Program Analysis Frameworks

Dr. Jeff Huang
Fall 2014
C/C++ Program Workflow

1. **Programmer Writes**
   - **High-level Language Source Code**

2. **Language Compiler**
   - **Object File**
     - **Other Objects Files (Precompiled)**
     - **LibraryFiles**

3. **Linker**
   - **Executable File**

4. **Loader**
   - **Program In Execution**

5. **Run**

6. **Failure**
   - **Error Messages**
Java Program Workflow
Tools

- Compiler frameworks
  - GCC, Clang, LLVM, Javac
- Static Analysis & Instrumentation
  - Soot, WALA, LLVM
- Dynamic Analysis
  - PIN, Valgrind, ASM
- Symbolic Execution
  - KLEE, JPF, CATG
- Model checking
  - JPF
- Language:
  - Java: Soot, WALA, JPF, ASM, CATG
  - C/C++: LLVM, KLEE, PIN, Valgrind
  - Android: Soot
  - JavaScript: WALA
Soot

• Control flow graph, dataflow analysis, pointer analysis, call graph
• ...
• Eclipse plugin

• Extending Soot with your own analysis
  – Android taint analysis
  – Null pointer analysis, array bound analysis
  – Data race detection
  – Slicing, Profiling
  – ...

Jimple

• A typed 3-address intermediate representation

```java
public static void main(String[] args)
{
    System.out.println("Hello World");
}
```

```java
public static void main(java.lang.String[])
{
    java.lang.String[] args;
    java.io.PrintStream temp$0;

    args := @parameter0: java.lang.String[];
    temp$0 = <java.lang.System: java.io.PrintStream out>);
    virtualinvoke temp$0.<java.io.PrintStream: void println(java.lang.String)>("Hello World");
    return;
}
```
Extending Soot

• Implement a BodyTransformer or a SceneTransformer
  – internalTransform method does the transformation

• Choose a pack for your transformation
  – Intraprocedural: jtp
  – Interprocedural: wjtp

• Write a main method that adds the transform to the pack, then runs Soot’s main
Soot

• Soot main page:

• Soot download page:

• Soot tutorials:
WALA

• Provides static analysis libraries
  – Pointer analysis, call graph, interprocedural data-flow analysis, context-sensitive slicing

• Limited code transformation
  – WALA IR (SSA-based) is immutable
  – Support bytecode instrumentation via Shrike
  – Recommendation: use WALA for computing analysis results, do transformation separately

• For JavaScript analysis
WALA

Build CallGraph for HTML

URL url = ...;
// use Rhino to parse JavaScript
JSCallGraphUtil.setTranslatorFactory(
    new CAstRhinoTranslatorFactory());
// build the call graph
CallGraph cg = JSCallGraphBuilderUtil.makeHTMLCG(url);
Java PathFinder (JPF)

• An explicit state model checker
  – Focus is on finding bugs in Java programs
  – The models are Java programs
  – The properties can be assertions, gov.nasa.jpf.Property objects, or JPF listener objects

• A framework for runtime Java verification
  – Model checking
  – Symbolic execution
  – UML state chart modeling
  – Numeric Verification (int overflow, fp over/underflow, ...)
  – ...
Java PathFinder (JPF)

System under Test (Java bytecode)

abstract virtual machine

JPF core

JPF extension

* execution semantics
* program properties

verification artifact

* report
* test case
* specification
...
Java PathFinder (JPF)
Java PathFinder (JPF)

• Pure Java up to ??-KLOC
  – Depends on logical complexity and state size, not KLOC.
  – Programs with 100K+ lines have been analyzed

• Multi-threaded code (*Of Course!*)

• Symbolic execution (Symbolic PathFinder)

• Building properties: deadlocks, race conditions, unhandled exceptions, application-specific assertions, ...

• Custom: implement your own using JPF’s extension
Native Methods

• JPF supports all bytecode instructions
  – Custom implementation: for tracking state changes caused by their execution

• Problem: native methods (JNI)
  – State changes cannot be tracked by JPF
  – Hence, No libraries with unsupported native calls
  – Relevant features: file I/O, GUI, networking, ...

