Security Testing

TDDC90 – Software Security

Ulf Kargén

Department of Computer and Information Science (IDA)

Division for Database and Information Techniques (ADIT)

LiU EXPANDING REALITY

Security testing vs “regular” testing

- “Regular” testing aims to ensure that the program meets customer requirements in terms of features and functionality.
- Tests “normal” use cases
  - ⇨ Test with regards to common expected usage patterns.

- Security testing aims to ensure that program fulfills security requirements.
  - Often non-functional.
  - More interested in misuse cases
    - ⇨ Attackers taking advantage of “weird” corner cases.

Functional vs non-functional security requirements

- Functional requirements – *What* shall the software do?
- Non-functional requirements – *How* should it be done?

- Regular functional requirement example (Webmail system):
  It should be possible to use HTML formatting in e-mails

- Functional security requirement example:
  The system should check upon user registration that passwords are at least 8 characters long

- Non-functional security requirement example:
  All user input must be sanitized before being used in database queries

How would you write a unit test for this?

LiU EXPANDING REALITY

Common security testing approaches

Often difficult to craft e.g. unit tests from non-functional requirements

Two common approaches:

- Test for known vulnerability types
- Attempt directed or random search of program state space to uncover the “weird corner cases”
  - Remember: in security, there is no such thing as an unlikely or “far fetched” usage scenario – attackers are **actively** looking for unhandled corner cases

In today’s lecture:

- Penetration testing (briefly)
- Fuzz testing or “fuzzing”
- Concolic testing

Penetration testing

- Manually try to “break” software
- Relies on human intuition and experience.
- Typically involves looking for known common problems.
- Can uncover problems that are impossible or difficult to find using automated methods
  - …but results completely dependent on skill of tester!

