Week: 10 of 12

COMP 2120 / COMP 6120

MORE TESTING, THEN STATIC ANALYSIS

A/Prof Alex Potanin and Dr Melina Vidoni
ANU Acknowledgment of Country

“We acknowledge and celebrate the First Australians on whose traditional lands we meet, and pay our respect to the elders past and present.”

DYNAMIC ANALYSIS AND ADVANCED AUTOMATED TESTING
Puzzle:
Find \( x \) such that \( p1(x) \) returns True

```python
def p1(x):
    if x ** 2 - 10 == 15:
        return True
    return False
```

ANU SCHOOL OF COMPUTING | COMP 2120 / COMP 6120 | WEEK 11 OF 12: STATIC ANALYSIS
Puzzle:
Find $x$ such that $p2(x)$ returns `True`

```python
def p2(x):
    if x > 0 and x < 1000:
        if ((x - 32) * 5/9 == 100):
            return True
    return False
```

return False
**Puzzle:**

Find \( x \) such that \( p3(x) \) returns True

def p3(x):
    if 3 < x < 100:
        z = x - 2
        c = 0
        while z >= 2:
            if z ** (x - 1) % x == 1:
                c = c + 1
                z = z - 1
        if c == x - 3:
            return True
    return False
FindBugs (2006!)

FindBugs™ - Find Bugs in Java Programs

This is the web page for FindBugs, a program which uses static analysis to look for bugs in Java programs. The current version of FindBugs 3.0.1, released on 13:05:33 EST, 06 March, 2015.

FindBugs 3.0.1 Release

- A number of changes described in the [changes document](#), including new bug patterns:
  - BSHIFT_WRONG_ADD_PRIORITY
  - CO_COMPARETO_INCORRECT_FLOATING
  - DC_PARTIALLY_CONSTRUCTED
  - DM_ROXED_PRIMITIVE_FORCOMPARE
  - DM_INVALID_MIN_MAX
  - ME_MUTABLE_ENUM_FIELD
  - ME_ENUM_FIELD_SETTER
  - MS_MUTABLE_COLLECTION
  - MS_MUTABLE_COLLECTION_PKGPROTECT
  - NAME_ADD_PATHEVINY

FindBugs requires JRE (or JDK) 1.7.0 or later to run. However, it can analyze programs compiled with JRE (or JDK) 1.6.

The current version of FindBugs is 3.0.1.
Security and Robustness

FUZZ TESTING
Barton P. Miller, Lars Fredriksen and Bryan So

**Study of the Reliability of UNIX Utilities**

Communications of the ACM (1990)

"On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash."
Fuzz Testing

1990 study found crashes in:
adb, as, bc, cb, col, diction, emacs, eqn, ftp, indent, lex, look, m4, make, nroff, plot, prolog, ptx, refer!, spell, style, tsort, uniq, vgrind, vi
Common Fuzzer-Found Bugs in C/C++

**Causes:** incorrect arg validation, incorrect type casting, executing untrusted code, etc.

**Effects:** buffer-overflow, memory leak, division-by-zero, use-after-free, assertion violation, etc. (“crash”)

**Impact:** security, reliability, performance, correctness

How to identify these bugs in languages like C/C++?
Automatic Oracles: Sanitizers

- Address Sanitizer (ASAN)
- LeakSanitizer (comes with ASAN)
- Thread Sanitizer (TSAN)
- Undefined-behavior Sanitizer (UBSAN)

https://github.com/google/sanitizers
AddressSanitizer

int get_element(int* a, int i) {
    return a[i];
}

int get_element(int* a, int i) {
    if (a == NULL) abort();
    return a[i];
}

int get_element(int* a, int i) {
    if (a == NULL) abort();
    region = get_allocation(a);
    if (in_heap(region)) {
        low, high = get_bounds(region);
        if ((a + i) < low || (a + i) > high) {
            abort();
        }
    }
    return a[i];
}

int get_element(int* a, int i) {
    if (a == NULL) abort();
    region = get_allocation(a);
    if (in_stack(region)) {
        if (popped(region)) abort();
    } else {
        if (in_heap(region)) {
        ...
    }
    return a[i];
}

Is it null?

Is the access out of bounds?

