Software Security

Security Testing

especially

Fuzzing

Erik Poll

Radboud Universiteit Nijmegen

TRU/e Master in Cyber Security

Overview

- Testing
- Abuse cases & negative tests
- Fuzzing
  - Dumb fuzzing
  - Mutational Fuzzing
    - example: OCPP
  - Generational aka grammar-based fuzzing
    - example: GSM
  - Whitebox fuzzing with **SAGE**
    - looking at symbolic execution of the code
  - Evolutionary fuzzing with `afl`
    - grey-box, observing execution of the (instrumented) code

Testing

SUT, test suite & test oracle

To test a SUT (System Under Test) we need two things

1. test suite, ie. collection of input data
2. a test oracle
   which decides if a test was passed ok or reveals an error
   – ie. some way to decide if the SUT behaves as we want

Both defining test suites and test oracles can be *a lot of work!*

- In the worst case, a test oracle is a long list which *for every individual test case, specifies exactly what should happen*
- A simple test oracle: just looking if application doesn't crash
  – *Moral of the story: crashes are good ! (for testing)*

Code coverage criteria

Code coverage criteria to measure how good a test suite is include

- statement coverage
- branch coverage

  Statement coverage does not imply branch coverage; eg for

  ```
  void f (int x, y) { if (x>0) {y++};
                                y--; }
  ```

  Here statement coverage needs 1 test case,
  branch coverage needs 2

- More complex coverage criteria exists, eg MCDC (Modified condition/decision coverage), commonly used in avionics

Possible perverse effect of coverage criteria

High coverage criteria may *dis*courage defensive programming, eg.

```
void m(File f){
  if <security_check_fails> {log (...);
                              throw (SecurityException);}
  try { <the main part of the method> }
  catch (SomeException) { log(...);
                          <some corrective action>;
                          throw (SecurityException); }
}
```

If the green defensive code, ie. the if & catch branches, are hard to trigger in tests, then programmers may be tempted (or forced) to remove this code to improve test coverage...

Annotations as test oracle

- Annotations in code, eg
  - SAL/PREfast annotations of C/C++ code
  - JML annotations of Java (last week, for 6EC version)

  can be used as test oracle by doing runtime assertion checking
  - So annotations provide a test oracle for free! You can test by sending random data, and see if annotations are violated

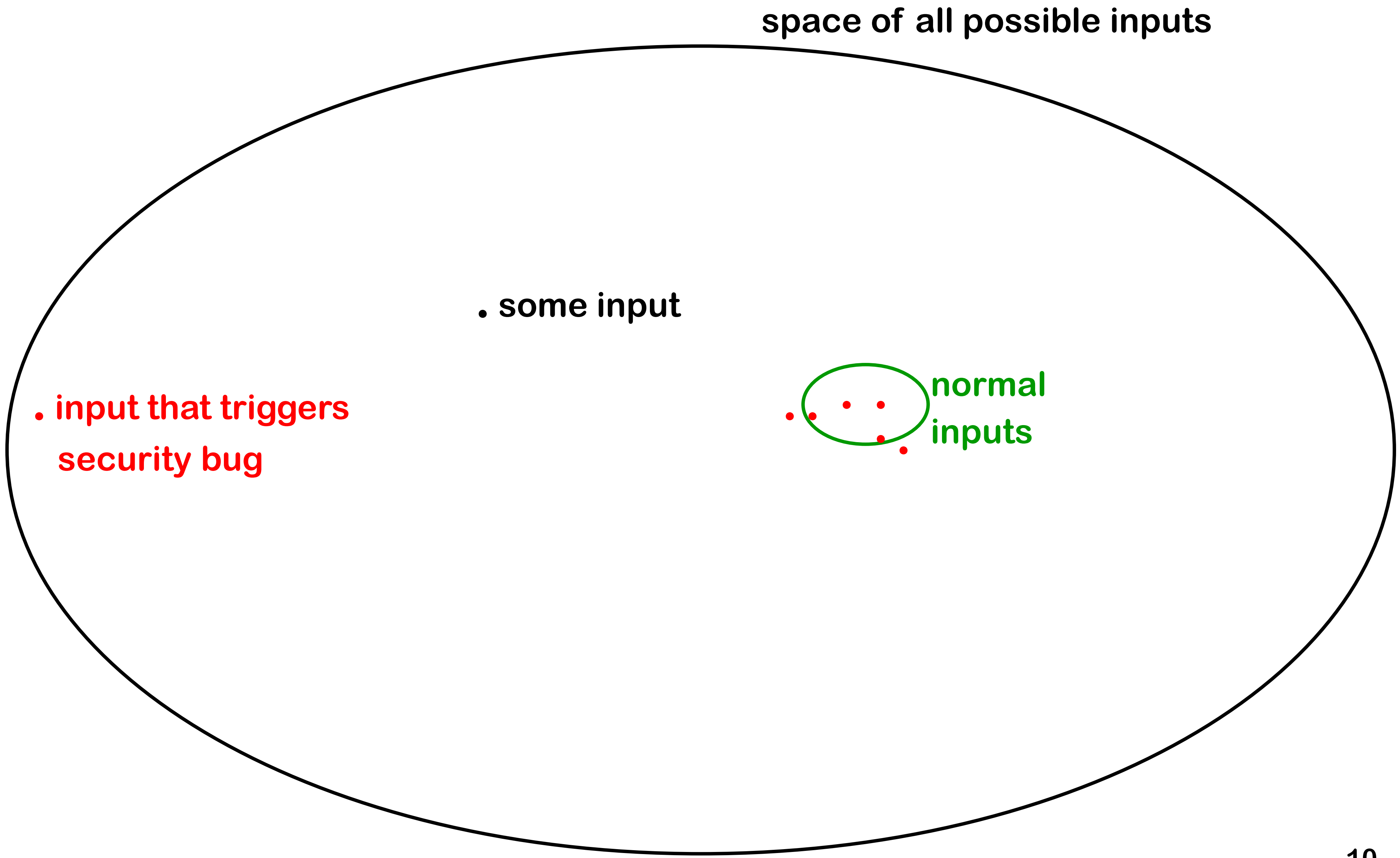
- Information flow policies (e.g. of SPARTA policies for Android) can also be used as test oracles
  - But: runtime checking for these require heavy instrumentation of the code, to trace the origin of data *inside* the running application, so-called dynamic taint tracking