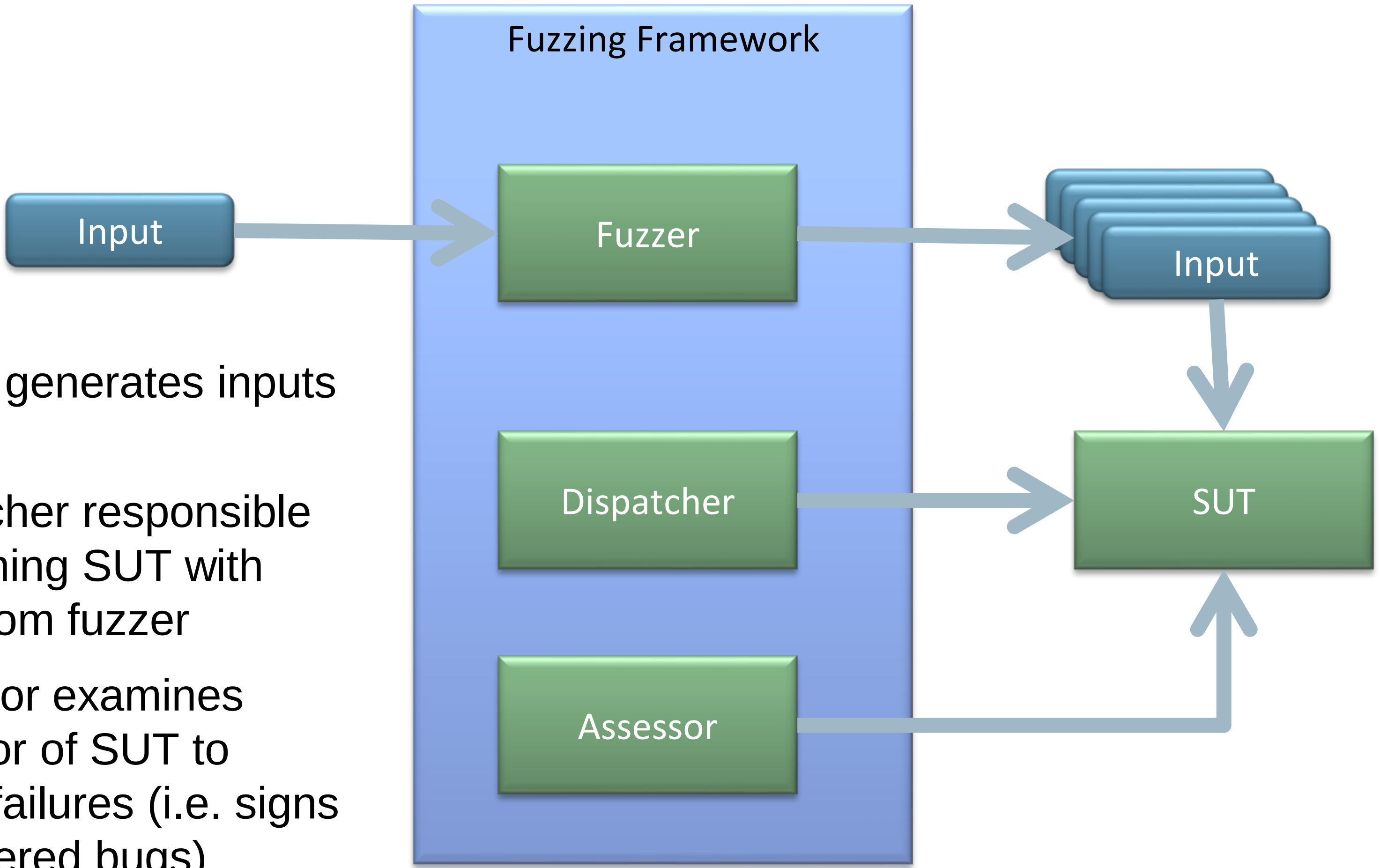
Fuzz testing

Idea: Send semi-valid input to a program and observe its behavior.

- Classical fuzzing is a black-box testing method – *System Under Test* (SUT) treated as a “black-box”
- The only feedback is the output and/or externally observable behavior of SUT.

- First proposed in a 1990 paper where completely random data was sent to 85 common Unix utilities in 6 different systems. 24 – 33% crashed.
  - Remember: Crash implies memory protection errors.
  - Crashes are often signs of exploitable flaws in the program!

Fuzz testing architecture



- Fuzzer generates inputs to SUT
- Dispatcher responsible for running SUT with input from fuzzer
- Assessor examines behavior of SUT to detect failures (i.e. signs of triggered bugs)

Fuzzing components: Input generation

Simplest method: Completely random

- Won’t work well in practice – Input deviates too much from expected format, rejected early in processing.

Two traditional approaches:

- Mutation based fuzzing
- Generation based fuzzing

Mutation based fuzzing

Start with a valid seed input, and “mutate” it.

- Flip some bits, change value of some bytes.
- Programs that have highly structured input, e.g. XML, may require “smarter” mutations.

Challenges:

- How to select appropriate seed input?
  - If official test suites are available, these can be used.
- How many mutations per input? What kind of mutations?

Generally mostly used for programs that take files as input.

- Trickier to do when interpretation of inputs depends on program state, e.g. network protocol parsers. (The way a message is handled depends on previous messages.)

Mutation based fuzzing – Pros and Cons

- ☺ Easy to get started, no (or little) knowledge of specific input format needed.
- ☹ Typically yields low code coverage, inputs tend to deviate too much from expected format – rejected by early sanity checks.

```
int parse_input(char* data, size_t size)
{
   int saved_checksum, computed_checksum;

   if(size < 4) return ERR_CODE;

   // First four bytes of ‘data’ is CRC32 checksum
   saved_checksum = *((int*)data);

   // Compute checksum for rest of ‘data’
   computed_checksum = CRC32(data + 4, size – 4);

   // Error if checksums don’t match
   if(computed_checksum != saved_checksum)
       return ERR_CODE;

   // Continue processing of ‘data’
   ...
```

Mutated inputs will always be rejected here!

Mutation based fuzzing – Pros and Cons

- ☺ Easy to get started, no (or little) knowledge of specific input format needed.
- ☹ Typically yields low code coverage, inputs tend to deviate too much from expected format – rejected by early sanity checks.
- ☹ Hard to reach "deeper" parts of programs by random guessing

```
int parse_record(char* data, int type)
{
   switch(type) {
      case 1234:
         parse_type_A(data);
         break;

      case 5678:
         parse_type_B(data);
         break;

      case 9101:
         parse_type_C(data);
         break;
   ...
```

*Very unlikely to guess "**magic constants**" correctly.*

If seed only contains Type A records, `parse_type_B` *will likely never be tested.*

Generation based fuzzing

Idea: Use a **specification** of the input format (e.g. a grammar) to automatically generate semi-valid inputs

Usually combined with various **fuzzing heuristics** that are known to trigger certain vulnerability types.

- Very long strings, empty strings
- Strings with format specifiers, “extreme” format strings
  - %n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
  - %s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
  - %5000000.x
- Very large or small values, values close to max or min for data type
  0x0, 0xffffffff, 0x7fffffff, 0x80000000, 0xfffffffe
- Negative values where positive ones are expected

Generation based fuzzing – Pros and Cons

- ☺ Input is much closer to the expected, much better coverage
- ☺ Can include models of protocol state machines to send messages in the sequence expected by SUT.