Is this a reference to a stack-allocated variable after return?
AddressSanitizer

Asan is a memory error detector for C/C++. It finds:

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

https://github.com/google/sanitizers/wiki/AddressSanitizer
Strengths and Limitations

• **Exercise**: Write down two strengths and two weaknesses of fuzzing. Bonus: Write down one or more assumptions that fuzzing depends on.
Strengths and Limitations

• **Strengths:**
  
  • Cheap to generate inputs
  
  • Easy to debug when a failure is identified

• **Limitations:**
  
  • Randomly generated inputs don’t make sense most of the time.
    
    • E.g. Imagine testing a browser and providing some ”input” HTML randomly: `dgsad5135o gsd;gj lsdkg3125j@!T%#( W+123sd asf j`
  
  • Unlikely to exercise interesting behavior in the web browser
  
  • Can take a long time to find bugs. Not sure when to stop.
Mutation-Based Fuzzing (e.g. Radamsa)
Mutation Heuristics

- **Binary input**
  - Bit flips, byte flips
  - Change random bytes
  - Insert random byte chunks
  - Delete random byte chunks
  - Set randomly chosen byte chunks to *interesting* values e.g. INT_MAX, INT_MIN, 0, 1, -1, ...
  - Other suggestions?

- **Text input**
  - Insert random symbols or keywords from a dictionary
  - Other suggestions?
American Fuzzy Lop (https://github.com/google/AFL)

2) The afl-fuzz approach

American Fuzzy Lop is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to effortlessly pick up subtle, local-scale changes to program control flow.

Simplifying a bit, the overall algorithm can be summed up as:

1. Load user-supplied initial test cases into the queue,
2. Take next input file from the queue,
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
6. Go to 2.

The discovered test cases are also periodically culled to eliminate ones that have been obsoleted by newer, higher-coverage finds; and undergo several other instrumentation-driven effort minimization steps.
Coverage-Guided Fuzzing (e.g. AFL)
Coverage-Guided Fuzzing with AFL

Pulling JPEGs out of thin air

This is an interesting demonstration of the capabilities of AFL: I was actually pretty surprised that it worked!

```bash
$ mkdir in_dir
$ echo 'hello' > in_dir/hello
$ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg
```

# Coverage-Guided Fuzzing with AFL

## The bug-o-rama trophy case

<table>
<thead>
<tr>
<th>lig/jpeg 1</th>
<th>libjpeg-turbo 12</th>
<th>libpng 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>libtiff 12345</td>
<td>mozjpeg 1</td>
<td>PHP 12345678</td>
</tr>
<tr>
<td>Mozilla Firefox 1234</td>
<td>Internet Explorer 1234</td>
<td>Apple Safari 1</td>
</tr>
<tr>
<td>Adobe Flash / PCRE 1234567</td>
<td>sqlite 1234567</td>
<td>OpenSSL 1234567</td>
</tr>
<tr>
<td>LibreOffice 1234</td>
<td>poppler 1234</td>
<td>freetype 12</td>
</tr>
<tr>
<td>GnuTLS 1</td>
<td>GnuPG 1234</td>
<td>OpenSSH 12345</td>
</tr>
<tr>
<td>PuTTY 12</td>
<td>ntpd 12</td>
<td>nginx 123</td>
</tr>
<tr>
<td>bash (post-Shellshock) 12</td>
<td>tcpdump 123456789</td>
<td>JavaScriptCore 1234</td>
</tr>
<tr>
<td>pdfium 12</td>
<td>ffmpeg 12345</td>
<td>libmatroska 1</td>
</tr>
<tr>
<td>libarchive 123456789w</td>
<td>wireshark 123</td>
<td>ImageMagick 123456789w</td>
</tr>
<tr>
<td>BIND 123---</td>
<td>QEMU 12</td>
<td>lems 1</td>
</tr>
</tbody>
</table>

[http://lcamtuf.coredump.cx/afl/](http://lcamtuf.coredump.cx/afl/)
ClusterFuzz @ Chromium