Abuse cases
&
Negative testing

Testing for functionality vs testing for security

- Normal testing will look at right, wanted behaviour for sensible inputs (aka the happy flow), and some inputs on borderline conditions
- Security testing also requires looking for the wrong, unwanted behaviour for really strange inputs
- Similarly, normal use of a system is more likely to reveal functional problems than security problems:
  - users will complain about functional problems, hackers won't complain about security problems

Security testing is HARD


space of all possible inputs
some input
input that triggers security bug
normal inputs

Abuse cases & negative test cases

- Thinking about abuse cases is a useful way to come up with security tests
  - *what would an attacker try to do?*
  - *where could an implementation slip up?*

- This gives rise to negative test cases:

  test which are *supposed* to fail

iOS goto fail SSL bug

```
...
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
err = sslRawVerify(...);
. . .
```

Negative test cases for flawed certificate chains

- David Wheeler's 'The Apple goto fail vulnerability: lessons learned' gives a good discussion of this bug & ways to prevent it, incl. the need for negative test cases

  http://www.dwheeler.com/essays/apple-goto-fail.html

- The FrankenCert test suite provides (broken) certificate chains to test for flaws in the program logic for handling certificate flaws.

  [Brubaker et al, Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations, Oakland 2014]

Other ways to spot this bug:

- Compiler warnings about dead code (aka unreachable code) could (should?) also have spotted this bug
- Code coverage requirements on the test suite would also have helped.
- Coding guidelines, notably to *always use { } , even for single statements*, would also have helped.

Fuzzing

Fuzzing

- Fuzzing aka fuzz testing is a highly effective, largely automated, security testing technique
- Basic idea: (semi) automatically generate random inputs and see if an application crashes
- The original form of fuzzing: generate very long inputs and see if the system crashes with a segmentation fault.
  - *What kind of bug would such a segfault signal?*
    - A buffer overflow problem
  - *Why would inputs ideally be very long?*
    - To make it likely that buffer overruns cross segment boundaries

Simple fuzzing ideas

What inputs would you use for fuzzing?

- **very long or completely blank strings**
- **max. or min. values of integers, or simply zero and negative values**
- **depending on what you are fuzzing, include special values, characters or keywords likely to trigger bugs, eg**
  - **nulls, newlines, or end-of-file characters**
  - **format string characters**
  - **semi-colons, slashes and backslashes, quotes**
  - **application specific keywords** `halt, DROP TABLES, ...`
  - **....**

Pros & cons of fuzzing

Pros

- Very little effort:
  - the test cases are automatically generated,
    as test oracle is simply looking for crashes
- Fuzzing of a C/C++ binary can quickly give a good picture of robustness of the code

Cons

- Will not find all bugs
- Crashes may be hard to analyse; but a crash is a clear *true positive* that something is wrong!
- For programs that take complex inputs, more work will be needed to get good code coverage, and hit interesting test cases.
  This has lead to lots of work on 'smarter' fuzzers.

Crash/error detection

- Crash detection is critical for fuzzing!
- To help in crash detection, debugging tools aka runtime checkers can be used, such as `valgrind`, `Purify`, `AddressSanitizer`, ...

  Such tools instrument code or run to code on simulators to catch more bugs

  Eg `Valgrind` provides

  - `memcheck` memory error detector, which detects buffer overruns, malloc/free errors, memory leaks, reads of uninitialised memory, ...
  - `helgrind` detector to help detect data races, deadlocks, and incorrect use of the POSIX threads API

Fuzzing web-applications

- *How could a fuzzer detect SQL injections or XSS weaknesses?*
  - For SQL injection: monitor database for error messages
  - For XSS, see if the website echoes HTML tags in user input
- There are various tools to fuzz web-applications: Spike proxy, HP Webinspect, AppScan, WebScarab, Wapiti, w3af, RFuzz, WSFuzzer, SPI Fuzzer Burp, Mutilidae, ...
- Some fuzzers crawl a website, generating traffic themselves, other fuzzers modify traffic generated by some other means.
- *Can we expect false positives/negatives?*
  - false negatives due to test cases not hitting the vulnerable cases
  - false positives & negatives due to incorrect test oracle, eg
    - for SQL injection: not recognizing some SQL database errors (false neg)
    - for XSS: signaling quoted echoed response as XSS (false pos)

Smarter fuzzing

1) Mutation-based fuzzers: apply random mutations to existing valid inputs
   - Eg observe network traffic, than replay with some modifications
   - More likely to produce interesting invalid inputs than just random input
2) Generation-based aka grammar-based fuzzers
   generate semi-well-formed inputs from scratch, based on some knowledge of file format or network protocol
   – Downside: more work to construct the fuzzer
3) Evolutionary fuzzers: observe how inputs are processed to learn which mutations are interesting
   - For example, afl, which uses a greybox approach
4) Whitebox approaches: analyse source code to determine interesting inputs
   - For example, SAGE

Example mutational fuzzing

Example: Fuzzing OCPP [research internship Ivar Derksen]

- OCPP is a protocol for charge points to talk to a back-end server
- OCPP can use XML or JSN messages

Example message in JSN format

```
{ "location": "some identifier",
  "retries": 5,
  "retryInterval": 30,
  "startTime": "2018-10-27T19:10:11",
  "stopTime":  "2018-10-27T22:10:11"  }
```

Example: Fuzzing OCPP

Simple classification of messages into

1. malformed JSN/XML

   (eg missing quote, bracket or comma)

2. well-formed JSN/XML, but not legal OCPP

   (eg using field names that are not in the OCPP specs)

3. well-formed OCPP

can be used for a simple test oracle:

- Malformed messages (type 1 & 2) should generate generic error responses
- Wellformed messages (type 3) should not
- The application should never crash

This does not involve any understanding of the protocol semantics yet!

