EXPRO / TEC-Standardisation
07/2022 – 07/2023

Applicability of Fuzz Testing to Space Software

A-Prof. Fabrizio Pastore
University of Luxembourg
fabrizio.pastore@uni.lu

ADCSS 2023 - November 15th, 2023

esa
European Space Agency

SnT

uni.lu
UNIVERSITY OF LUXEMBOURG

SnT Centre – University of Luxembourg

PEOPLE

480+
Workforce

70+
Nationalities

40%
Alumni who stay in Luxembourg

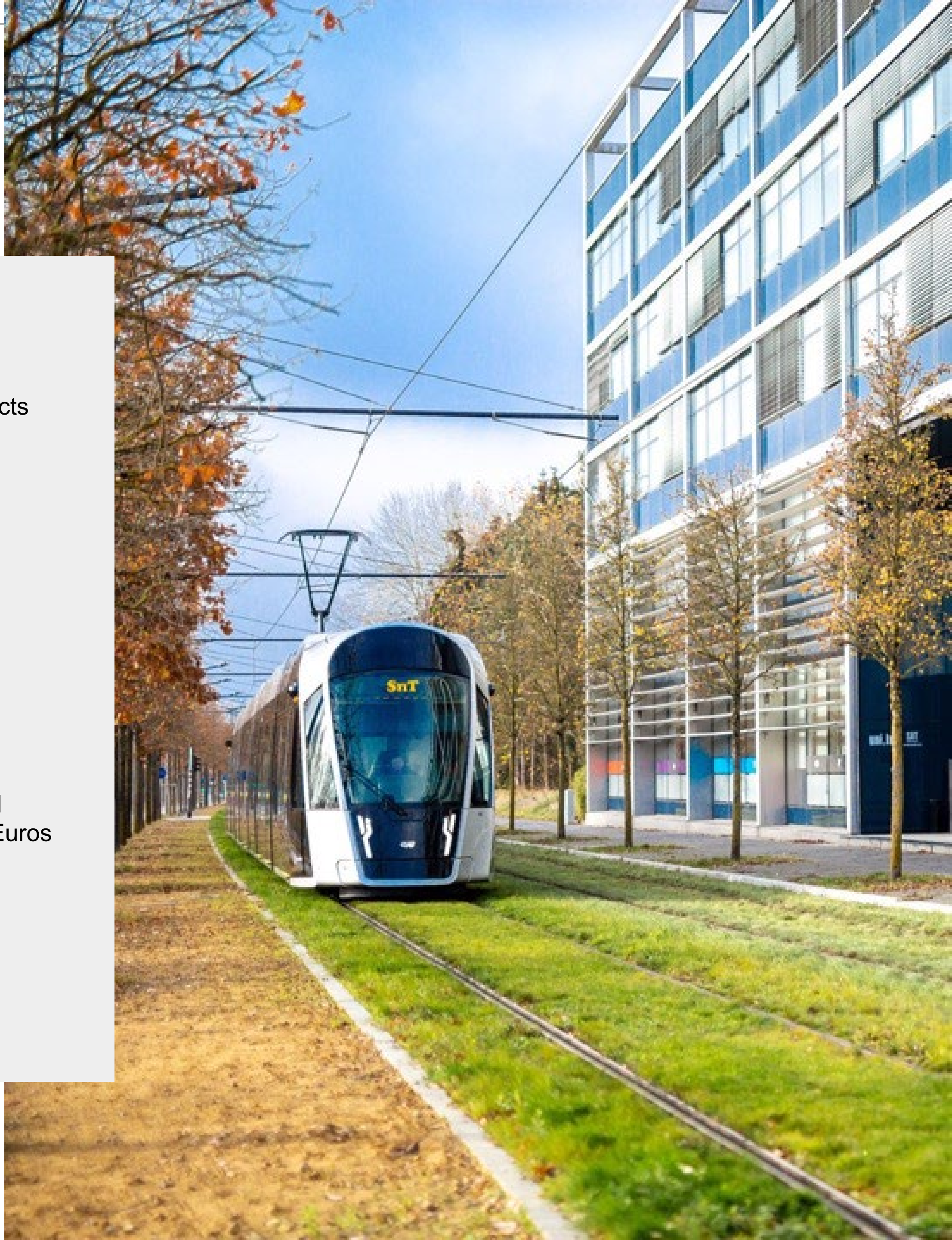
PARTNERSHIPS & INNOVATION

50%
Doctoral Candidates on Industrial Projects

+65
Partners

8M
Partners annual contribution in Euros

5
Spin-offs



Software Validation and Verification Group – www.SVV.lu

- Established in 2012
- Headed by Prof. Lionel Briand
- 24 members
  - 3 faculty members
  - 2 research scientists
  - 9 research associates
  - 7 PhD candidates
  - 3 research engineers

Research Topics

Requirements Quality Assurance
- Managing and Evolving Large Sets of NL Requirements
- Regulatory Compliance

