LLFI and the Art of Fault Injection: Part 2

Karthik Pattabiraman
University of British Columbia (UBC)

Recap (Part 1)

- Fault Injection Goals and Techniques
- LLFI Features and Philosophy
- LLFI for hardware faults
- LLFI for software faults
- Conclusion

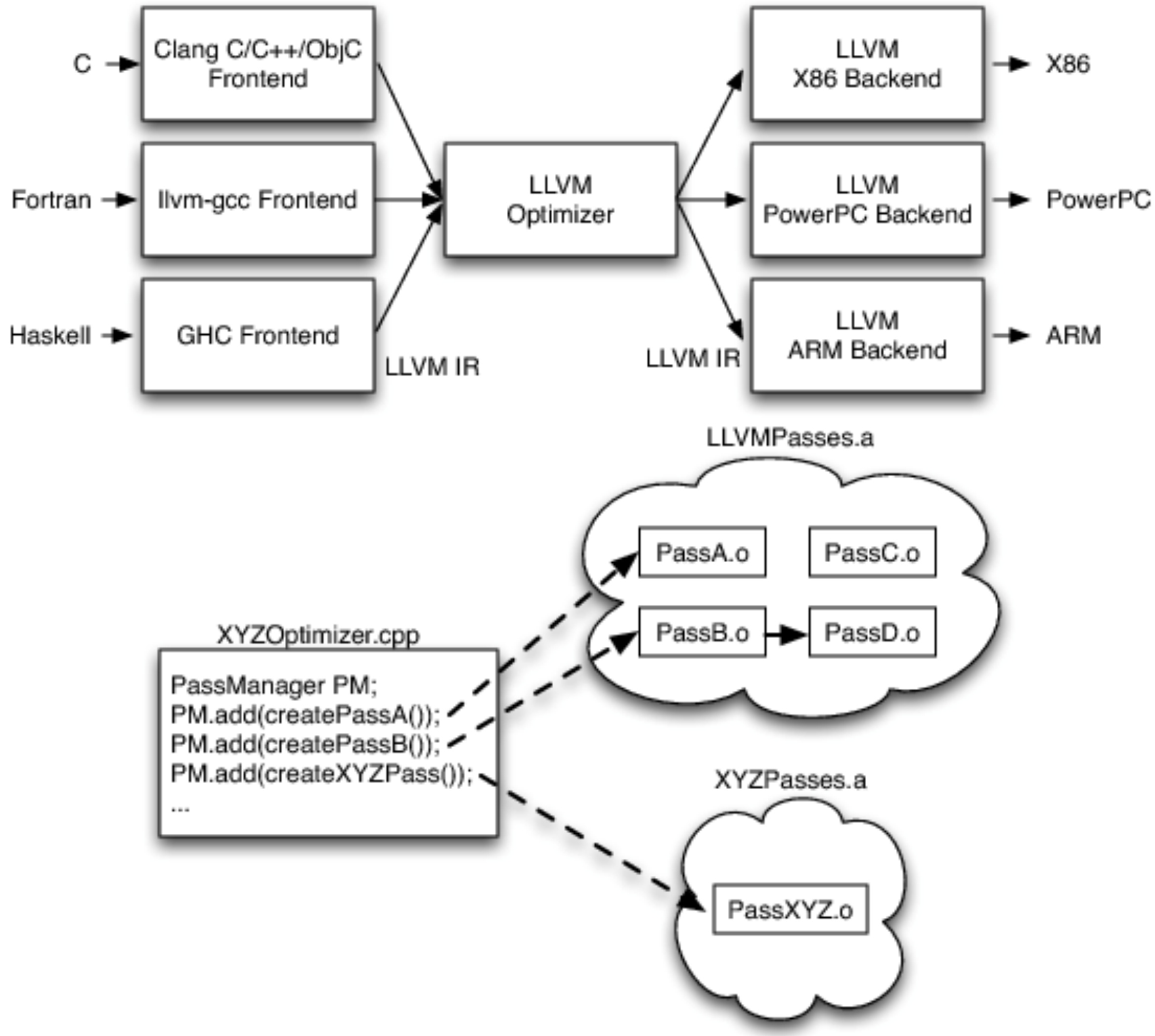
Outline (Part 2)