Test results with fuzzing OCPP server

- Mutation fuzzer generated 26,400 variants from 22 example OCPP messages in JSN format
- Problems spotted by Ivar's simple test oracle:
  - 945 malformed JSN requests (type 1) resulted in malformed JSN response

    Server should never emit malformed JSN!
  - 75 malformed JSN requests (type 1) and 40 malformed OCPP requests (type 2) result in a valid OCPP response that is not an error message.

    Server should not process malformed requests!

- Root cause: the Google's `gson` library for parsing JSN by default uses lenient mode rather than strict mode

Generational fuzzing
aka
Grammar-based fuzzing

CVEs as inspiration for fuzzing file formats

- Microsoft Security Bulletin MS04-028

  Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution

  Impact of Vulnerability: Remote Code Execution

  Maximum Severity Rating: Critical

  Recommendation: Customers should apply the update immediately

  Root cause: a zero sized comment field, without content.

- CVE-2007-0243

  Sun Java JRE GIF Image Processing Buffer Overflow Vulnerability

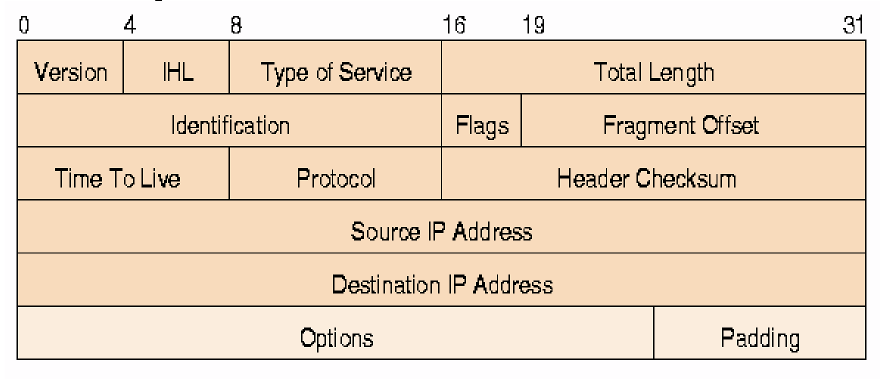
  Critical: Highly critical   Impact: System access   Where: From remote

  Description: A vulnerability has been reported in Sun Java Runtime Environment (JRE). … The vulnerability is caused due to an error when processing GIF images and can be exploited to cause a heap-based buffer overflow via a specially crafted GIF image with an image width of 0. Successful exploitation allows execution of arbitrary code.

  Note: a buffer overflow in (native library of) a memory-safe language

Generation-based fuzzing

For a given file format or communication protocol, a generational fuzzer tries to generate files or data packets that are slightly malformed or hit corner cases in the spec.



Possible starting :

grammar defining legal inputs, or a data format specification

Typical things to fuzz:

- many/all possible value for specific fields
  esp undefined values, or values Reserved for Future Use (RFU)
- incorrect lengths, lengths that are zero, or payloads that are too short/long

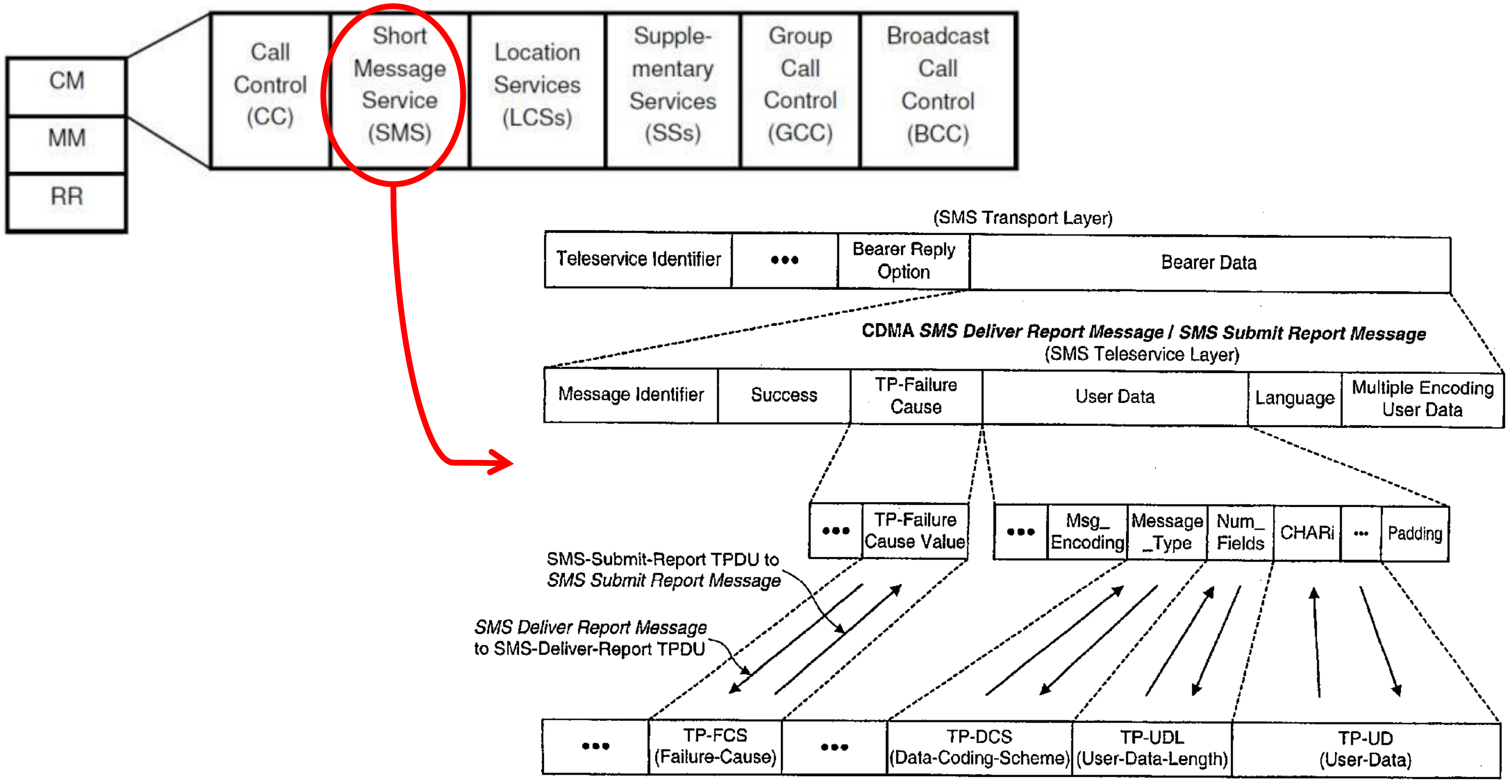
Tools for building such fuzzers:

