Static Analysis to Improve Software Reliability and Security

Jeff Foster
University of Maryland, College Park

September 15, 2008
Technology Runs on Software

• Software is everywhere
  ▪ A fundamental breakthrough of computer science

• It’s hard to build software that works correctly
  ▪ We’ve been making good progress for ~50 years
  ▪ But, there’s still a long way to go

• Unreliable software is costly
  “[T]he national annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion.”

  —NIST 2002 planning report
Current Practice

• Testing
  - Make sure program runs correctly on set of inputs

  ![Diagram showing inputs going through a program to produce outputs, which are then compared to an oracle to determine correctness.]

  - Drawbacks: Expensive, difficult, hard to cover all code paths, no guarantees
Current Practice (cont’d)

• Code Auditing
  ▪ Convince someone else your source code is correct
  ▪ Drawbacks: Expensive, hard, no guarantees
And If You’re Worried about Security...

A malicious adversary is trying to exploit anything you miss!

What more can we do?
Tools for Software Quality

• My research focus: **Static analysis**
  - Tools that analyze program source code

• Two compelling advantages
  - Can reason about *all possible* runs of the program
  - Will never misread or misinterpret source code

• Disadvantage: Static analysis is hard
  - Theoretically undecidable
  - Doable in practice, but requires
    - Deep understanding of how languages work
    - Experimentation to validate ideas
Recent Projects

- **Saffire** — Type check multi-lingual programs
- **Locksmith** — Data race detection for C
- **Pistachio** — Checking network protocol implementations
- **CMod** — Enforcing modularity in C
- **Uno** — Ownership and uniqueness inference for Java
- **CQual** — User-defined type qualifiers for C
- **JQual** — User-defined type qualifiers for Java

http://www.cs.umd.edu/projects/PL
Papers, software, documentation, etc
Saffire

Type Checking Foreign Function Interfaces

[Furr and Foster, PLDI 2005, ESOP 2006, TOPLAS 2008]
Introduction

• Many high-level languages contain a foreign function interface (FFI)
  ▪ Java, OCaml, SML, Haskell, COM, SOM, ...
  ▪ Allows access to functions written in other languages

• Lots of reasons to use them
  ▪ Gives access to system calls
  ▪ Access legacy libraries that may be infeasible to port
  ▪ Performance
  ▪ Suitability of language for particular problem
Dangers of FFIs

- In most FFIs, programmers write “glue code”
  - Translates data between host and foreign languages
  - Typically written in one of the languages

- FFIs are often easy to misuse
  - Little or no checking done at language boundary
  - Mistakes can silently corrupt memory
  - One solution: interface generators
    - But there’s still lots of hand-written code around
Example JNI Code

• Java:

```java
Class Foo {
    int x;
    private native void bar(Foo obj);
}
```

• C:

```c
void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls,"x","I");
    int y = GetIntField(obj,fid);
    ...
}
```
Example JNI Code

- Java:

```java
class Foo {
    int x;
    private native void bar(Foo obj);
}
```

- C:

```c
void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls,"x","I");
    int y = GetIntField(obj,fid);
    ...
}
```
Example JNI Code

• Java:

        Class Foo {
            int x;
            private native void bar(Foo obj);
        }

• C:

        void Java_Foo_bar(JNIEnv* obj) {
            jobject cls = GetObjectClass(obj);
            jfieldID fid = GetFieldID(cls, "x", "I");
            int y = GetIntField(obj, fid);
            ...
        }

I = Int

obj.x
Example JNI Code

• Java:

```java
Class Foo {
    int x;
    private native void bar(Foo obj);
}
```

• C:

```c
void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls,"x","I");
    int y = GetIntField(obj,fid);
    y = obj.x;
    ... 
}
```
Example JNI Code

• Java:

```java
Class Foo {
    int x;
    private native void bar(Foo obj);
}
```

• C:

```c
void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, "x", "I");
    int y = GetIntField(obj, fid);
    ...
}
```
Example JNI Code

• Java:

```java
Class Foo {
    int x;
    private native void bar(Foo obj);
}
```