<table>
<thead>
<tr>
<th>ID</th>
<th>Pri</th>
<th>M</th>
<th>Stars</th>
<th>ReleaseBlock</th>
<th>Component</th>
<th>Status</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>1133812</td>
<td>1</td>
<td>---</td>
<td>2</td>
<td>---</td>
<td>Blink&gt;GetUserMedia&gt;Webcam</td>
<td>Untriaged</td>
<td>---</td>
</tr>
<tr>
<td>1133763</td>
<td>1</td>
<td>---</td>
<td>1</td>
<td>---</td>
<td>---</td>
<td>Untriaged</td>
<td>---</td>
</tr>
<tr>
<td>1133701</td>
<td>1</td>
<td>---</td>
<td>1</td>
<td>---</td>
<td>Blink&gt;JavaScript</td>
<td>Untriaged</td>
<td>---</td>
</tr>
<tr>
<td>1133254</td>
<td>1</td>
<td>---</td>
<td>2</td>
<td>---</td>
<td>---</td>
<td>Untriaged</td>
<td>---</td>
</tr>
<tr>
<td>1133124</td>
<td>1</td>
<td>---</td>
<td>1</td>
<td>---</td>
<td>---</td>
<td>Untriaged</td>
<td>---</td>
</tr>
<tr>
<td>113024</td>
<td>2</td>
<td>---</td>
<td>3</td>
<td>---</td>
<td>Internals&gt;Network</td>
<td>Started</td>
<td>dmcardle@ct</td>
</tr>
<tr>
<td>1132958</td>
<td>1</td>
<td>---</td>
<td>2</td>
<td>---</td>
<td>UI&gt;Accessibility; Blink&gt;Accessibility</td>
<td>Assigned</td>
<td>sin...@chrom</td>
</tr>
<tr>
<td>1132907</td>
<td>2</td>
<td>---</td>
<td>2</td>
<td>---</td>
<td>Blink&gt;JavaScript&gt;GC</td>
<td>Assigned</td>
<td>dinfuehr@chr</td>
</tr>
</tbody>
</table>
Can fuzzing be applied to unit testing?

- Where “inputs” are not just strings or binary files?
- Yes! Possible to randomly generate strongly typed values, data structures, API calls, etc.
- Recall: Property-Based Testing

```java
@Property
public void testSameLength(List<Integer> input) {
    var output : List<Integer> = sort(input);
    // Check length
    assert output.size() == input.size() : "Length should match";
}
```
Generators

Random List<Integer>

List list = new ArrayList();
while (randomBoolean()) {  // randomly stop/go
    list.append(randomInt());  // random element
} return list;

List list = new ArrayList();
int len = randomInt();  // pick a random length
for (int i = 0 to len) {
    list.append(randomInt());  // random element
} return list;

Exercise: Write a generator for Creating random HashMap<String, Integer>
Mutators

Mutator for list: List<Integer>

```java
int k = randomInt(0, len(list));
int action = randomChoice(ADD, DELETE, UPDATE);
switch (action) {
    case UPDATE: list.set(k, randomInt()); // update element at k
    case ADD: list.addAt(k, randomInt()); // add random element at k
    case DELETE: list.removeAt(k); // delete k-th element
}
```

Exercise: Write a mutator

HashMap<String, Integer>
The Fuzzing Book

https://www.fuzzingbook.org/

The Fuzzing Book
Tools and Techniques for Generating Software Tests
by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Heiser

About this Book

Welcome to "The Fuzzing Book!" Software has bugs, and catching bugs can involve lots of effort. This book addresses this problem by automating software testing, specifically by generating tests automatically. Recent years have seen the development of novel techniques that lead to dramatic improvements in test generation and software testing. They now are mature enough to be assembled in a book – even with executable code.

A Textbook for Paper, Screen, and Keyboard

You can use this book in four ways:
- You can read chapters in your browser. Check out the list of chapters in the menu above, or start right away with the introduction to

---

ANU SCHOOL OF COMPUTING | COMP 2120 / COMP 6120 | WEEK 11 OF 12: STATIC ANALYSIS
TESTING PERFORMANCE
Performance Testing

• Goal: Identify *performance bugs*. What are these?
  • Unexpected bad performance on some subset of inputs
  • Performance degradation over time
  • Difference in performance across versions or platforms

• Not as easy as functional testing. What’s the oracle?
  • Fast = good, slow = bad // but what’s the threshold?
  • How to get reliable measurements?
  • How to debug where the issue lies?
Performance Regression Testing

- Measure execution time of critical components
- Log execution times and compare over time