SNOOZE, SPIKE, Peach, Sulley, antiparser, Netzob, ...

Example : GSM protocol fuzzing

[MSc theses of Brinio Hond and Arturo Cedillo Torres]

GSM is a extremely rich & complicated protocol



SMS message fields

Field	size
Message Type Indicator	2 bit
Reject Duplicates	1 bit
Validity Period Format	2 bit
User Data Header Indicator	1 bit
Reply Path	1 bit
Message Reference	integer
Destination Address	2-12 byte
Protocol Identifier	1 byte
Data Coding Scheme (CDS)	1 byte
Validity Period	1 byte/7 bytes
User Data Length (UDL)	integer
User Data	depends on CDS and UDL

Example: GSM protocol fuzzing

Lots of stuff to fuzz!

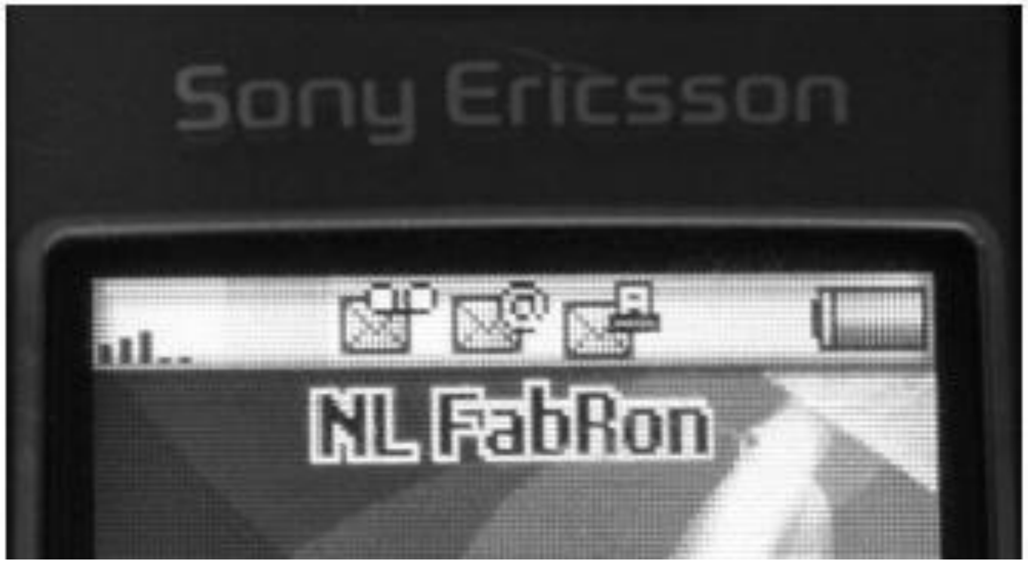
We can use a USRP

with open source cell tower software (OpenBTS)