• C:

```c
void Java_Foo_bar(JNIEnv *obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, "x", "I");
    int y = GetIntField(obj, fid);
    ...}
```
Example JNI Code

- **Java:**

```java
class Foo {
    int x;
    private native void bar(Foo obj);
}
```

- **C:**

```c
void Java_Foo_bar(JNIEnv obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls,"x","I");
    int y =GetIntField(obj,fid);
    ...
}
```

**Types must match!**
**Saffire**

- **Static Analysis of Foreign Function Interfaces**
  - Static type inference across OCaml, Java FFIs
- Analyzes C “glue code”
  - Layers a richer type system on top of the C types
  - Checks C code accesses high-level data at right type
- Key design point: Only as complex as necessary
  - FFI glue code is messy
    - ...but not all that complicated (to avoid mistakes!)
  - Can use fairly simple analysis in surprising places
Keys Ideas behind Saffire

• FFIs conflate high-level data into a single type
  - JNI: C type `jobject` represents all Java objects
  - OCaml FFI: C type `value` represents all OCaml data

• Refine this type to carry around extra type info
  - `ctype ::= void | int | jtype jobject | ...`
  - `jtype ::= { class name; fields; methods } | ...`
  - `ctype ::= void | int | mtype value | ...`
  - `mtype ::= ...`
Keys Ideas behind Saffire (cont’d)

• C code has a “subatomic” view of high-level ops
  ▪ JNI: Single field access takes three steps
    - Get class object, get field ID, access field
  ▪ OCaml FFI: “Pattern matching” done with if/then/elses
    - Check “boxedness” of data, check constructor, get value

• Two consequences
  ▪ Must track precise values of intermediate values like jfieldID, jmethodID
  ▪ jtype and mtype must be able to represent partial info
Key Ideas (cont’d)

• Glue code types depend on *data values*
  ▪ JNI: Class, field, method names represented by C strings
  ▪ OCaml FFI: Tests and accesses depend on ints and ptrs
  ▪ Data values can be passed around, stored, returned, etc.

• Use *singleton types* to track data
  ▪ Ex: "foo" : str{"foo"}  // str == char *
  ▪ Ex: void bar(char *x);  x : str{ν}
    - The variable ν stands for an unknown string
JNI Challenge: Wrapper Functions

• Accepts any object \texttt{obj} with int field \texttt{field} 
  ▪ Polymorphic in type of obj and \textit{contents} of field 

```c
int my_getIntField(jobject obj, char *field) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, field, "I");
    return GetIntField(obj,fid);
}
```

  - String types are singletons, hence contents = type

• Wrappers come up often in practice
  - And JNI has >200 functions! Need to treat polymorphically
Example

```c
int my_getIntField(jobject obj, char *field) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, field, "I");
    return GetIntField(obj, fid);
}
```

\[ \forall \nu . \{\ldots; \nu : \text{int}, \ldots; \ldots\} \text{ jobject } \times \text{ str}\{\nu\} \rightarrow \text{ int} \]

- Second arg is some string \( \nu \)
- First arg is some object with an \text{ int} field of name \( \nu \)
- The function returns an \text{ int}
Soundness

• Theorem: If a program is well-typed, then glue code never uses OCaml or Java data at the wrong type
  ▪ Proof for slightly restricted versions of each system
  ▪ Uses standard techniques
Implementation

Diagram:
- **Input Program**
  - OCaml/Java Source
  - C Source
- **Type Extractor**
- **Type Repository**
- **C Analysis Engine**
- **Potential Bugs**
### JSaffire Results