- ☹ Requires input format to be known.
- ☹ May take considerable time to write the input format grammar/specification.

Fuzzing components: The Dispatcher

Responsible for running the SUT on each input generated by fuzzer module.

- Must provide suitable environment for SUT.
  - E.g. implement a “client” to communicate with a SUT using the fuzzed network protocol.
- SUT may modify environment (file system, etc.)
  - Some fuzzing frameworks allow running SUT inside a virtual machine and restoring from known good snapshot after each SUT execution.

Fuzzing components: The Dispatcher

Another approach is *in-memory fuzzing*

- Fuzz some interface repeatedly *within one process* – avoids (often very significant) overhead from restarting the process for each input
- One popular example of this is libFuzzer (built into LLVM/clang compiler)

Example: We want to stress test the functionality of the function `parse_input` for potential security bugs using libFuzzer.

- Instead of re-running the program each time with an input file (that will eventually be parsed by `parse_input`), we write a fuzzer driver stub:

```
#include ”parser.h” // Declares parse_input

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
   parse_input(Data, Size);
   return 0;
}
```

- Compile with `clang -fsanitize=fuzzer driver.c -o fuzzer` to have clang *generate* a fuzzer that repeatedly calls `parse_input` with mutated inputs
- Run `./fuzzer seeds_dir/` to start fuzzing
- Note: Important that `parse_input` does not modify global state!

LiU EXPANDING REALITY

Fuzzing components: The Assessor

Must automatically assess observed SUT behavior to determine if a fault was triggered.

- For C/C++ programs: Monitor for memory access violations, e.g. out-of-bounds reads or writes.
- Simplest method: Just check if SUT crashed.
- Problem: SUT may catch signals/exceptions to gracefully handle e.g. segmentation faults
  - ⇨ Difficult to tell if a fault, (which could have been exploitable with carefully crafted input), have occurred

Improving fault detection

One solution is to attach a programmable debugger to SUT.

- Can catch signals/exceptions prior to being delivered to application.
- Can also help in manual diagnosis of detected faults by recording stack traces, values of registers, etc.

However: All faults do not result in failures, i.e. a crash or other observable behavior (e.g. Heartbleed).

- An out-of-bounds read/write or use-after-free may e.g. not result in a memory access violation.
- Solution: Use a dynamic-analysis tool that can monitor what goes on “under the hood”
  - Can potentially catch more bugs, but SUT runs slower.
  - ⇨ Need more time for achieving the same level of coverage

Memory error checkers (“sanitizers”)

AddressSanitizer (built into gcc/clang: `-fsanitize=address`)

- Applies instrumentation during compilation: Additional code is inserted in program to check for memory errors.
  - Incurs around 2x overhead, but significantly increases chance of detecting triggered bugs
- Monitors all calls to malloc/new/free/delete – can detect if memory is freed twice, used after free, out of bounds access of heap allocated memory, etc.
- Inserts checks that stack buffers are not accessed out of bounds
- Helps fuzzer to spot bugs that wouldn’t result in crash – also a nice debugging aid!

Plethora of other sanitizers have been proposed, following success of AddressSanitizer

- Focus on different kinds of memory errors:
  - ThreadSanitizer – checks for data race errors
  - LeakSanitizer – checks for memory leaks
  - UndefinedBehaviorSanitizer – checks for many causes of undefined behavior (array index out of bounds, integer overflows, sign errors, etc.)

Fuzzing web applications

- Errors are often on a higher abstraction level compared to e.g. simple coding errors leading to buffer overflows
- Assessing results is harder – program doesn’t just crash when something “bad” happens.
- Determining inputs can be hard
  - Much wider set of input vectors

Web application input vectors

- Inputs much more diverse than e.g. a file or a TCP connection:
  - Form fields (name-value pairs and hidden fields)
  - URL (names of pages and directories)
  - HTTP headers
  - Cookies
  - Uploaded files
  - AJAX
  - …

- How to identify inputs?
  - Manually
  - Using an automated crawler – tries to visit all links and “map out” the web application

Web application input generation

- How to fuzz?
  - Semi-random input generation doesn't work here – need more structured input
  - Web app fuzzers generally focus on the "*test for known vulnerability types*" approach rather than state-space exploration

  - XSS: Try including script tags
  - SQL injection: Try supplying SQL-injection attack strings
  - Try accessing resources that should require authentication
  - …

Assessing results of web app fuzzing

Automatically detecting faults generally much harder than for native programs running on the CPU!