to fuzz any phone

Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones

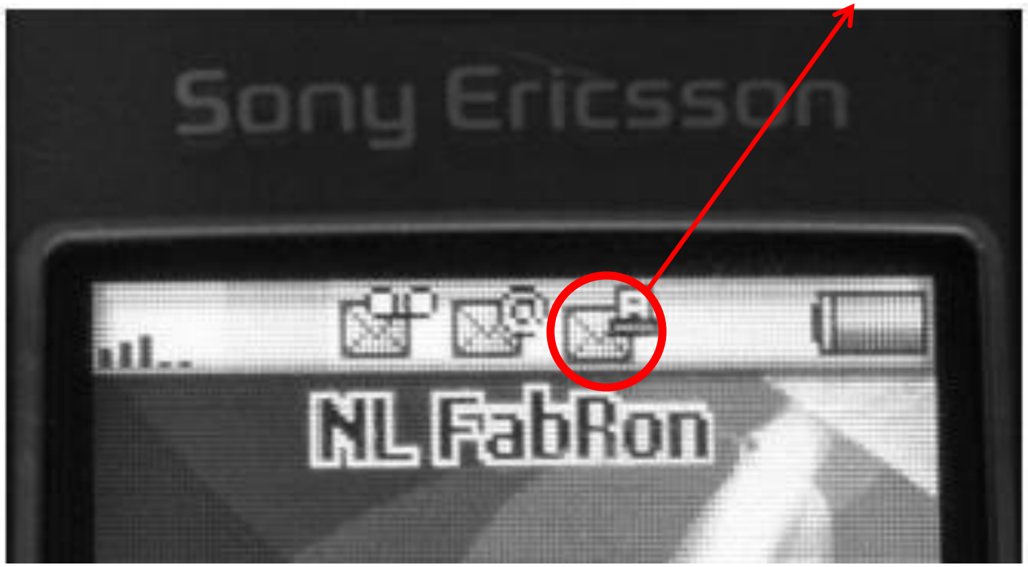


Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones

- eg possibility to receive faxes (!?)

you have a fax!

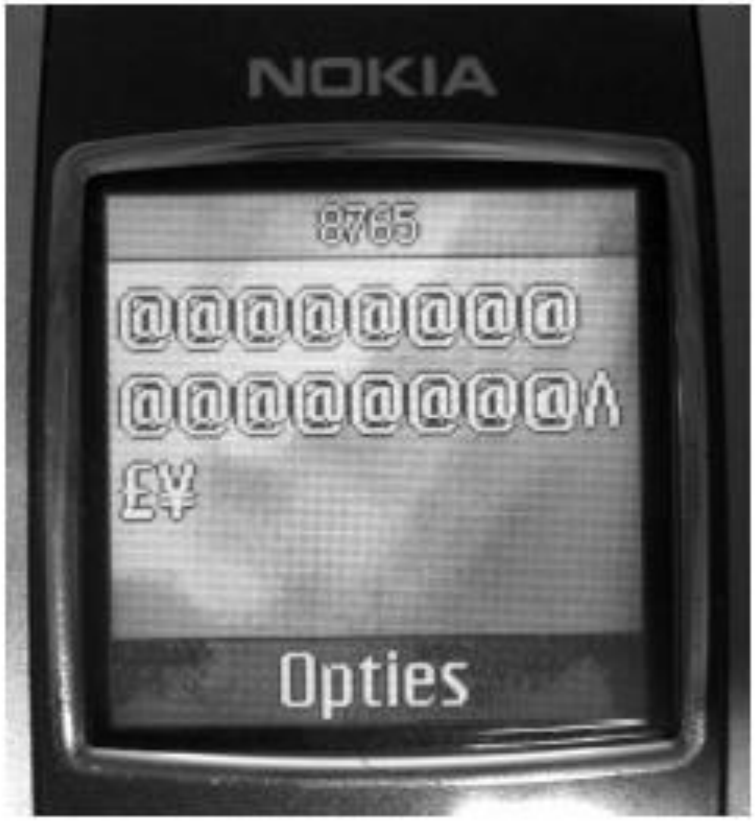


Only way to get rid if this icon; reboot the phone

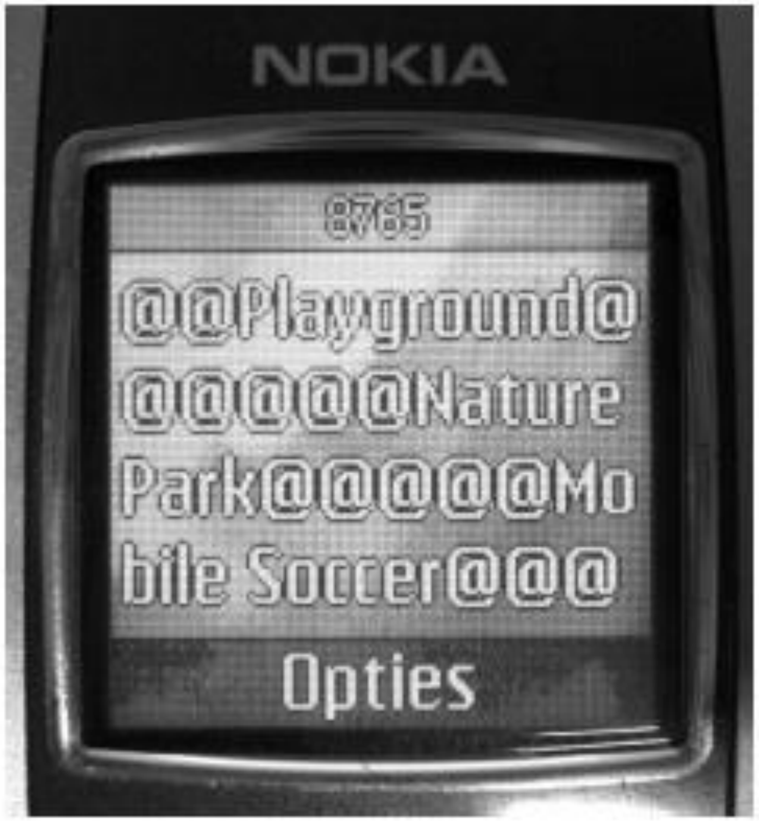
Example: GSM protocol fuzzing

Malformed SMS text messages showing raw memory contents, rather than content of the text message

(a) Showing garbage



(b) Showing the name of a wallpaper and two games



Our results with GSM protocol fuzzing

- Lots of success to DoS phones: phones crash, disconnect from the network, or stop accepting calls
  - eg requiring reboot or battery removal to restart, to accept calls again, or to remove weird icons
  - after reboot, the network might redeliver the SMS message, if no acknowledgement was sent before crashing, re-crashing phone