<table>
<thead>
<tr>
<th>Program</th>
<th>C LoC</th>
<th>Java LoC</th>
<th>Ext</th>
<th>Time (s)</th>
<th>Err</th>
<th>Wrn</th>
<th>FPos</th>
<th>Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>libgconf-java-2.10.1</td>
<td>1,119</td>
<td>670</td>
<td>93</td>
<td>1.32s</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>libglade-java-2.10.1</td>
<td>149</td>
<td>1,022</td>
<td>6</td>
<td>0.64s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libgnome-java-2.10.1</td>
<td>5,606</td>
<td>5,135</td>
<td>599</td>
<td>6.53s</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libgtk-java-2.6.2</td>
<td>27,095</td>
<td>32,395</td>
<td>3,201</td>
<td>1.04s</td>
<td>74</td>
<td>8</td>
<td>36</td>
<td>18</td>
</tr>
<tr>
<td>libgtkhtml-java-2.6.0</td>
<td>455</td>
<td>729</td>
<td>72</td>
<td>0.65s</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libgtkmozembed-java-1.7.0</td>
<td>166</td>
<td>498</td>
<td>23</td>
<td>0.66s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libvte-java-0.11.11</td>
<td>437</td>
<td>184</td>
<td>36</td>
<td>0.67s</td>
<td>0</td>
<td>26</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jnetfilter</td>
<td>1,113</td>
<td>1,599</td>
<td>105</td>
<td>5.38s</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libreadline-java-0.8.0</td>
<td>1,459</td>
<td>324</td>
<td>17</td>
<td>0.63s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>pgpjava</td>
<td>10,136</td>
<td>123</td>
<td>12</td>
<td>1.11s</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>posix1.0</td>
<td>978</td>
<td>293</td>
<td>26</td>
<td>0.70s</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Java Mustang compiler</td>
<td>532k</td>
<td>1974k</td>
<td>2495</td>
<td>630s</td>
<td>1</td>
<td>88</td>
<td>96</td>
<td>2620</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>156</td>
<td>124</td>
<td>142</td>
<td>2642</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
JSaffire Errors

• 68 functions declared with the wrong arity

• 56 C pointers passed where object expected
  ▪ Most from a software rewrite

• 18 type mismatches
  ▪ e.g., \texttt{String} \neq \texttt{byte[]} 
  ▪ Mustang error: \texttt{int} vs. \texttt{long}

• 14 functions named incorrectly
  ▪ Functions must follow a strict convention to be called from Java
JSaffire Warnings

- 1 malformed Java class string
- 13 incorrect type declarations
  - JNI contains several typedef’s for jobject (e.g., jstring, jintarray)
  - Warn when C function was declared with the wrong type, even when the value was of the right type
- 110 dead C functions
  - C function appeared to implement a certain Java native method, but no native method was defined in the Java class file
JSaffire False Pos. and Imprecision

• 140 false positives
  ▪ Most from C code using subtyping on Java types
    - JSaffire based on unification, so considered these type errors

• 2642 imprecision messages
  ▪ Vast majority (2620) in Mustang
    - Breaks some of our assumptions, plus heavy use of fn ptrs
  ▪ 16 due to unresolved overloading
    - JSaffire didn’t have enough info to find a consistent type
  ▪ 6 passing arguments to JNI functions packed in an array
Conclusion

- FFIs are a useful part of a language

- FFI code is messy
  - But not complicated, hence analyzable

- Saffire: Type checking multi-lingual code
  - The first system that does type checking across FFIs
  - Results suggest Saffire is an efficient, effective tool
Locksmith

Data Race Detection for C

Data Races

• Two threads “simultaneously” access a location
  ▪ At least one access is a write

• Data races can cause major problems
  ▪ 2003 Northeastern US blackout
    - Partially due to data race in a C++ program
  ▪ Threator-25 medical accelerator
    - Data race caused some patients to receive lethal radiation

• Races complicate program analysis and understanding
  ▪ Meaning of programs with races often undefined
Programming Against Data Races

• \( x \sim l \) ("\( x \) is correlated with lock \( l \))
  - Means that \( l \) is held during some access to \( x \)
    - Ex: `lock(&l); ...; x = 3; ...; unlock(&l);`

• \( x \) is guarded by lock \( l \) if \( x \) is always correlated with \( l \)
  - I.e., \( l \) is always held when \( x \) is accessed

• If all shared variables are guarded by some lock, then the program is race-free
  - Not the only way to prevent data races
  - But widely used and easy to understand
Locksmith