- LLFI Philosophy and Architecture
- Writing Hardware Fault Injector
- Writing Software Fault Injector
- Applications and conclusions

LLFI Philosophy

- Fault Injection is a combination of compile-time and runtime
- At compile time, identify instructions to be fault injected and instrument them with calls to a runtime library
- At runtime, actually inject the faults at the appropriate time when the instrumented library functions are called

LLVM Compiler Infrastructure

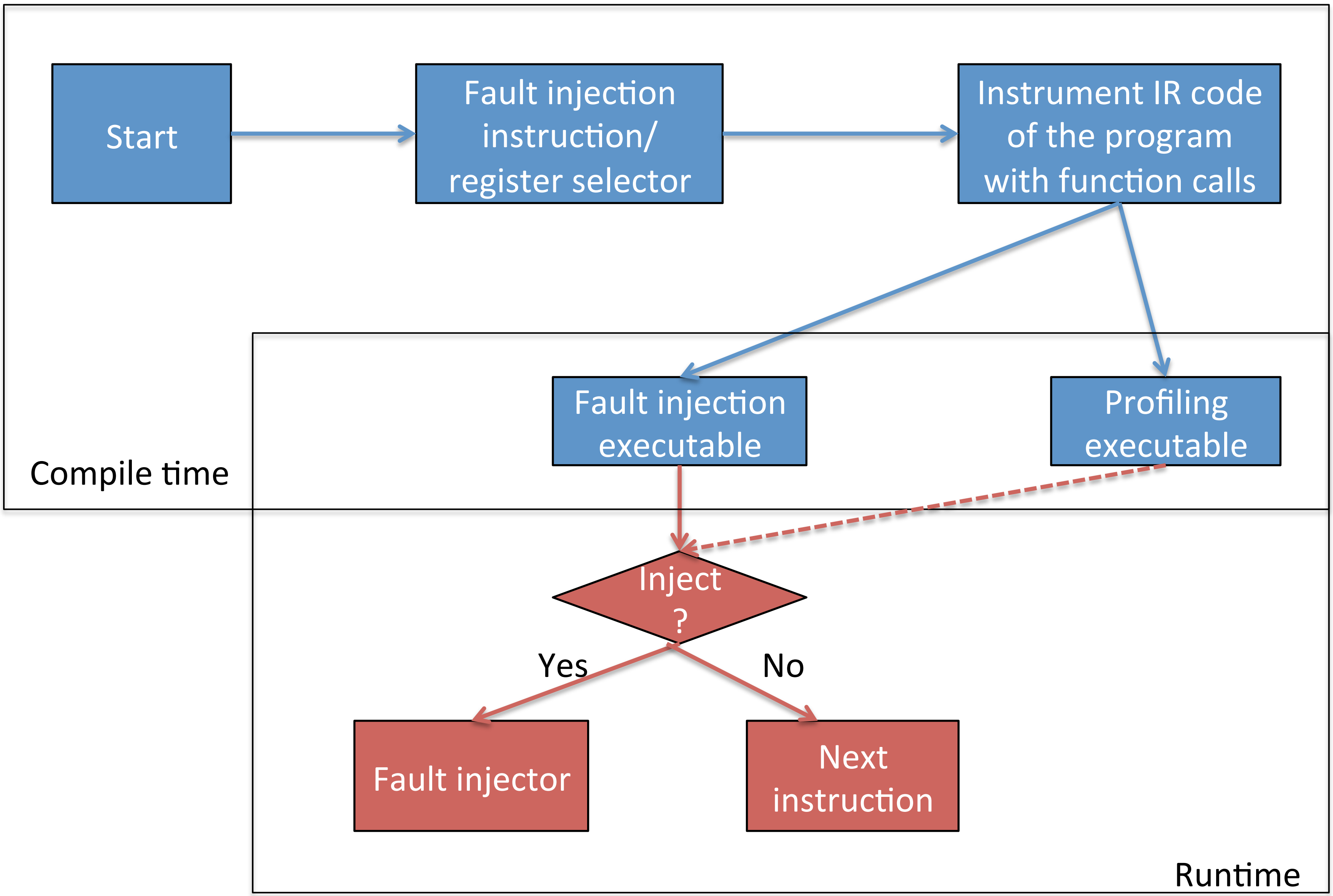


Source: “The architecture of Open-Source Applications”, http://www.aosabook.org/en/llvm.html