- Often requires abstract “understanding” of what is correct behavior
  - For example: Detecting CSRF errors requires knowledge of what resources are supposed to be protected

- Difficult to cut the human out of the loop completely!

Assessing results of web app fuzzing

Typical things to look for:

- Server errors
  - Error messages (requires ad-hoc parsing of responses)
  - Error codes (5xx errors, etc.)
- Signs of input (e.g. JavaScript) in responses
  - Possible for reflected XSS
  - Much harder for stored XSS!
- Error messages or SQL output indicating SQL-injection problems

Recent advancements in fuzzing

Limitations of black-box fuzz testing

- Many programs have an infinite input space and state space - Combinatorial explosion!
- Conceptually a simple idea, but many subtle practical challenges
- Difficult to create a truly generic fuzzing framework that can cater for all possible input formats.
  - For best results often necessary to write a custom fuzzer for each particular SUT.
- (Semi)randomly generated inputs are very unlikely to trigger certain faults.

Limitations of fuzz testing
Example from first lecture on vulnerabilities

```
char buffer[100];
if(strlen(input) > 100)
{
    printf(“String too long!”);
    exit(1);
}
strcpy(buffer, input);
```

The off-by-one error will only be detected if `strlen(input) == 100`

Very unlikely to trigger this bug using black-box fuzz testing!

LiU EXPANDING REALITY

Modern fuzzers

- Mutation-based fuzzing can typically only find the “low-hanging fruit” – shallow bugs that are easy to find
- Generation-based fuzzers almost invariably gives better coverage, but requires much more manual effort

- More modern fuzzers improve coverage by “looking under the hood” of programs – i.e., no longer black-box.
  - **Evolutionary/Greybox fuzzing** combines mutation with an evolutionary algorithm to gradually uncover program state space. Popularized by the extremely successful fuzzer “American Fuzzy Lop” (AFL).
  - **Concolic execution** generates test cases based on the control-flow structure of the SUT. Our next topic…

Symbolic execution for program testing

Recall symbolic execution:

- Encode a program path as a query to a SAT/SMT solver
- Have the solver find satisfying assignments

In the lab you manually encoded paths as queries to a SMT solver.

- Of course also possible to perform this encoding automatically.

Concolic testing

Idea: Combine concrete and symbolic execution

- Concolic execution (CONCrete and symbOLIC)

Concolic execution workflow:

1. Execute the program for real on some input, and record path taken.
2. Encode path as query to SMT solver and negate one branch condition
3. Ask the solver to find new satisfying input that will give a different path

Reported bugs are always accompanied by an input that triggers the bug (generated by SMT solver)

⇨ Complete – Reported bugs are always real bugs

Can handle “magic constants” and complex constraints that are unlikely to be solved by random mutation

Concolic testing – small example

```
void weather(int temperature, unsigned int precipitation)
{
  char forecast[5];

  if(temperature > 0) {
    if(precipitation > 0)
      strcpy(forecast, “rain”);
    else
      strcpy(forecast, “nice”);
  } else {
    if(precipitation == 0)
      strcpy(forecast, “cold”);
    else if(precipitation > 20)
      strcpy(forecast, “blizzard”);
    else
      strcpy(forecast, “snow”);
  }

  ...
```

Buffer overflow!

LiU EXPANDING REALITY

Concolic testing – small example

```
void weather(int temperature, unsigned int precipitation)
{
  char forecast[5];

  if(temperature > 0) {
    if(precipitation > 0)
      strcpy(forecast, “rain”);
    else
      strcpy(forecast, “nice”);
  } else {
    if(precipitation == 0)
      strcpy(forecast, “cold”);
    else if(precipitation > 20)
      strcpy(forecast, “blizzard”);
    else
      strcpy(forecast, “snow”);
  }

  ...
```

First round:

Concrete inputs (arbitrary):
temperature=1, precipitation=1

Recorded path constraints:
$temperature > 0 \land precipitation > 0$

New path constraint:
$temperature > 0 \land \neg (precipitation > 0)$

Solution from solver:
temperature=1, precipitation=0

Concolic testing – small example

```
void weather(int temperature, unsigned int precipitation)
{
  char forecast[5];

  if(temperature > 0) {
    if(precipitation > 0)
      strcpy(forecast, “rain”);
    else
      strcpy(forecast, “nice”);
  } else {
    if(precipitation == 0)
      strcpy(forecast, “cold”);
    else if(precipitation > 20)
      strcpy(forecast, “blizzard”);

                       );
```

Second round:

Concrete inputs:

temperature=1, precipitation=0

Recorded path constraints:

temperature > 0 ∧ ¬ (precipitation > 0)

New path constraint:

¬ (temperature > 0)

Solution from solver:

temperature=0, precipitation=0

Note: ‘precipitation ’ is unconstrained – no need to care about later branch conditions when negating an earlier condition.