• Enforce race freedom in C programs
  ▪ Input is a “bare” C program (no special annotations)
  ▪ Ensure every location is guarded by some lock

• Aims to be sound
  ▪ If Locksmith reports no races,
  ▪ then the program is race free
Locksmith: Race Detection for C

void foo(pthread_mutex_t *l, int *p) {
    pthread_mutex_lock(l);
    *p++;
    pthread_mutex_unlock(l);
}
...

pthread_mutex_t L1, L2;
int x, y;
foo1(&L1, &x);
foo2(&L2, &y);

Analysis is context sensitive: Distinguishes different calls to same function

Static analysis representation:
Graph representing “flow” and correlation
*
*p accessed with *l held
Infer x accessed with L1 held
Infer y accessed with L2 held

Actuals “flow” to formals at call #1
Actuals “flow” to formals at call #2
Challenges

• Need to determine *shared locations*
  - Don’t need to hold locks for thread-local data
  - Data can switch from thread-local to shared

• *Flow-sensitivity* for locks
  - Need to compute what locks held at each point
  - Can acquire and release a lock at any time

• Need to model C language features and idioms
  - Structs, void *’s, type casts, locks in data structures
## Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size (LOC)</th>
<th>Time (sec)</th>
<th>Warnings</th>
<th>Unguarded</th>
<th>Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Apps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aget</td>
<td>1,914</td>
<td>0.85</td>
<td>62</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>ctrace</td>
<td>2,212</td>
<td>0.59</td>
<td>10</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>engine</td>
<td>2,608</td>
<td>0.88</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>knot</td>
<td>1,985</td>
<td>0.78</td>
<td>12</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>pfscan</td>
<td>1,948</td>
<td>0.46</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>smtprc</td>
<td>8,624</td>
<td>5.37</td>
<td>46</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Linux drivers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3c501</td>
<td>17,443</td>
<td>9.18</td>
<td>15</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>eql</td>
<td>16,568</td>
<td>21.38</td>
<td>35</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hp100</td>
<td>20,370</td>
<td>143.23</td>
<td>14</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>plip</td>
<td>19,141</td>
<td>19.14</td>
<td>42</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>sis900</td>
<td>20,428</td>
<td>71.03</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>slip</td>
<td>22,693</td>
<td>16.99</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sundance</td>
<td>19,951</td>
<td>106.79</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>synclink</td>
<td>24,691</td>
<td>1521.07</td>
<td>139</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>wavelan</td>
<td>20,099</td>
<td>19.70</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

dual core Xeon with 4GB memory
Summary

• Locksmith able to find data races
  ▪ Many can cause the program to misbehave
  ▪ Relatively low false positive rate

• Measured tradeoffs of many different design pts
  ▪ E.g., different sharing analyses, struct modeling, etc

• Proven key parts of Locksmith sound

• One of the most scalable, precise static data race detection systems for C
Pistachio

Checking Implementations of Network Protocols

[Udrea, Lumezanu, Foster, USENIX Security 2006, Inf. & Comp. 2008]
Motivation

• Network protocols must be reliable and secure

• Much work on proving *abstract* protocols correct
  ▪ But implementations can introduce vulnerabilities

• Goal: Check that implementations match specifications
  ▪ Ensure that the protocol we’ve modeled abstractly and thought hard about is actually what’s in the code
Pistachio Architecture

RFC/IETF Standard

C Source Code

Bug Database

Rule-Based Specification

Pistachio

Evaluate Warnings

Theorem Prover

Errors Detected

Existing Documents
Pistachio’s Approach

- Simulate program with **symbolic execution**
  - \( \{x = 5, y = 6\} x = x + y; \{x = 11, y = 6\} \)

- Two techniques to improve scalability
  - Intersect sets of facts at *join points*
    - Reduces amount of data to track
  - Try to find a *fixpoint* of facts for loops
    - Worked >80% of the time in our experiments
    - Else, give up and move on after some number of iterations

- Result: Pistachio is really fast
  - Analysis takes less than a minute generally
  - Does introduce imprecision
Experimental Framework