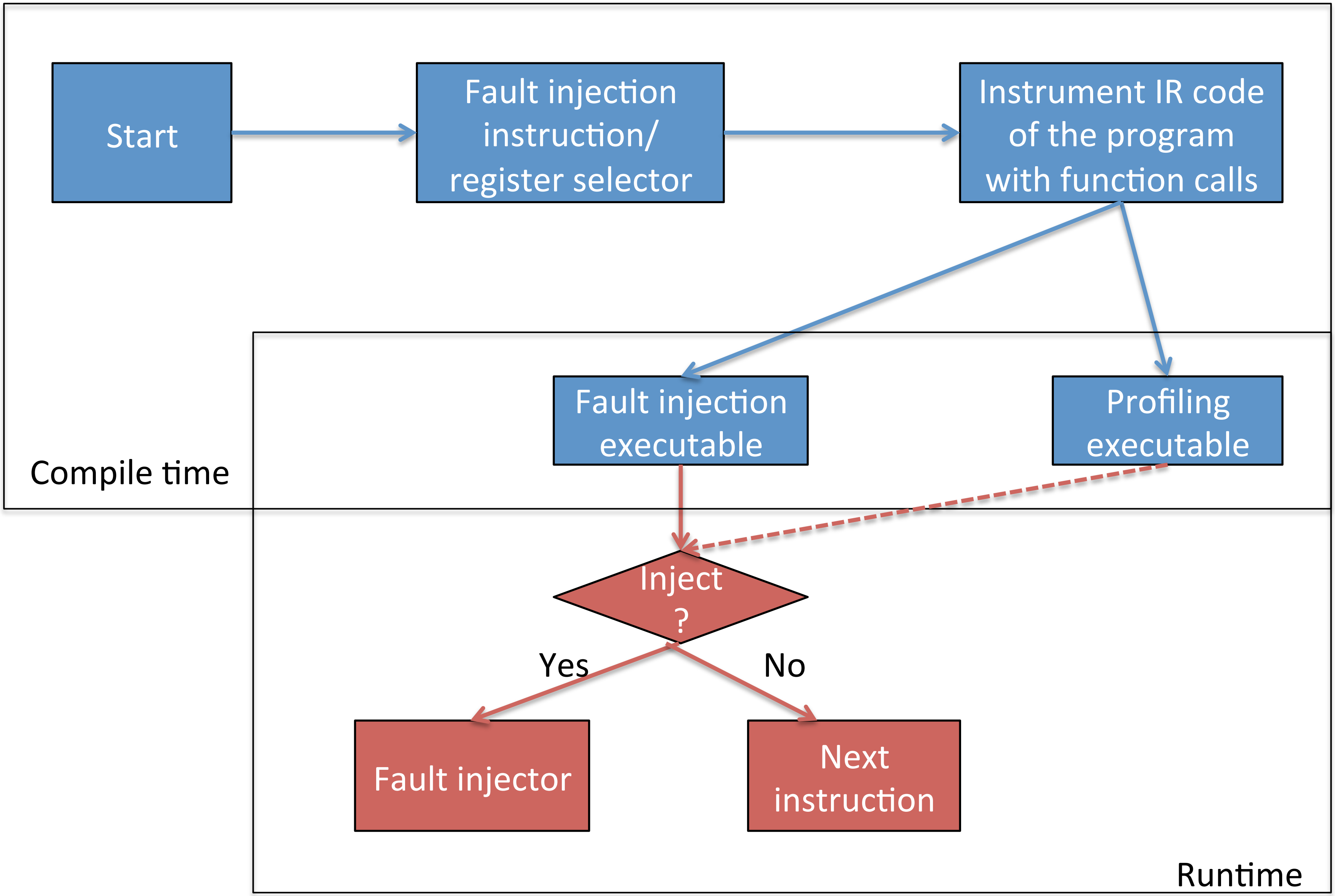
LLFI Architecture

- **Integrated with LLVM Pass Manager**
  - A pass is a scan of the program's LLVM IR code that performs an analysis or transformation
  - LLFI is a series of LLVM passes to identify and instrument selected instructions and registers
- **Runtime libraries are simple and portable**
- **Unified Yaml config file for configuring both**