LiU EXPANDING REALITY

Concolic testing – small example

```
void weather(int temperature, unsigned int precipitation)
{
  char forecast[5];

  if(temperature > 0) {
    if(precipitation > 0)
      strcpy(forecast, "rain");
    else
      strcpy(forecast, "nice");
  } else {
    if(precipitation == 0)
      strcpy(forecast, "cold");
    else if(precipitation > 20)
      strcpy(forecast, "blizzard");
    else
      strcpy(forecast, "snow");
  }

  ...
```

Third round:

Concrete inputs:

temperature=0, precipitation=0

Recorded path constraints:

$\neg$ (temperature > 0) $\wedge$ precipitation = 0

New path constraint:

$\neg$ (temperature > 0) $\wedge$ $\neg$(precipitation = 0)

Solution from solver:

temperature=0, precipitation=1

Concolic testing – small example

```
void weather(int temperature, unsigned int precipitation)
{
  char forecast[5];

  if(temperature > 0) {
    if(precipitation > 0)
      strcpy(forecast, “rain”);
    else
      strcpy(forecast, “nice”);
  } else {
    if(precipitation == 0)
      strcpy(forecast, “cold”);
    else if(precipitation > 20)
      strcpy(forecast, “blizzard”);
    else
      strcpy(forecast, “snow”);
  }

  ...
```

Fourth round:

Concrete inputs:

temperature=0, precipitation=1

Recorded path constraints:

$\neg$ (temperature > 0) $\wedge$ $\neg$(precipitation = 0) $\wedge$ $\neg$(precipitation > 20)

New path constraint:

$\neg$ (temperature > 0) $\wedge$ $\neg$(precipitation = 0) $\wedge$ precipitation > 20

Solution from solver:

temperature=0, precipitation=21

Concolic testing – small example

```
void weather(int temperature, unsigned int precipitation)
{
  char forecast[5];

  if(temperature > 0) {
    if(precipitation > 0)
      strcpy(forecast, “rain”);
    else
      strcpy(forecast, “nice”);
  } else {
    if(precipitation == 0)
      strcpy(forecast, “cold”);
    else if(precipitation > 20)
      strcpy(forecast, “blizzard”);
    else
      strcpy(forecast, “snow”);
  }

  ...
```

Fifth round:

Concrete inputs:

temperature=0, precipitation=21

Recorded path constraints:

$\neg$ (temperature > 0) $\wedge$ $\neg$(precipitation = 0) $\wedge$ precipitation > 20

Bug found!

Challenges with concolic testing: Path explosion

Number of paths increase exponentially with number of branches

- Most real-world programs have an infinite state space!
  - For example, number of loop iterations may depend on size of input

Not possible to explore all paths:

- Depth-first search (as in the previous example) will easily get “stuck” in one part of the program
  - May e.g. keep exploring the same loop with more and more iterations
- Breadth-first search will take a very long time to reach “deep” states
  - May take “forever” to reach the buggy code

Challenges with concolic testing: Path explosion

Try “smarter” ways of exploring the program state space

- May want to try to run loops many times to uncover possible buffer overflows
- …but also want to maximize coverage of different parts of the program

For example, the Microsoft SAGE system implements “whitebox fuzzing”

- Performs concolic testing, but prioritizes paths based on how much they improve coverage
- Results can be assessed similar to black-box fuzzing (with dynamic analysis tools, etc.)

Rationale for "whitebox fuzzing"

Has proven to work well in practice

- Used in production at Microsoft to test e.g. Windows, Office, etc. prior to release
    - Has uncovered many serious vulnerabilities that was missed by other approaches (black-box fuzzing, static analysis, etc.)

Interestingly, SAGE works directly at the machine-code level

- Note: Source code not needed for concolic execution – sufficient to collect constraints from one concrete sequence of machine-code instructions.
- Avoids hassle with different build environments, third-party libraries, programs written in different languages etc.
- …but sacrifices some coverage due to additional approximations needed when working on machine code

Limitations of concolic testing

- The success of concolic testing is due to the massive improvement in SAT/SMT solvers during the last two decades.
  - Main bottleneck is still often the solvers.
  - Black-box fuzzing can perform a much larger number of test cases per time unit – may be more time efficient for “shallow” bugs.

- Solving SAT/SMT problems is NP-complete.
  - Solvers like e.g. Z3 use various “tricks” to speed up common cases
  - …but may take unacceptably long time to solve certain path constraints.