Source: https://chromium.googlesource.com/chromium/src/+refs/heads/main/docs/speed/addressing_performance_regression.md
A Study of Performance Variations in the Mozilla Firefox Web Browser

Jan Larres¹

Alex Potanin¹

Yuichi Hirose²

¹ School of Engineering and Computer Science
Email: {larresjan,alex}@ecs.vuw.ac.nz

² School of Mathematics, Statistics and Operations Research
Email: hirose@msor.vuw.ac.nz
Victoria University of Wellington, New Zealand

While hacking on the Talos harness in the summer of 2011 to add support for new platforms and tests, we encountered the results from Jan Larres’s master’s thesis, in which he investigated the large amounts of noise that appeared in the Talos tests. He analyzed various factors including hardware, the operating system, the file system, drivers, and Firefox that might influence the results of a Talos test. Building on that work, Stephen Lewchuk devoted his internship to trying to statistically reduce the noise we saw in those tests.

Based on their work and interest, we began forming a plan to eliminate or reduce the noise in the Talos tests. We brought together harness hackers to work on the harness itself, web developers to update Graph Server, and statisticians to determine the optimal way to run each test to produce predictable results with minimal noise.

At Mozilla, one of our very first automated tests cannot easily be attributed to either genuine changes or change in the harness since its inception in 2007; even though many of the original assumptions are broken over time, these tests remain as one of the few changed by hand.

In the summer of 2011, we finally began to look askance at the noise and the variation in the Talos numbers, and we began to wonder how we could make some small modification to the system to start improving it. We had no idea we were about to open Pandora’s Box.

In this chapter, we will detail what we found as we peeled back layer after layer of this software, what problems we uncovered, and what steps we took to address them in hopes that you might learn from both our mistakes and our successes.

ANU SCHOOL OF COMPUTING | COMP 2120 / COMP 6120 | WEEK 11 OF 12: STATIC ANALYSIS
Profiling

- Finding bottlenecks in execution time and memory
- Flame graphs are a popular visualization of resource consumption by call stack.
Domain-Specific Perf Testing (e.g. JMeter)

http://jmeter.apache.org
Performance-driven Design

• Modeling and simulation
  • e.g. queuing theory
• Specify load distributions and derive or test configurations
Stress testing

• Robustness testing technique: test beyond the limits of normal operation.

• Can apply at any level of system granularity.

• Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered “correct” behavior under normal circumstances.
Soak testing

• **Problem:** A system may behave exactly as expected under artificially limited execution conditions.
  • E.g., Memory leaks may take longer to lead to failure (also motivates static/dynamic analysis, but we’ll talk about that later).

• **Soak testing:** testing a system with a significant load over a significant period of time (*positive*).

• Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.
Slides credit Christopher Meiklejohn

CHAOS ENGINEERING
What kind of failures can happen here?

How likely is that error to happen?

How do I fix it?

Monolithic Application
What kind of failures can happen here?
How likely is that error to happen?
How do I fix it?

Remember, these calls are messages sent on an unreliable network.
Failures in Microservice Architectures

1. Network may **be partitioned**

2. Server instance **may be down**

3. Communication between services may **be delayed**

4. Server **could be overloaded** and responses delayed

5. Server **could run out of** memory or CPU

All of these issues can be indistinguishable from one another!

Making the calls across the network to multiple machines makes the probability that the system is operating under failure **much higher**.

These are the problems of **latency** and **partial failure**.
Where Do We Start?

How do we even **begin to test these scenarios?**

Is there any **software** that can be used to test these types of failures?

Let’s look at a **few ways** companies do this.
Game Days

Purposely injecting failures into critical systems in order to:

• Identify flaws and “latent defects”
• Identify subtle dependencies (which may or may not lead to a flaw/defect)
• Prepare a response for a disastrous event

Comes from “resilience engineering” typical in high-risk industries

Practiced by Amazon, Google, Microsoft, Etsy, Facebook, Flickr, etc.
Game Days

Our applications are built on and with “unreliable” components

Failure is inevitable (fraction of percent; at Google scale, ~multiple times)

Goals:

• **Preemptively trigger** the failure, observe, and fix the error
• Script testing of **previous failures** and ensure system remains resilient
• Build the necessary relationships between teams **before** disaster strikes
Example: Amazon GameDay

Full data center destruction (Amazon EC2 region)