  But: not all these SMS messages could be sent over real network
- There is surprisingly little correlation between problems and phone brands & firmware versions
  - how many implementations of the GSM stack did Nokia have?
- *The scary part: what would happen if we fuzz base stations?*

[Fabian van den Broek, Brinio Hond and Arturo Cedillo Torres, Security Testing of GSM Implementations, Essos 2014]

[Mulliner et al., SMS of Death, USENIX 2011]

Whitebox fuzzing with SAGE

Whitebox fuzzing using symbolic execution

- The central problem with fuzzing:
  how can we generate inputs that trigger interesting code executions?
  - Eg fuzzing the procedure below is unlikely to hit the error case

```
int foo(int x) {
    y = x+3;
    if (y==13) abort(); // error
}
```

- The idea behind whitebox fuzzing: if we know the code, then by analysing the code we can find interesting input values to try.
- SAGE (Scalable Automated Guided Execution) is a tool from Microsoft Research that uses symbolic execution of x86 binaries to generate test cases.

```
m(int x,y){
   x = x + y;
   y = y - x;
   if (2*y > 8) { ...
                }
   else if (3*x < 10){ ...
                     }
   }
```

Can you provide values for `x` *and* `y` *that will trigger execution of the two if-branches?*

Symbolic execution

```
m(int x,y){

   x = x + y;

   y = y - x;

   if (2*y > 8) { ...

             }

   else if (3*x < 10){ ...

                  }

   }
```

Suppose `x` *= N and* `y` *= M.*

`x` *becomes N+M*

`y` *becomes M-(N+M) = -N*

if-branch taken if $-2N > 8$, ie $N < -4$

Aka the path condition

2nd if-branch taken if
$N \geq -4$ & $3(M+N) < 10$

SMT solvers (such as Yikes, Z3, ...) are tools that can simplify such constraints and produce test data that meets them, or prove that they are not satisfiable.

This generates test data (i) *automatically* and (ii) *with good coverage.*

Symbolic execution for test generation

- Symbolic execution can be used to automatically generate test cases with good coverage
- Basic idea symbolic execution:
  instead of giving variables a concrete value (say 42), variables are given a symbolic value (say α), and the program is executed with these symbolic values to see when certain program points are reached
- Downside of symbolic execution:
  - it is very expensive (in time & space)
  - things explode with loops
  - …

  SAGE mitigates this by using a *single* symbolic execution to generate many test inputs for *many* execution paths

SAGE example

Example program

```
void top(char input[4]) {
      int cnt = 0;
      if (input[0] == 'b') cnt++;
      if (input[1] == 'a') cnt++;
      if (input[2] == 'd') cnt++;
      if (input[3] == '!') cnt++;
      if (cnt >= 3) crash();
  }
```

What would be interesting test cases? How could you find them?

SAGE example

Example program

```
void top(char input[4]) {
      int cnt = 0;
      if (input[0] == 'b') cnt++;
      if (input[1] == 'a') cnt++;
      if (input[2] == 'd') cnt++;
      if (input[3] == '!') cnt++;
      if (cnt >= 3) crash();
  }
```

path contraints:

$i_0 \neq$ 'b'

$i_1 \neq$ 'a'

$i_2 \neq$ 'd'

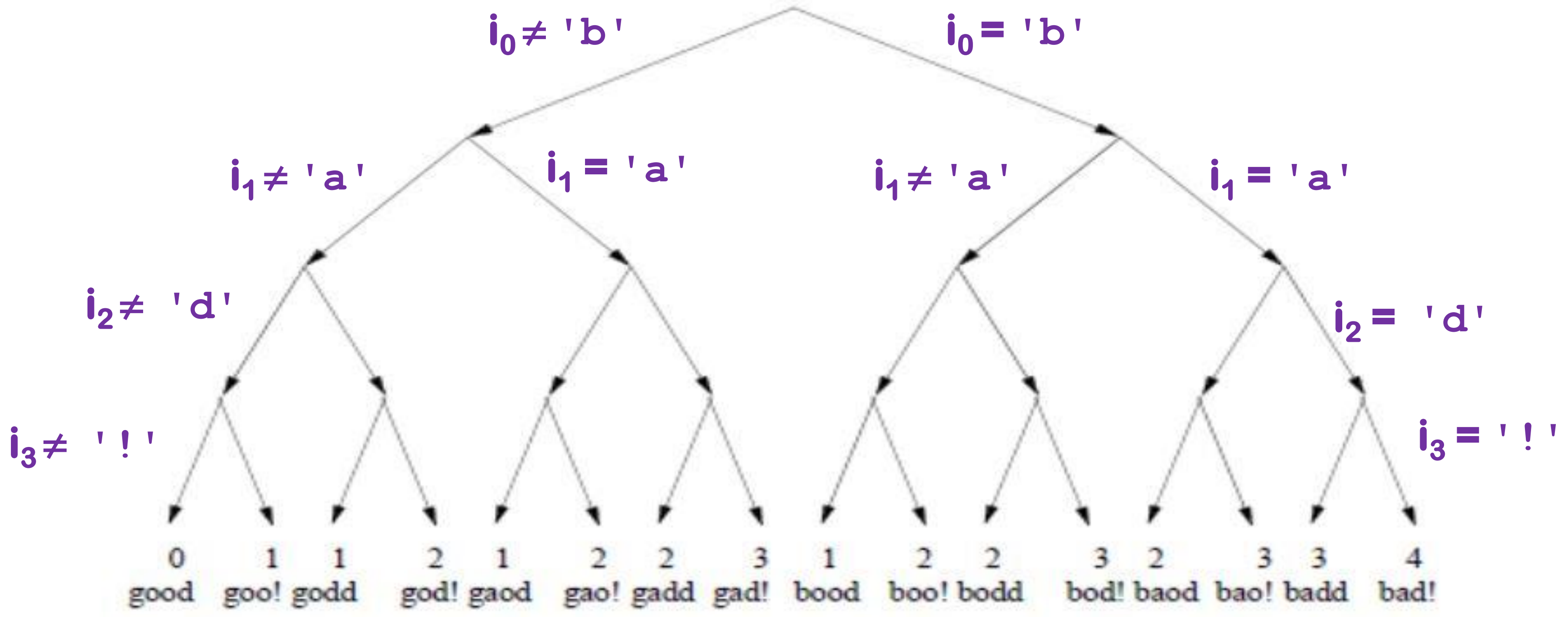
$i_3 \neq$ '!'