Limitations of concolic testing

If program uses any kind of cryptography, symbolic execution will typically fail.

- Consider previous checksum example:
- CRC32 is linear and reversible – solver can “repair” checksum if rest of data is modified.

```
...
   // Compute checksum for rest of ‘data’
   computed_checksum = CRC32(data + 4, size – 4);

   // Error if checksums don’t match
   if(computed_checksum != saved_checksum)
       return ERR_CODE;
...
```

What if program used e.g. SHA256 here instead?

Solver would get ”stuck” trying to solve this constraint!

Generation-based fuzzing could handle this without problem!

Greybox fuzzing

- Probability of hitting a “deep” level of the code decreases exponentially with the “depth” of the code for mutation based fuzzing
  - Number of trials before we solve a branch constraint by random guessing increases exponentially with depth

- Similarly, the time required for solving an SMT query is high, and increases exponentially with the depth of the path constraint.

- Black-box fuzzing is clearly too “dumb”, but concolic execution turns out to be overkill for many constraints in real-life code
  - Idea of *greybox fuzzing* is to find a sweet spot in between.

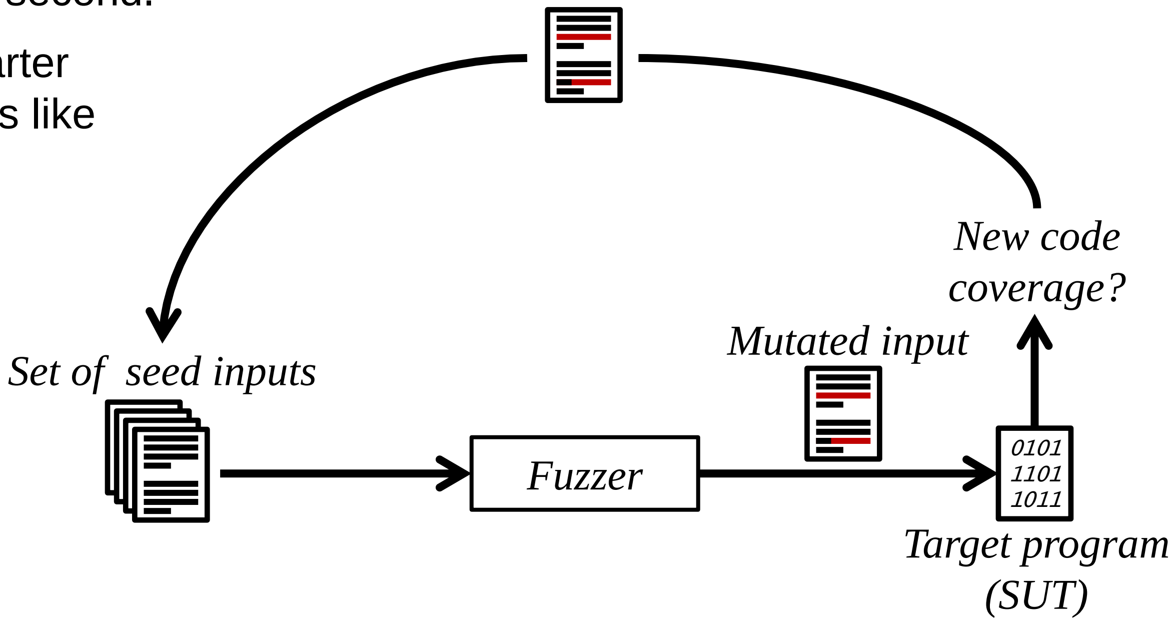
Greybox fuzzing

- Instead of recording full path constraint (as in concolic execution), record light-weight *coverage information* to guide fuzzing.
- Use evolutionary algorithm to incrementally solve nested constraints
  - In a sense, fuzzer “learns” input format by trial and error
- American Fuzzy Lop (AFL) was the first successful implementation of this principle – have found thousands of serious vulnerabilities in open-source programs
  - Many recent fuzzers based on same principle
    - AFL++ (AFL fork with many advanced features)
    - Honggfuzz
    - libFuzzer
    - + huge number of academic/experimental fuzzers
- *Greybox fuzzing is the de-facto fuzzing technique used in industry today*

Greybox fuzzing

Performs “regular” mutation-based fuzzing (using several different strategies) and measures code coverage.

- **Every** generated input **that resulted in any new coverage** is saved and later re-fuzzed
  - This extremely simple evolutionary algorithm allows to gradually “learn” how to reach deeper parts of the program.
- Coverage instrumentation highly optimized for speed – can reach several thousand test cases per second.
  - This often beats smarter (and slower) methods like concolic execution!