- No advanced notice of which data center will be taken offline
- No notice of when the data center will be taken offline
- Only advance notice (months) that a GameDay will be happening
- Real failures in the production environment

Discovered latent defect where the monitoring infrastructure responsible for detecting errors and paging employees was located in the zone of the failure!
Cornerstones of Resilience

1. Anticipation: know what to expect

2. Monitoring: know what to look for

3. Response: know what to do

4. Learning: know what just happened (e.g., postmortems)
Some Example Google Issues

Terminate network in Sao Paulo for testing:
• Hidden dependency takes down links in Mexico which would have remained undiscovered without testing

Turn off data center to find that machines won’t come back:
• Ran out of DHCP leases (for IP address allocation) when a large number of machines come back online unexpectedly.
Netflix: Cloud Computing

Significant deployment in Amazon Web Services in order to remain **elastic** in times of high and low load (first public, 100% w/o content delivery.)

Pushes code into production and modifies runtime configuration hundreds of times a day

Key metric: **availability**
Chaos monkey/Simian army

- A Netflix infrastructure testing system.
- “Malicious” programs randomly trample on components, network, datacenters, AWS instances...
  - Chaos monkey was the first – disables production instances at random.
  - Other monkeys include Latency Monkey, Doctor Monkey, Conformity Monkey, etc...
    Fuzz testing at the infrastructure level.
  - Force failure of components to make sure that the system architecture is resilient to unplanned/random outages.
- Netflix has open-sourced their chaos monkey code.
Netflix UI: AppBoot

What happens if the bookmark service is **down**?

- My List
- Bookmarks
- User Profiles
- Ratings
- Recommendations
- Search
- AppBoot

- Remote Call
- Microservice
Graceful Degradation: Anticipating Failure

Allow the system to degrade in a way it’s still usable

Fallbacks:
• Cache miss due to failure of cache;
• Go to the bookmarks service and use value at possible latency penalty

Personalized content, use a reasonable default instead:
• What happens if recommendations are unavailable?
• What happens if bookmarks are unavailable?
Principles of Chaos Engineering

1. Build a **hypothesis** around steady state behavior

2. Vary **real-world events**
   experimental events, crashes, etc.

3. Run **experiments** in production
   control group vs. experimental group
   draw conclusions, invalidate hypothesis

4. Automate **experiments** to run continuously

Does everything seem to be **working properly?**

Are users **complaining?**
Steady State Behavior

Back to **quality attributes: availability!**

FIGURE 2. A graph of SPS (requests per second) over a 24-hour period. This metric varies slowly and predictably throughout a day. The orange line shows the trend for the prior week. The y-axis isn’t labeled because the data is proprietary.

SPS is the primary indicator of the system’s overall health.
Mini Break in Monday Lecture
TESTING USABILITY
Automating GUI/Web Testing

- This is hard
- Capture and Replay Strategy
  - mouse actions
  - system events
- Test Scripts: (click on button labeled "Start" expect value X in field Y)
- Lots of tools and frameworks
  - e.g. Selenium for browsers
- (Avoid load on GUI testing by separating model from GUI)
- Beyond functional correctness?
Manual Testing

- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?
- Higher Quality Feedback to Developers

**Generic Test Case: User Sends MMS with Picture Attached.**

<table>
<thead>
<tr>
<th>Step ID</th>
<th>User Action</th>
<th>System Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Go to Main Menu</td>
<td>Main Menu appears</td>
</tr>
<tr>
<td>2</td>
<td>Go to Messages Menu</td>
<td>Message Menu appears</td>
</tr>
<tr>
<td>3</td>
<td>Select “Create new Message”</td>
<td>Message Editor screen opens</td>
</tr>
<tr>
<td>4</td>
<td>Add Recipient</td>
<td>Recipient is added</td>
</tr>
<tr>
<td>5</td>
<td>Select “Insert Picture”</td>
<td>Insert Picture Menu opens</td>
</tr>
<tr>
<td>6</td>
<td>Select Picture</td>
<td>Picture is Selected</td>
</tr>
<tr>
<td>7</td>
<td>Select “Send Message”</td>
<td>Message is correctly sent</td>
</tr>
</tbody>
</table>
Usability: A/B testing