SAGE executes the code for some concrete `input`, say 'good'

It then collects *path constraints* for an arbitrary symbolic input, say $i_0 i_1 i_2 i_3$

Search space for interesting inputs

Based on this one execution, combining all these constraints now yields 16 test cases



Note: the initial execution with the input 'good' wass not very interesting, but these others are

SAGE success

- SAGE proved successful at uncovering security bugs, eg

  Microsoft Security Bulletin MS07-017 aka CVE-2007-0038: Critical

  Vulnerabilities in GDI Could Allow Remote Code Execution

  Stack-based buffer overflow in the animated cursor code in Microsoft Windows 2000 SP4 through Vista allows remote attackers to execute arbitrary code or cause a denial of service (persistent reboot) via a large length value in the second (or later) anih block of a RIFF .ANI, cur, or .ico file, which results in memory corruption when processing cursors, animated cursors, and icons

  Security vulnerablity in parsing the ANI-format. SAGE generated a well-formed ANI file triggering the bug, without knowing the ANI format.

- First experiments also found bugs in handling a compressed file format, media file formats, and generated 43 test cases to crash Office 2007

Evolutionary Fuzzing with `afl`

`(American Fuzzy Lop)`

Evolutionary Fuzzing with `afl`

- Downside of generation-based fuzzing:
  - lots of work work to write code to do the fuzzing, even if you use tools to generate this code based on some grammar
- Downside of mutation-based fuzzer:
  - chance that random changes in inputs hits interesting cases is small
- `afl` (American Fuzzy Lop) takes an *evolutionary* approach to learn interesting mutations based on measuring *code coverage*
  - basic idea: if a mutation of the input triggers a new execution path through the code, then it is an interesting mutation & it is kept; if not, the mutation is discarded.
  - by trying random mutations of the input and observering their effect on code coverage, `afl` can learn what interesting inputs are

afl [http://lcamtuf.coredump.cx/afl]

- Supports programs written in C/C++/Objective C and variants for Python/Go/Rust/OCaml
- Code instrumented to observe execution paths:
  - if source code is available, by using modified compiler
  - if source code is not available, by running code in an emulator
- Code coverage represented as a 64KB bitmap, where control flow jumps are mapped to changes in this bitmap
  - different executions could result in the same bitmap, but chance is small
- Mutation strategies include: bit flips, incrementing/decrementing integers, using pre-defined interesting integer values (eg. 0, -1, MAX_INT,....), deleting/combining/zeroing input blocks, ...
- The fuzzer forks the SUT to quickly process lots of test cases
- **Big win: no need to specify the input format!**