How does LLFI work?



How does LLFI work?



Fault Injection Passes

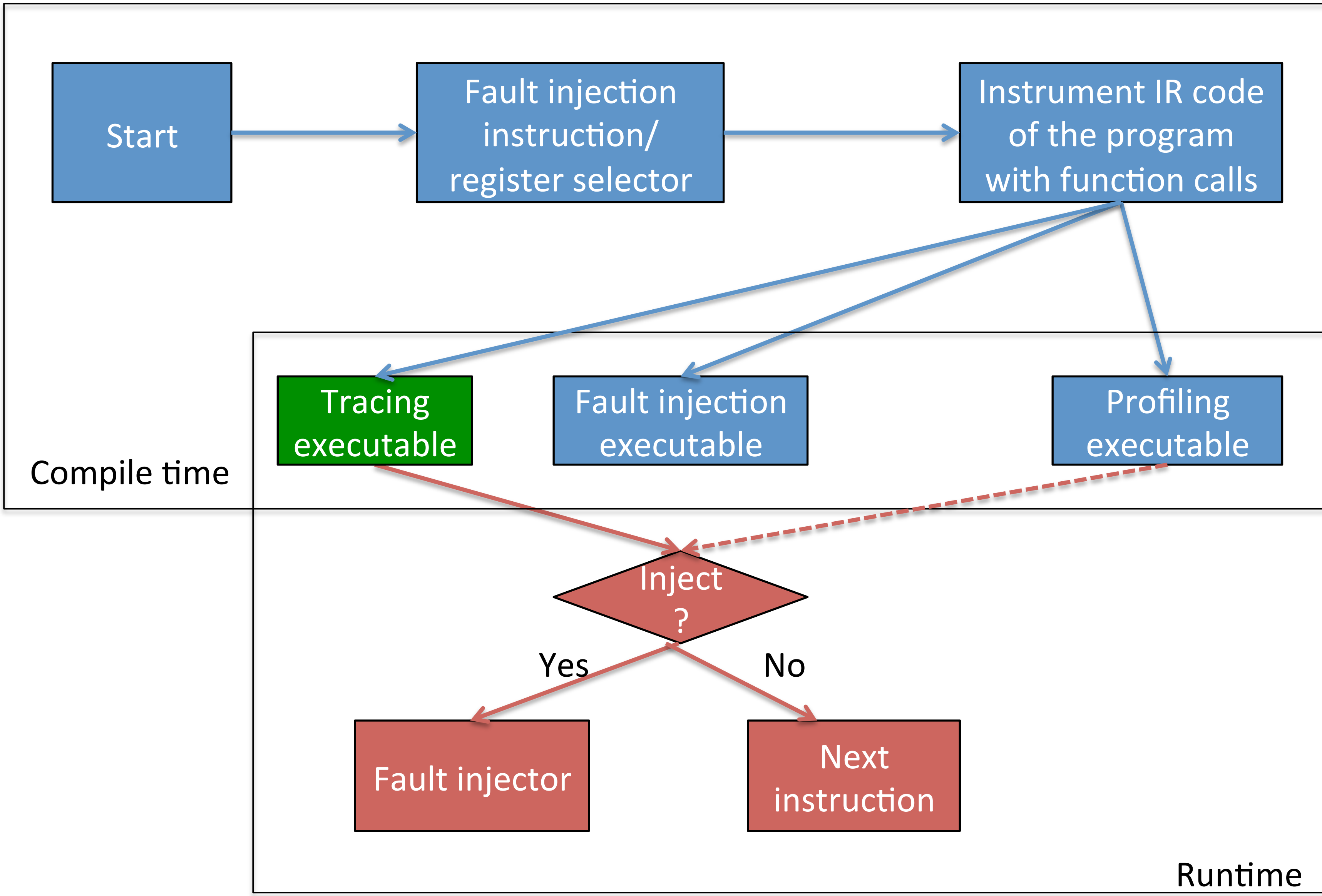
- LLFI allows developer to write fault injection passes for choosing registers or instructions
  - These are wrappers around LLVM passes
  - Provides many housekeeping functions
  - Automatically performs instrumentation of chosen instructions or registers