- Controlled randomized experiment with two variants, A and B, which are the control and treatment.
- One group of users given A (current system); another random group presented with B; outcomes compared.
- Often used in web or GUI-based applications, especially to test advertising or GUI element placement or design decisions.
Example

• A company sends an advertising email to its customer database, varying the photograph used in the ad...
Example: group A (99% of users)

Act now! Sale ends soon!
Example: group B (1%)
A/B Testing

• Requires good metrics and statistical tools to identify significant differences.
• E.g. clicks, purchases, video plays
• Must control for confounding factors
What smells?

class Foo {
    int a; int b;

    public boolean equals(Object other) {
        Foo foo = (Foo) other;
        if (foo != null)
            if (foo.a != this.a)
                return false;
        if (foo.b == this.b)
            return true;
        else return false;
    }

    public int a() {
        return this.a();
    }

    public int b() {
        return this.b();
    }
}

What smells?

```c
int dtls1_process_heartbeat(SSL *s)
{
    unsigned char *p = &s->s3->rec.data[0], *pl;
    unsigned short hbytype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbytype = *p++;
    n23(p, payload);
    pl = p;

    if (s->msg_callback)
        s->msg_callback(s, s->version, TLS1_RT_HEARTBEAT,
                        &s->s3->rec.data[0], s->s3->rec.length,
                        s, s->msg_callback_arg);

    if (hbytype == TLS1_HB_REQUEST)
    {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 byte
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
    }
}
```
Static Analysis

• Try to discover issues by analyzing source code. No need to run.
• Defects of interest may be on uncommon or difficult-to-force execution paths for testing.
• What we really want to do is check the entire possible state space of the program for particular properties.
Defects Static Analysis can Catch

- Defects that result from inconsistently following simple design rules.
  - **Security**: Buffer overruns, improperly validated input.
  - **Memory safety**: Null dereference, uninitialized data.
  - **Resource leaks**: Memory, OS resources.
  - **API Protocols**: Device drivers; real time libraries; GUI frameworks.
  - **Exceptions**: Arithmetic/library/user-defined
  - **Encapsulation**: Accessing internal data, calling private functions.
  - **Data races**: Two threads access the same data without synchronization

*Key: check compliance to simple, mechanical design rules*
ANU SCHOOL OF COMPUTING | COMP 2120 / COMP 6120 | WEEK 11 OF 12: STATIC ANALYSIS

https://github.com/marketplace?category=code-quality
```java
package com.google.devtools.staticanalysis;

public class Test {
    public boolean foo() {
        return getString() == "foo".toString();
    }

    public String getString() {
        return new String("foo");
    }
}
```

```java
package com.google.devtools.staticanalysis;

import java.util.Objects;

public class Test {
    private String foo;

    public boolean foo() {
        return Objects.equals(getString(), "foo".toString());
    }

    public String getString() {
        return new String("foo");
    }
}
```
How do they work?

class Foo {
    int a; int b;

    public boolean equals(Object other) {
        if (foo != null)
            if (foo.a != this.a)
                return false;
            else return true;

        public int a() {
            return this.a();
        }

        public int b() {
            return this.b();
        }
    }
}

int dtls1_process_heartbeat(SSL *s)
{
    unsigned char *p = s->s3->rrec.data[0], *pl;

    unsigned short htype;
    unsigned int padding = 16; /* Use minimum padding */

    htype = *p++;

    /* Read type and payload length first */
    n2s(p, payload);

    pt = p;

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                        &s->s3->rrec.data[0], s->s3->rrec.length,
                        s, s->msg_callback_arg);

    if (htype == TLS1_HB_REQUEST)
    {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 byte */
        *message type, plus 2 bytes payload length, plus
        /* payload, plus padding */

        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);

        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        /* Write bytes */
        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
Two fundamental concepts

• **Abstraction.**
  • Elide details of a specific implementation.
  • Capture semantically relevant details; ignore

• **Programs as data.**
  • Programs are just trees/graphs!
  • ...and we know lots of ways to analyze trees/graphs, right?
Defining Static Analysis

• **Systematic** examination of an **abstraction** of program **state space**.
  • Does not execute code! (like code review)

• **Abstraction**: A representation of a program that is simpler to analyze.
  • Results in fewer states to explore; makes d

• **Check if a particular property** holds over the entire state space:
  • Liveness: “something good eventually happens.”
  • Safety: “this bad thing can’t ever happen.”
  • Compliance with mechanical design rules.
The Bad News: Rice's Theorem

Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

"Any nontrivial property about the language recognized by a Turing machine is undecidable."