afl's instrumentation of compiled code

Code is injected at every branch point in the code

```
cur_location = <COMPILE_TIME_RANDOM_FOR_THIS_CODE_BLOCK>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

where `shared_mem` is a 64 KB memory region

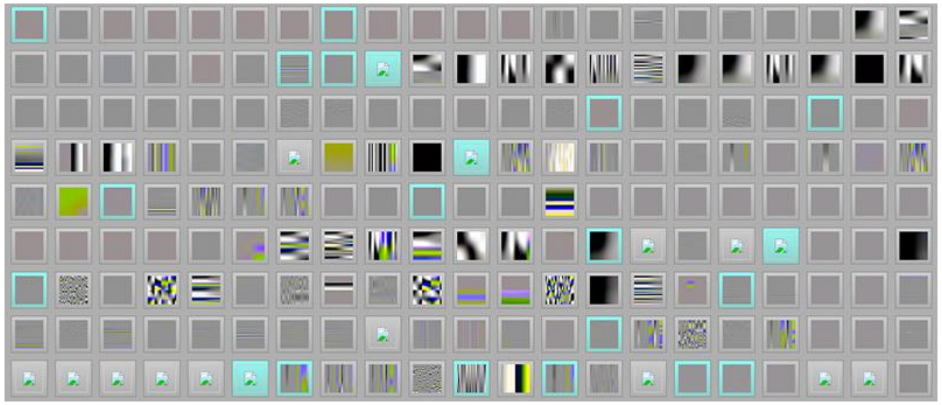
Intuition: for every jump from `src` to `dest` in the code a different byte in `shared_mem` is changed.

This byte is determined by the compile time randoms inserted at source and destination.

Cool example: learning the JPG file format

- Fuzzing a program that expects a JPG as input, starting with 'hello world' as initial test input, afl can learn to produce legal JPG files
  - along the way producing/discovering error messages such as
    - `Not a JPEG file: starts with 0x68 0x65`
    - `Not a JPEG file: starts with 0xff 0x65`
    - `Premature end of JPEG file`
    - `Invalid JPEG file structure: two SOI markers`
    - `Quantization table 0x0e was not defined`

and then JPGs like



[Source http://lcamtuf.blogspot.nl/2014/11/pulling-jpegs-out-of-thin-air.html]

Vulnerabilities found with `afl`

IJG jpeg 1
libtiff 1 2 3 4 5
Mozilla Firefox 1 2 3 4
Adobe Flash / PCRE 1 2 3 4
LibreOffice 1 2 3 4
GnuTLS 1
PuTTY 1 2
bash (post-Shellshock) 1 2
pdfium 1 2
libarchive 1 2 3 4 5 6 ...
BIND 1 2 3 ...
Oracle BerkeleyDB 1 2
FLAC audio library 1 2
strings (+ related tools) 1 2 3 4 5 6 7
Info-Zip unzip 1 2
NetBSD bpf 1
clamav 1 2 3 4 5
clang / llvm 1 2 3 4 5 6 7 8 ...
mutt 1
pdksh 1 2
redis / lua-cmsgpack 1
perl 1 2 3 4 5 6 7...
SleuthKit 1

libjpeg-turbo 1 2
mozjpeg 1
Internet Explorer 1 2 3 4
sqlite 1 2 3 4...
poppler 1
GnuPG 1 2 3 4
ntpd 1 2
tcpdump 1 2 3 4 5 6 7 8 9
ffmpeg 1 2 3 4 5
wireshark 1 2 3
QEMU 1 2
Android / libstagefright 1 2
libsndfile 1 2 3 4
file 1 2 3 4
libtasn1 1 2 ...
man & mandoc 1 2 3 4 5 ...
libxml2 1 2 4 5 6 7 8 9 ...
nasm 1 2
procmail 1
Qt 1 2...
taglib 1 2 3
libxmp
fwknop [reported by author]

libpng 1
PHP 1 2 3 4 5
Apple Safari 1
OpenSSL 1 2 3 4 5 6 7
freetype 1 2
OpenSSH 1 2 3
nginx 1 2 3
JavaScriptCore 1 2 3 4
libmatroska 1
ImageMagick 1 2 3 4 5 6 7 8 9 ...
lcms 1
iOS / ImageIO 1
less / lesspipe 1 2 3
dpkg 1 2
OpenBSD pfctl 1
IDA Pro [reported by authors]
glibc 1
ctags 1
fontconfig 1
wavpack 1
privoxy 1 2 3
radare2 1 2
X.Org 1 2

Moral of the story

- If you ever produce code that handles some non-trivial input format, run a tool like `afl` to look for bugs

Conclusions

- Fuzzing is a great technique to find an interesting class of security flaws!
  - Eg fuzzing GSM packets shows how crappy the software implementing the GSM protocol stack in phones is
- The bottleneck: how to do smart fuzzing without too much effort

  Successful approaches include

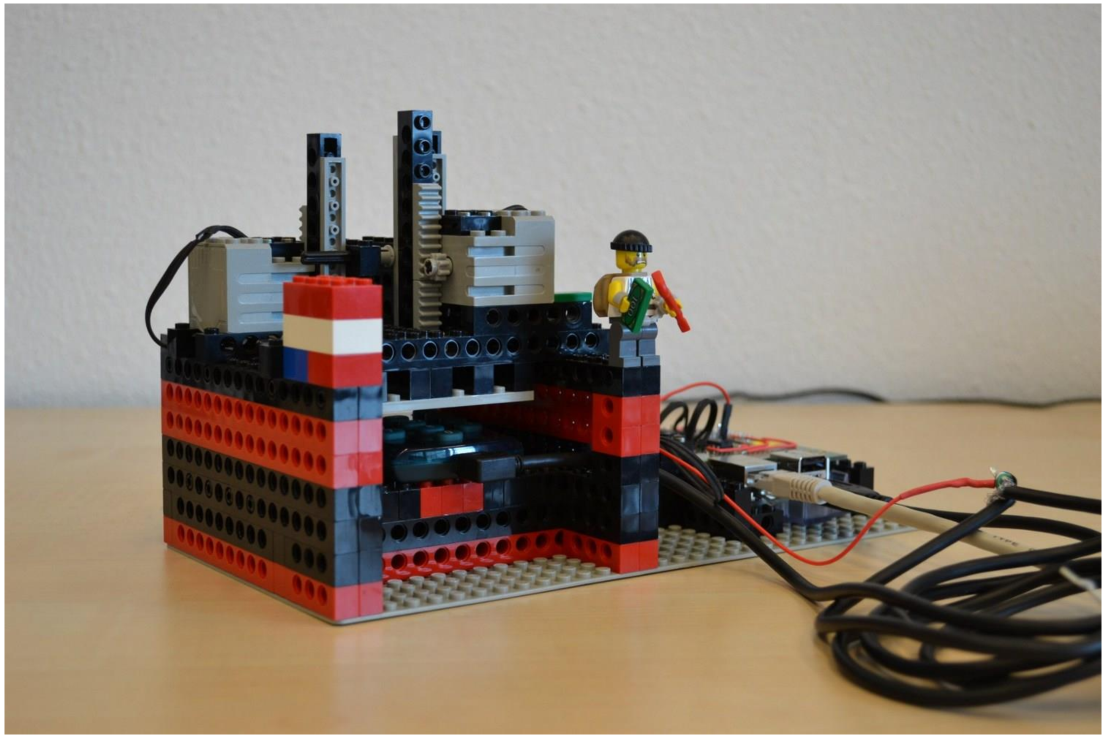
  - White-box fuzzing based on symbolic execution with SAGE
  - Evolutionary mutation-based fuzzing with `afl`
- A newer generation of tools not only tries to find security flaws, but also to then build exploits for them, eg. `angr`

To read (see links on the course page)

- David Wheeler, The Apple goto fail vulnerability: lessons learned
- Patrice Godefroid et al., SAGE: whitebox fuzzing for security testing, ACM Queue

Next week

- Preventing all these input flaws in the first place, using the LangSec approach
- More automated fuzzing, using state machines as formalism



http://tinyurl.com/legolearning