LiU EXPANDING REALITY

Greybox fuzzing example

```
void foo(char[3] data)
{
  if(data[0] == 'b')
    if(data[1] == 'a')
      if(data[2] == 'd')
        bug();
}
```

Random input-generation without feedback would require on average *~16 million attempts* to find bug (256^3)

Greybox fuzzing example

```
void foo(char[3] data)
{
  if(data[0] == 'b')
    if(data[1] == 'a')
      if(data[2] == 'd')
        bug();
}
```

Greybox fuzzing:

Start with random seed input “xgc”

Greybox fuzzing example

```
void foo(char[3] data)
{
  if(data[0] == 'b')
    if(data[1] == 'a')
      if(data[2] == 'd')
        bug();
}
```

Greybox fuzzing:

Start with random seed input “xgc”

Covers first “if” only

Input corpus:

“xgc”

Greybox fuzzing example

```
void foo(char[3] data)
{
  if(data[0] == 'b')
    if(data[1] == 'a')
      if(data[2] == 'd')
        bug();
}
```

Randomly mutate one byte of input.

After (on average) 3x256 runs, second "if" is covered

New coverage → Mutated input is saved

Input corpus:

"xgc"

Runs:

3x256

LiU EXPANDING REALITY

Greybox fuzzing example

```
void foo(char[3] data)
{
  if(data[0] == 'b')
    if(data[1] == 'a')
      if(data[2] == 'd')
        bug();
}
```

For *each* input in corpus:
Randomly mutate one byte of input.

After (on average) 2x3x256 runs, third "if" is covered

New coverage → Mutated input is saved

Input corpus:	**Runs:**
"xgc"	3x256 +
"bgc"	2x3x256

LiU EXPANDING REALITY

Greybox fuzzing example

```
void foo(char[3] data)
{
  if(data[0] == 'b')
    if(data[1] == 'a')
      if(data[2] == 'd')
        bug();
}
```

For *each* input in corpus:
Randomly mutate one byte of input.

After (on average) 3x3x256 runs, bug is found!

Input corpus:	**Runs:**
“xgc”	3x256 +
“bgc”	2x3x256 +
“bac”	3x3x256

LiU EXPANDING REALITY

Greybox fuzzing example

```
void foo(char[3] data)
{
  if(data[0] == 'b')
    if(data[1] == 'a')
      if(data[2] == 'd')
        bug();
}
```

About 4000 times faster than random fuzzing without feedback!

- Enormous time saving even with 2-10x slowdown from coverage instrumentation

Input corpus:	**Runs:**
“xgc”	3x256 +
“bgc”	2x3x256 +
“bac”	3x3x256 =
“bad”	**4608**

LiU EXPANDING REALITY

Greybox fuzzing evolution

Success of greybox fuzzing is due in large part to implementation-level optimizations that allows extremely fast input generation

- In-memory fuzzing
- "Persistent-mode" fuzzing (similar to in-memory, but inputs are generated by separate fuzzer process)
- Using a *fork server* – makes clever use of Unix fork() system call to allow fast SUT restarts
- Highly optimized coverage instrumentation
- *cmplog* instrumentation – checks difference (e.g. Hamming distance) between operands in branch conditions that check for equivalence (for example, `if (a == b) {…}`)
  - Save inputs that get closer to solving equivalence check
- Attempt to optimize prioritization/scheduling of inputs in seed queue and choices of mutation operations
- etc. …

Fuzzing summary

Black-box fuzzing

- The only option if code instrumentation is not possible (e.g. when fuzzing embedded systems)
- Mutation
  - Semi-random input generation.
  - Good for quickly finding “low-hanging bugs”, but poor code coverage
- Generation
  - Generates inputs from given input specification/grammar
  - Very good coverage (if input specification is good)
  - Lots of work required for creating input specification

Fuzzing Summary

Concolic testing – White-box testing method.