- Used Pistachio on two protocols:
  - LSH implementation of SSH2 (0.1.3–2.0.1)
    - 87 rules initially, + 9 more to target specific bugs
  - OpenSSH (1.0p1–2.0.1)
    - Same specification as above
  - RCP implementation in Cygwin (0.5.4–1.3.2)
    - 51 rules initially, + 7 more to target specific bugs

- Rule development time – approx. 7 hours
Example SSH2 Rule

“It is STRONGLY RECOMMENDED that the ‘none’ authentication method not be supported.”

recv(_, in, _)  
in[0] = SSH_MSG_USERAUTH_REQUEST  
isOpen[in[1..4]] = 1  
in[21..25] = “none”  
=>
send(_, out, _)  
out[0] = SSH_MSG_USERAUTH_FAILURE

If we see a call to recv()  
And it’s a user auth request  
Where the auth method is none  
Then call send() with a failure message
Example Bug

```c
fmsgrecv(clisock, SSH2_MSG_SIZE);
if(!parse_message(MSGTYPE_USERAUTHREQ, inmsg, len(inmsg), &authreq))
    return;
...
if(authreq.method == USERAUTH_PKI) {
    ...}
} else if (authreq.method == USERAUTH_PASSWD) {
    ...}
} else {  // Oops! Forgot to check for 'none'
    ...
}

sz = pack_message(MSGTYPE_REQSUCCESS, payload, outmsg, SSH2_MSG_SIZE);
fmsgsend(clisock, outmsg, sz);
```

Message received
Check for user auth request
Handle PKI method
Handle password method
Oops! Forgot to check for 'none'
Send success
Summary

• Rule-based specifications easy to create from standards documents
  ■ And more precise, too
• Evaluated on LSH, OpenSSH, RCP
• Found wide variety of known bugs and vulnerabilities
  ■ Well over 100 bugs, of many different kinds
  ■ Roughly 5% false negatives, 38% false positives
  ■ Analysis runs very quickly
• Suggests Pistachio is a valuable tool
Current and Future Directions
Promise of Static Analysis

• We’ve made huge strides in the field of static analysis over the last 10-15 years
  ▪ Scalability
    - Algorithms that run on large programs
  ▪ Applicability
    - Ability to check wide range of interesting program properties
  ▪ Formal foundations
    - Can check if what we’re doing is internally consistent

• The result: a well-equipped toolbox
  ▪ Companies today use static analysis, sell static analysis
Dark Underbelly of Static Analysis

• Still hard to develop a new static analysis
  ▪ Progress measured in PhD dissertation/paper units

• Static analysis tools are brittle
  ▪ If they don’t do what you want, you’re out of luck

• Tool designers choose analysis abstractions
  ▪ Won’t work if analysis reasoning != programmer reasoning

We need more flexible, more adaptable, more user-centered techniques
Path Projection

- A new visualization for *program paths*
  - Common output in static analysis error reports

- Based on InfoViz principles.
  - Core idea: Fit as much context on screen as possible

- Performing user studies to measure effectiveness
  - Improvement in *triaging* Locksmith error reports

- Looking into other applications, other viz. ideas

[Khoo, Foster, Hicks, Sazawal, PASTE 2008]
Diamondback Ruby (DRuby)

- Scripting languages are really popular
  - They don’t “get in your way”
  - Good for rapid prototyping

- What happens when little scripts grow up?
  - Missing language features that make code reuse, evolution, and maintenance easier

- DRuby: Variant of Ruby that adds static safety features of more traditional languages to Ruby
  - Current focus on mixing static/dynamic typing
    - Aiming for a “get what you pay for” model of static checking
...and?

- Can we leverage the most common kind of checking people do today, testing?
  - Use symbolic evaluation to turn individual test cases into richer specifications
  - Infer specifications from test cases

- Can we give programmers ways to mix different kinds of static analysis?
  - Use a simple, scalable analysis most places, and a precise but expensive analysis where needed

- Can we make a user-centered static analysis toolbox?
Thanks!