• Remedy options
  – Custom implementation in Java model classes
  – Delegation to host JVM via the Model-Java Interface
Using JPF

• http://babelfish.arc.nasa.gov/trac/jpf/wiki
• Google “A Hands-On Java PathFinder Tutorial”
LLVM

- A compiler framework with collection of reusable components and tools
  - C/C++/Objective-C/C++ ... -> LLVM IR -> (Optimizer) -> LLVM IR -> native code (X86, ARM, PPC, etc.)
LLVM

• LLVM tutorial

• **Clang**: a frontend for C language family
  – “clang -emit-llvm -c hello.cpp -o hello.bc”

• **llvm-gcc/g++**: LLVM C/C++ frontend
  – “llvm-g++ --emit-llvm -c hello.cpp -o hello.bc”

• **llvm-dis**: disassemble a bitcode file into a human-readable .ll file
  – “llvm-dis hello.bc”

• **opt**: analyze and transform LLVM bitcode
  – “opt -O1 -o hello.bc hello2.bc”

• **llc**: code generator from bitcode to architecture-specific assembly code
  – “llc hello2.bc -o hello2.s”
  – “llvm-gcc hello2.s -o hello”
Program Analysis with LLVM

• Write your own pass

```cpp
namespace {
    struct Hello : public FunctionPass {
        static char ID;
        Hello() : FunctionPass(ID) {}
        virtual bool runOnFunction(Function &F) {
            errs() << "Hello: ";
            errs().write_escaped(F.getName()) << '\n';
            return false;
        }
    };
    char Hello::ID = 0;
    static RegisterPass<Hello> X("hello", "Hello World Pass");
}
```

– “opt –load yourpass yourcommand hello.bc –o hello2.bc”
KLEE

- A symbolic execution tool built upon LLVM (2.9)
- Find memory errors and division by zero defects
- Generate test cases to trigger the error
KLEE

• Mark your variable symbolic

```c
int main() {
    int a;
    klee_make_symbolic(&a, sizeof(a), "a");
    return get_sign(a);
}
```

• Run “klee hello.bc”

• Hack KLEE to implement your own symbolic analysis!
  – klee.llvm.org
PIN

• Dynamic binary instrumentation
  – Instruments executables directly
  – No need to recompile or relink
  – Handle dynamically-generated code
  – Attach to running processes

• Actively developed at Intel
• Large user base in academia and industry
Using PIN

Launch and instrument an application

$ pin -t pintool -- application

Instrumentation engine (provided in the kit)

Instrumentation tool (write your own, or use one provided in the kit)

Attach to and instrument an application

$ pin -t pintool -pid 1234

• Official user’s manual

• PIN tutorial
Valgrind

• Another popular dynamic binary instrumentation framework
  – http://valgrind.org/

• Open source

• Easy to execute, e.g.:
  – valgrind --tool=memcheck ls

• Overhead is the problem
  – 5-10X slowdown without any instrumentation
Valgrind Tools

http://www.valgrind.org/info/tools.html

- MemCheck – check memory errors
- Cachegrind – profile cache misses
- Callgrind – profile callgraphs
- Massif – profile heap (usage overtime)
- Helgrind/DRD – detect data races
Writing a New Valgrind Tool


• Use a template
  – The tool lackey is good candidate
  – Two parts to fill in
    • Instrumenter
    • Runtime

• Instrumenter
  – Initialization
  – Instrumentation
  – Finalization
  – System calls interception

• Runtime
  – Transfer functions
  – Memory management for abstract state
ASM

• Dynamic instrumentation for Java
  – Modify existing classes or dynamically generate classes
• “All purpose bytecode manipulation and analysis framework”
• http://asm.ow2.org

• Basic process
  – Construct ClassFileTransformer, ClassWriter
  – Stack up the visitors for classes, methods, fields, annotations, etc.
  – Add “–javaagent” to hook into class loading process