- Input generated from control-structure of code to systematically explore different paths of the program.
  - Record executed path, negate branch constraint, solve to get new input, repeat…
- Can handle complex constraints
  (magic constants, simple checksums, etc.)
- Can reach close to 100% code coverage for smaller code bases, but...
- … scalability is a serious problem on large code bases

Fuzzing Summary

Greybox fuzzing

- Semi-random input generation like mutational fuzzing
- But uses coverage feedback to “incrementally” solve constraints and reach new code
  - Any mutated input that leads to new code coverage is saved and re-fuzzed
- Much lower input generation overhead than e.g. concolic testing
  - Often better at quickly finding bugs than “smarter” methods like concolic execution
- But shares some limitations with mutation-based fuzzing (e.g. magic constants, checksums)

- Research frontier:
  - Combine greybox fuzzing with concolic execution – solve “easy” constraints with fuzzing, handle “difficult” constraints with symbolic execution
  - More efficient (but approximate) methods than SMT solvers to solve branch constraints in concolic execution, e.g., gradient descent

Conclusions

- Fuzzing automatically generates inputs, either semi-randomly or from structure of code
  - Test cases not based on requirements
  - Pro: Not “biased” by developers’ view of “how things should work”, can uncover unsound assumptions or corner cases not covered by specification.
  - Con: Fuzzing and concolic testing mostly suited for finding *implementation* errors, e.g. buffer overflows, arithmetic overflows, etc.

- Generally hard to test for high-level errors in requirements and design using these methods

Conclusions

- Different methods are good at finding different kinds of bugs, but none is a silver bullet.
  - Fuzzing cannot be the only security assurance method used!
  - Static analysis
  - Manual reviews (code, design documents, etc.)
  - Regular unit/integration/system testing!
  - …

Conclusions

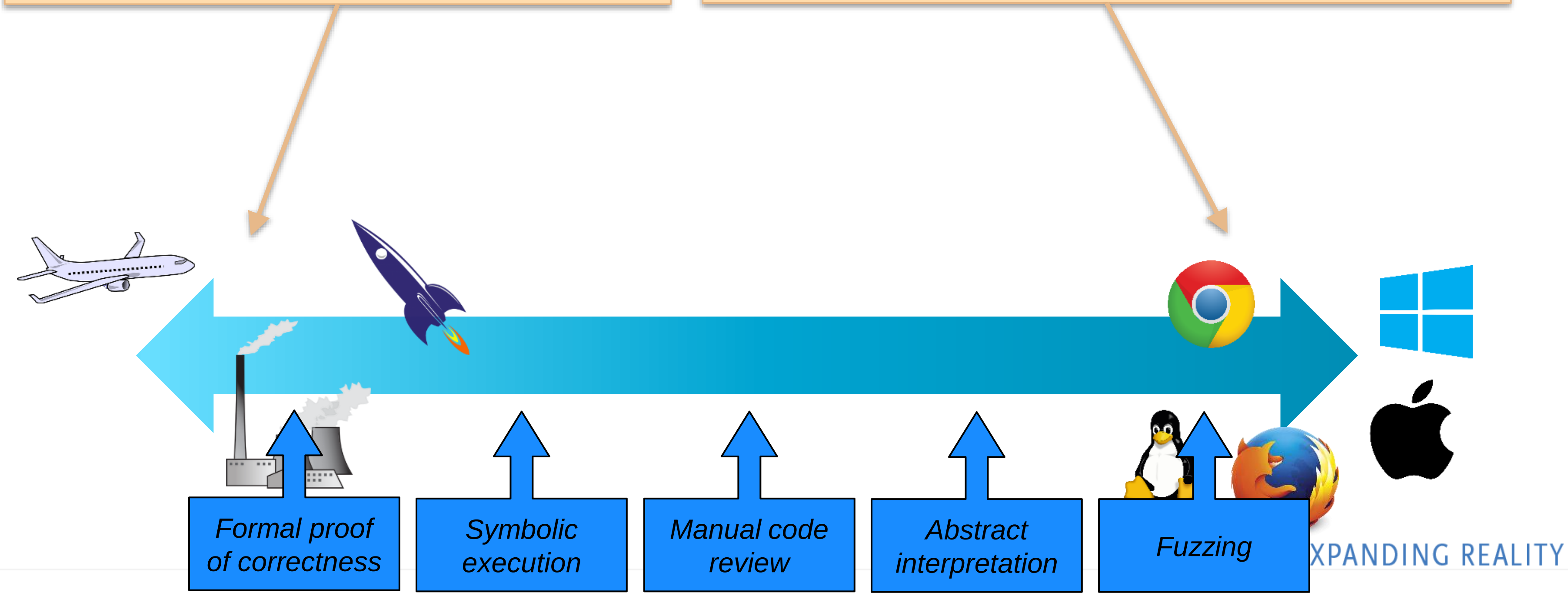
Software security assurance "spectrum":

Thousands of lines of code

- *Extremely high security/safety requirements*
- *Programs **must** be correct*

~10-100 millions of lines of code

- *Try to minimize number of bugs*
- *... but assume **code is not bug free***
- *Minimize impact of bugs: Defense in depth*


Formal proof of correctness
Symbolic execution
Manual code review
Abstract interpretation
Fuzzing