- Need to register the FI pass with LLFI manager for external visibility and configuration

Runtime Libraries

- Called at runtime during application execution
- Can inject fault at specific cycles (i.e., instances), or when instruction is first reached
- Allow developers to inject faults based on the runtime state of the application (e.g., memory consumption, number of open file handles)

How does LLFI work?



Tracing Instrumentation

- Similar to fault injection instrumentation
- Typically added after every instruction that can be affected by the fault injected data
- Can be customized if needed – ability to choose registers or instructions to trace

Tracing libraries

- Dump the traced variables to a file for later comparison (with the golden run)

- Each trace location assigned a unique static ID to correlate it with the instrumented code

- Efficient tools to compare the trace data with the golden run and identify the difference

Outline (Part 2)

- LLFI Philosophy and Architecture
- Writing Hardware Fault Injector
- Writing Software Fault Injector
- Applications and conclusions

Two kinds of injectors

- Hardware fault injector
  - Can inject faults into all parts of the code/data
- Software fault injector
  - Can inject faults into selected code/data locations
  - Typically at API/library function calls
- We will discuss hardware injector first

Hardware Fault Injector

- Inherit from class *HardwareFIInstSelector*

- Implement the function

  bool isInstFITarget(Instruction *inst)

  Specify the criteria for choosing instructions

  Cannot depend on the order of instructions (use a different method *getInitFIInsts* if it does)

Example: Hardware fault injector

- Choose instructions with a specific opcode
  - Specified in a list *opcodelist* (we'll get to how later)

```
namespace llfi {
bool InstTypeFIInstSelector::isInstFITarget(Instruction *inst) {
  unsigned opcode = inst->getOpcode();
  if (opcodelist->find(opcode) != opcodelist->end()) {
    return true;
  }
  return false;
}
```

Injector's Name

Function to implement

Current instruction

More complex Example: Hardware Fault Injector

- Inject into a specific function(s)

```
bool FuncNameFIInstSelector::isInstFITarget(Instruction *inst) {
  std::string func = inst->getParent()->getParent()->getName();
  func = demangleFuncName(func);

  if (funclist->find(func) != funclist->end()) {
    return true;
  }
  return false;
}
```

Basic Block enclosing instruction

Function enclosing the basic block

Function Name

Housekeeping Items

- Every fault injector needs to be registered with the LLFI pass manager

- Override the following function:

  void getCompileTimeInfo(std::map<std::string, std::string>& info)

  Call: RegisterFIInstSelector()

getCompileTimeInfo function

Allows fault injector to provide info about itself, and the parameters it takes

failure_class = HardwareFault/API/Data/< custom class >/ ...

failure_mode = SpecifiedFunctions/BufferOverflow/DataCorruption/< custom mode >/...

targets = < function names >/< instruction names >/< custom targets >/...

injector = < fi_type >/ChangeValueInjector/ < custom injector >/...