Henry Gordon Rice, 1953
SIMPLE SYNTACTIC AND STRUCTURAL ANALYSES
Type Analysis

```java
public void foo() {
    int a = computeSomething();

    if (a == "5")
        doMoreStuff();
}
```
Abstraction: abstract syntax tree

- Tree representation of the syntactic structure of source code.
  - Parsers convert concrete syntax into abstract syntax, and deal with resulting ambiguities.
- Records only the semantically relevant information.
  - Abstract: doesn’t represent every detail (like parentheses); these can be inferred from the structure.
- (How to build one? Take compilers!)

Example: $5 + (2 + 3)$
class X {
    Logger logger;
    public void foo() {
        ...
        if (logger.inDebug()) {
            logger.debug("We have "+conn + "connections.");
        }
    }
}
class Logger {
    boolean inDebug() {
        ...
    }
    void debug(String msg) {
    }
}
Syntactic Analysis

Find every occurrence of this pattern:

```
public foo() {
    ...
        logger.debug("We have " + conn + "connections.");
    }
```

```
public foo() {
    ...
        if (logger.inDebug()) {
            logger.debug("We have " + conn + "connections.");
        }
    }
```
Abstract syntax tree walker

- Check that we don’t create strings outside of a `Logger.inDebug` check
- Abstraction:
  - Look only for calls to `Logger.debug()`
  - Make sure they’re all surrounded by `if (Logger.inDebug())`
- Systematic: Checks all the code
- Known as an Abstract Syntax Tree (AST) walker
  - Treats the code as a structured tree
  - Ignores control flow, variable values, and the heap
  - Code style checkers work the same way
class X {
    Logger logger;
    public void foo() {
        ...
        if (logger.inDebug()) {
            logger.debug("We have "+ conn + "connections.");
        }
    }
}

class Logger {
    boolean inDebug() {...}
    void debug(String msg) {...}
}
Structural analysis for possible NPEs?

```java
if (foo != null)
    foo.a();
foo.b();
```
Which of these should be flagged for NPE?
Surely safe? Surely bad? Suspicious?

// Limitations of structural analysis

A

```java
if (foo != null)
    foo.a();
foo.b();
```

B

```java
if (foo == null)
    foo = new Foo();
foo.b();
```

C

```java
if (foo != null)
    foo.a();
else
    foo = new Foo();
foo.b();
```

D

```java
if (foo != null)
    foo.a();
else
    foo.b();
```
CONTROL-FLOW AND DATA-FLOW ANALYSIS
Control/Dataflow analysis

• **Reason** about all possible executions, via paths through a control flow graph.
  • Track information relevant to a property of interest at every program point.

• Define an **abstract domain** that captures only the values/states relevant to the property of interest.

• **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.
Control flow graphs

• A tree/graph-based representation of the flow of control through the program.
  • Captures all possible execution paths.
• Each node is a basic block: no jumps in or out.
• Edges represent control flow options between nodes.
  • Intra-procedural: within one function.
    • cf. inter-procedural

1. a = 5 + (2 + 3)
2. if (b > 10) {
   3.   a = 0;
   4. }
5. return a;
How can CFG be used to identify this issue?

```java
public int foo() {
    doStuff();
    return 3;
    doMoreStuff();
    return 4;
}
```
NPE analysis revisited

A
```
1 if (foo != null)
2    foo.a();
3    foo.b();
```

B
```
1 if (foo == null)
2    foo = new Foo();
3    foo.b();
```

C
```
1 if (foo != null)
2    foo.a();
3 else
4    foo = new Foo();
5
6    foo.b();
```

D
```
1 if (foo != null)
2    foo.a();
3 else
4    foo.b();
```
Abstract Domain for NPE Analysis

• Map of Var  ->  \{Null,.NotNull, Unknown\}

• For example:
  
  foo  ->  Null
  bar  -> NonNull
  baz  ->  Unknown

• Mapping tracked at every program point (before/after each CFG node). Updated across nodes and edges.

