I);
void Interpreter::visitLoadInst(LoadInst &I); {}
void Interpreter::visitStoreInst(StoreInst &I); {}

void Interpreter::run() {
  while (!ECStack.empty()) {
    // Interpret a single instruction & increment the "PC".
    ExecutionContext &SF = ECStack.back();  // Current stack frame
    Instruction &I = *SF.CurInst++;         // Increment before execute

    // Track the number of dynamic instructions executed.
    ++NumDynamicInsts;

    LLVM_DEBUG(dbgs() << "About to interpret: " << I);
    visit(I);   // Dispatch to one of the visit* methods...
  }
}
```

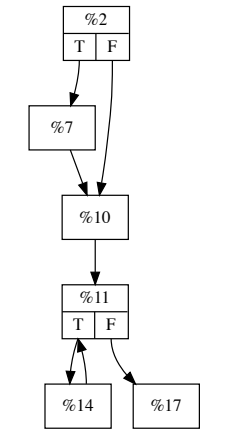
```
void Interpreter::visitBinaryOperator(BinaryOperator &I) {
  ExecutionContext &SF = ECStack.back();
  Type *Ty    = I.getOperand(0)->getType();                    // operand type
  GenericValue Src1 = getOperandValue(I.getOperand(0), SF); // 1st operand
  GenericValue Src2 = getOperandValue(I.getOperand(1), SF); // 2nd operand
  GenericValue R;                                              // result

  // dispatch each operation
  switch (I.getOpcode()) {
    case Instruction::Add:   R.IntVal = Src1.IntVal + Src2.IntVal; break;
    case Instruction::Sub:   R.IntVal = Src1.IntVal - Src2.IntVal; break;
    case Instruction::Mul:   R.IntVal = Src1.IntVal * Src2.IntVal; break;
    // ... ...
  }

  // save the result
  SetValue(&I, R, SF);
}
```

Tracing / Injecting Faults

```
void Interpreter::run(Action &action) {
  while (!ECStack.empty()) {
    // ... ...
    visit(I, &action);   // Dispatch to one of the visit* methods...
  }
}

void Interpreter::visitSomeOperator(SomeOperator &I, Action &action) {
  // ... ...

  // insert our action here
  if (!act) action.act(&I, R, SF);
  // save the result
  SetValue(&I, R, SF);
}
```

```
class Action {
public:
  virtual ~Action() {}
  virtual void act(Value *V, GenericValue Val, ExecutionContext &SF) = 0;
};

class TraceAction : public Action {
public:
  void act(Value *V, GenericValue Val, ExecutionContext &SF) {
    // save Val to trace
    trace.addSample(Val);
  }
};

class FaultAction : public Action {
private: FaultModel model;
public:
  void act(Value *V, GenericValue Val, ExecutionContext &SF) {
    // inject faults
    model.inject(Val);
  }
};
```

Control-Flow Analysis



CFG for 'exampleCFG' function

- Basic blocks are already organized in a CFG
- Visualization is also implemented

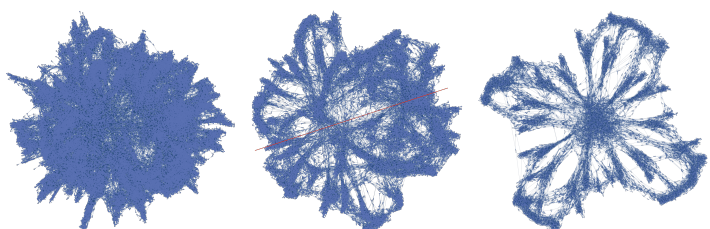
```
$ opt -dot-cfg-only cfg.ll -o cfg.dot
$ dot -Tpdf cfg.dot -o cfg.pdf
```

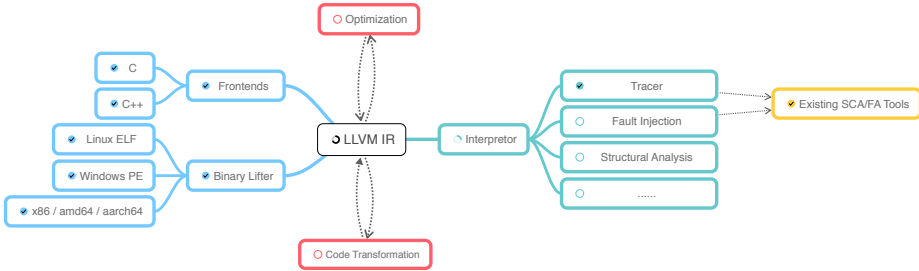
- Many projects exist for further analysis based on CFG

```
int exampleCFG(int a, int b) {
  if (b == 2) { ++b; }
  while (a < 5) { ++a; }
  return a + b;
}
```

Single Static Assignment

- Variables can only assigned once, and can only be used after assignment
- Registers are already organized in a SSA format
- Facilitate structural analysis
  - used to break *WhibOx* 2017 winning challenge





- Source codes will be open very soon
- Looking forward to your contributions:
  - development
  - ideas / features
  - issues

Thank You !