Putting it together: Hardware injector

```
class InstTypeFIInstSelector: public HardwareFIInstSelector {
 public:
  InstTypeFIInstSelector(std::set<unsigned> *opcodelist) {
    this->opcodelist = opcodelist;
  }
  ~InstTypeFIInstSelector() {
    delete opcodelist;
  }
  virtual void getCompileTimeInfo(std::map<std::string, std::string>& info){
    info["failure_class"] = "HardwareFault";
    info["failure_mode"] = "SpecifiedInstructionTypes";
    info["targets"] = "<include list in yaml>";
    info["injector"] = "<fi_type>";
  }

 private:
  virtual bool isInstFITarget(Instruction* inst);
 private:
  std::set<unsigned> *opcodelist;
};
```

Runtime Library

- Implement *injectFault* function to inject fault
  - *llfi_index*: static index of instruction to inject into
  - *size*: Size of the data type being injected in words
  - *fi_bit*: Bit to be flipped (if bit-flip injector)
  - *buf*: Buffer to write the resulting wrong value
  - Example: Bit-flip fault injector

```
virtual void injectFault(long llfi_index, unsigned size, unsigned fi_bit,
                    char *buf) {
  unsigned fi_bytepos = fi_bit / 8;
  unsigned fi_bitpos = fi_bit % 8;
  buf[fi_bytepos] ^= 0x1 << fi_bitpos;
}
```

Installing and Using the new Injector

- Create a new Makefile for the injector
- Compile it using make (see Wiki for details)
- Invoke it from the command line as follows:

cust‍ominstselector -fiinstselectorname <name>

Outline (Part 2)

- LLFI Philosophy and Architecture
- Writing Hardware Fault Injector
- Writing Software Fault Injector
- Applications and conclusions

Software Fault Injectors

- Main difference with hardware injectors
  - Injection only at specific program points, primarily function calls to external libraries or API calls

- Two ways of writing
  - C++ code (similar to Hardware Fault injectors)
  - Using FIDL (Fault Injection Description Language)
  - We will consider FIDL in the next few slides

FIDL Features

- Easy specification of
  - Where to inject
  - What to inject
  - When to inject
- Uses Aspect Oriented Programming (AOP) to separate fault injection logic from other code
  - Automatically weaves FI code into the LLFI code

FIDL Script Specification

Failure_Class:
Failure_Mode:
New_Failure_Mode:
{
Trigger:
// instructions (based on tester's primary metrics) are selected.//

Trigger*:
// instructions (based on tester's secondary metrics) are selected.//

Target:
// registers are selected.//

Action:
// fault type is described//
}

Fault trigger module

Fault injector module

FIDL Syntax: Trigger

- ***Trigger:*** defines IR instructions of interest based on tester's primary metrics

  - Trigger: call [<function name>]
    - E.g1
      - Trigger: call [open, fopen]
    - open/fopen function calls in program trigger FIDL code execution.

FIDL Syntax: Trigger*

- ***Trigger****: defines IR instructions of interest based on tester's secondary metrics to narrow down the injection space to specific locations
  - Trigger*: [<instruction LLFI-indices>]
    - E.g1
      - Trigger*: [43, 58, 60,108, 219, 254]

FIDL Syntax: Target

- ***Target***: defines the desired IR register(s) .
- Target: <function-name> : [args]
  - Args is one of the following options:
    - src [0]/src[1]/src[2]/…src[n]
    - dst
    - RetVal
  - Examples:
    - Eg1, Target : malloc :: src [0]
    - E.g, Target : malloc :: dst
    - E.g, Target : memcpy :: src [1]
      memmove :: src [2]

FIDL Syntax: Actions

Fault Injector Category	Definition	Example Failure Mode
Corrupt	Changes the Data/Address to wrong Values through bit flipping	Wrong Source
Freeze	Creates an artificial loop	No Output
Delay	Creates an artificial delay	CPU Hog
SetValue	Set target to a specific value	No Open
Perturb	Inserts a new erroneous behavior through defining a specific function	Memory leak