• // let’s say foo  ->  Null and bar  ->  Null
  foo = new Foo();
  // at this point, we have foo  ->  NotNull and bar  ->  Null
Data-Flow Analysis Examples

```
1  if (foo != null)
2      foo.a();
3  else
4      foo.b();
```
Data-Flow Analysis Examples

```
if (foo != null)
  foo.a()
else
  foo.b()
```

**ERROR!!!!**
Data-Flow Analysis Examples

```
if (foo != null)
    foo.a();
else
    foo = new Foo();
foo.b();
```
Data-Flow Analysis Examples

```java
if (foo != null)
    foo.a();
else
    foo = new Foo();
foo.b();
```
Data-Flow Analysis Examples

Exercise: Work this out for yourself. Is foo.b() safe?

```
1 if (foo == null)
2   foo = new Foo();
3   foo.b();
```
Data-Flow Analysis Examples

if (foo == null)
    Then
    foo = new Foo()
    Else

foo.b()
Data-Flow Analysis Examples

```java
if (foo == null)
    foo = new Foo()
```

- `{foo -> Unknown}`
- `{foo -> Null} Then
- `{foo -> NotNull}
- `{foo -> Null} Else
- `{foo -> NotNull}
- `{foo -> NotNull`
- `{foo -> NotNull}

```java
1. if (foo == null)
2.    foo = new Foo();
3.    foo.b();
```
Interpreting abstract states

• “Null” means “must be NULL at this point, regardless of path taken”
• “NotNull” is similar
• “Unknown” means “may be NULL or not null depending on the path taken”

• Unknown must be dealt with due to Rice’s theorem
  • Can make analysis smarter (at the cost of more algorithmic complexity) to reduce Unknowns, but can’t get rid of them completely

• Whether to raise a flag on UNKNOWN access depends on usability/soundness.
  • False positives if warning on UNKNOWN
  • False negatives if no warning on UNKNOWN
Sound Analysis

All Defects

Complete Analysis

Unsound and Incomplete Analysis
Examples of Data-Flow Analyses

• Null Analysis
  • Var -> {Null, NotNull, UNKNOWN}

• Zero Analysis
  • Var -> {Zero, NonZero, UNKNOWN}

• Sign Analysis
  • Var -> {-, +, 0, UNKNOWN}

• Range Analysis
  • Var -> {[0, 1], [1, 2], [0, 2], [2, 3], [0, 3], ..., UNKNOWN}

• Constant Propagation
  • Var -> {1, 2, 3, ..., UNKNOWN}

• File Analysis
  • File -> {Open, Close, UNKNOWN}

• Tons more!!!
Data-Flow Analysis: Challenges

• Loops
  • Fixed-point algorithms guarantee termination at the cost of losing information ("Unknown")
• Functions
  • Analyze them separately or analyze whole program at once
  • “Context-sensitive” analyses specialize on call sites (think: duplicate function body for every call site via inlining)
• Recursion
  • Makes context-sensitive analyses explode (cf. loops)
• Object-oriented programming
• Heap memory
  • Need to abstract mapping keys not just values
• Exceptions
Static Analysis vs. Testing

• Which one to use when?

• Points in favor of Static Analysis
  • Don’t need to set up run environment, etc.
  • Can analyze functions/modules independently and in parallel
  • Don’t need to think of (or try to generate) program inputs

• Points in favor of Testing / Dynamic Analysis
  • Not deterred by complex program features
  • Can easily handle external libraries, platform-specific config, etc.
  • Ideally no false positives
  • Easier to debug when a failure is identified
Key Points

• Describe random test-input generation strategies such as fuzz testing
• Write generators and mutators for fuzzing different types of values
• Characterize challenges of performance testing and suggest strategies
• Reason about failures in microservice applications
• Describe chaos engineering and how it can be applied to test resiliency of cloud-based applications
• Describe A/B testing for usability
Key Points

• Give a one sentence definition of static analysis. Explain what types of bugs static analysis targets.
• Give an example of syntactic or structural static analysis.
• Construct basic control flow graphs for small examples by hand.
• Give a high-level description of dataflow analysis and cite some example analyses.
• Explain at a high level why static analyses cannot be sound, complete, and terminating; assess tradeoffs in analysis design.
• Characterize and choose between tools that perform static analyses.
• Contrast static analysis tools with software testing and dynamic analysis tools as a means of catching bugs.