FIDL Script: Example

- Injects a fault into the fopen/open calls to the file handle (emulates file not found errors)

```
Failure_Class: Class1
Failure_Mode: FMode1

New_Failure_Mode:
    Trigger:
        call: [fopen, open]
    Target: dst
    Action: Corrupt
```

FIDL Script: More Complex Example

```
Failure_Class: Class2
Failure_Mode: FMode2

New_Failure_Mode:
    Trigger:
        call: [fread, fwrite]
    Trigger*: [1, 50, 55, 60]
    Target:
        src:
            fread: [2]
            fwrite: [0]
    Action:
        Perturb: Custom_Injector

Custom_Injector: |
    int *Target = (int *) buf;
    *Target = *Target + 1000;
```

Creating C++ code from FIDL script

```
FIDL-Algorithm.py takes a FIDL (Fault Injection Description Language) yaml and
generates an instruction/register selector C++ code, and a fault injection
run-time C++ code.

Usage: FIDL-Algorithm.py [OPTIONS]

List of options:
-a <FIDL yaml> : add a FI run-time and selector from a FIDL yaml
-r <name/type> : removes the specified injector by '<FMode>(<FClass>)'
                 or remove all 'custom' or 'default' injector
-l <type>      : lists all active injectors/selectors by 'custom' or 'default'
-h             : shows help

Every time the content of a FIDL yaml is changed, this script should be executed
(-a <FIDL yaml>) to reflect the change(s) in the generated C++ code.

Failure Class and Failure Mode pair should be unique, otherwise the previous
Failure Class and Failure Mode pair is overwritten.
```

Outline (Part 2)

- LLFI Philosophy and Architecture
- Writing Hardware Fault Injector
- Writing Software Fault Injector
- Applications and conclusions

Papers describing LLFI

- SELSE'13
  - Using LLFI for soft-computing applications
- DSN'14
  - Evaluating accuracy of LLFI for hardware faults
- QRS'15
  - Understanding the effect of different FI parameters
- ISSRE'14 Industry Track
  - Design of LLFI for software fault injection
- Safecomp'16
  - Design and evaluation of the FIDL language

Work in my group using LLFI

- DSN'13/TECS-1: Identifying approx. computing regions
- CASES'14/TECS-2: Identifying SDC-prone code regions
- FTXS'14: Extension of LLFI for OpenMP applications
- DSN'15: Identifying long-latency crash causing regions
- ISSRE'15: Checkpoint corruption minimization
- DSN'16: Estimating the overall SDC rate of a program
- SC'16 (LLFI-GPU): LLFI for GPU error propagation
- ICST'17: Error propagation analysis using LLFI
- DSN'17: Effect of multiple bit Vs. single bit faults (Tuesday afternoon 1:45 PM – "Hardware" session 2B)

Other papers/groups using LLFI

- **Ashraf et al., SC'15**
  - Error propagation in MPI applications
- **Fikah et al., DEPEND'15**
  - Effect of double bit flip faults
- **Fault Prophet, MIT, 2015**
  - Forecasting the effect of faults on programs
- **Ni et al., SC'16**
  - Protecting SDC-prone regions in parallel programs

Conclusions

- **LLFI is modular and consists of 2 components**
  - Instruction/register selector which are LLVM passes and run at compile time to instrument the code
  - Runtime libraries to inject faults during execution
- **LLFI is easily extensible for building your own hardware and/or software fault injectors**
  - C++ API built on top of LLVM architecture
  - FIDL language for software fault injectors

TODOs

- **Download LLFI and install it on your computers**
  – Installation instructions on Github
  – Join llfi-development list if you have questions

- **Complete the signup sheet if you haven't done so**
  – You'll receive a short survey in the email about the tutorial – should take you about 5 mins to complete
  – Will send pointer to slides and other material in email
  – Let us know if we can improve the tutorial in any way