Design-time Verification
- Simulation of CPS Models
- Schedulability Analysis

Testing & Analysis
- Model-Based and Regression Testing
- Functional Safety of AI-based systems
- Security Analysis and Testing

Run-time Verification
- Monitoring and Trace Checking
- Specification Languages
- Log Analysis & Anomaly Detection

SVV.lu
software verification & validation

EXPRO / TEC-Standardisation
07/2022 – 07/2023

Applicability of Fuzz Testing to Space Software

A-Prof. Fabrizio Pastore
University of Luxembourg
fabrizio.pastore@uni.lu

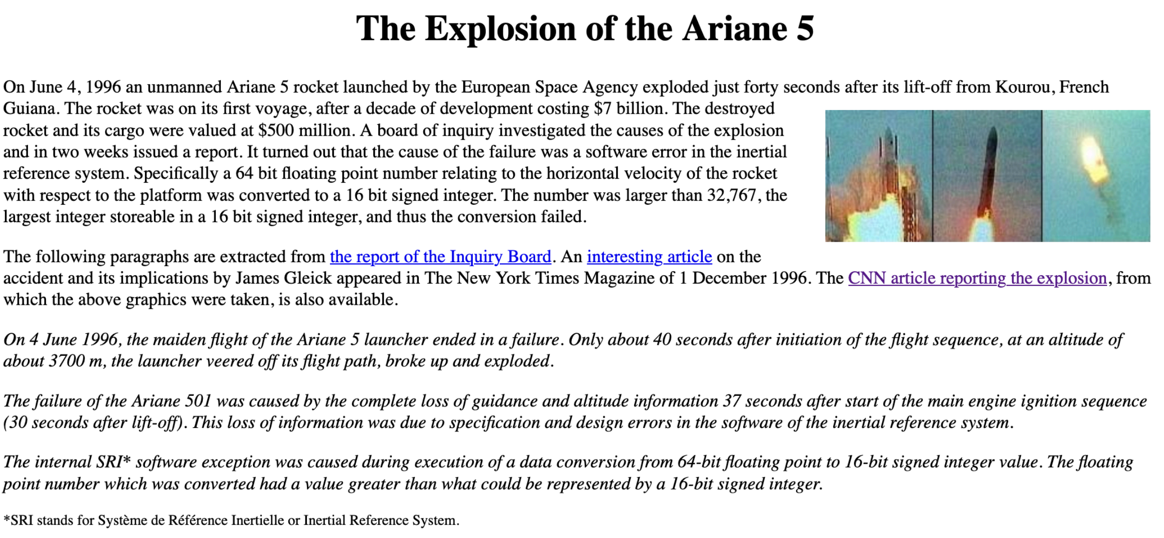
ADCSS 2023 - November 15th, 2023

European Space Agency

UNIVERSITY OF LUXEMBOURG

Software
has a prominent role
in Space Systems

Software failures have critical impact



The Explosion of the Ariane 5

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed.

The following paragraphs are extracted from the report of the Inquiry Board. An interesting article on the accident and its implications by James Gleick appeared in The New York Times Magazine of 1 December 1996. The CNN article reporting the explosion, from which the above graphics were taken, is also available.

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.

The failure of the Ariane 501 was caused by the complete loss of guidance and altitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system.

The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.*

*SRI stands for Système de Référence Inertielle or Inertial Reference System.

NASA: DOS Glitch Nearly Kil Rover

By Mark Hachman on August 23, 2004 at 7:07 pm Comments

STANFORD, CALIF. — A software glitch that paralyzed the Mars year was caused by an unanticipated characteristic of a DOS file said Monday.

The flaw, since fixed, was only discovered after days of agonizin complicated by the limited "windows" of communication allowe said Robert Denise, a member of the Flight Software Developme Propulsion Laboratory.

A technical glitch led to Israeli spacecraft crash, company says

By Nicole Chavez and Samantha Beech, CNN
Updated 0227 GMT (1027 HKT) April 13, 2019

Metric mishap caused loss of NASA orbiter

September 30, 1999
Web posted at: 4:21 p.m. EDT (2021 GMT)

In this story:

Metric system used by NASA for many years

Error points to nation's conversion lag

RELATED STORIES, SITES

NASA's Climate Orbiter was lost September 23, 1999

nature

COMPUTING

Software Error Doomed Japanese Hitomi Spacecraft

Space agency declares the astronomy satellite a loss

By Alexandra Witze, Nature magazine on April 29, 2016

The Washington Post
Democracy Dies in Darkness

Military Satellite in Wrong Orbit

By **William Harwood**
May 1, 1999

A $433 million Air Force rocket mysteriously misfired a half-hour after launch today, putting an $800 million military communications satellite into a useless orbit.

It was the third failure in three straight flights of the Lockheed Martin Titan IV rocket system and if the Milstar 2 satellite cannot be salvaged -- an option that does not immediately appear likely -- losses over the past nine months would total nearly $3 billion.

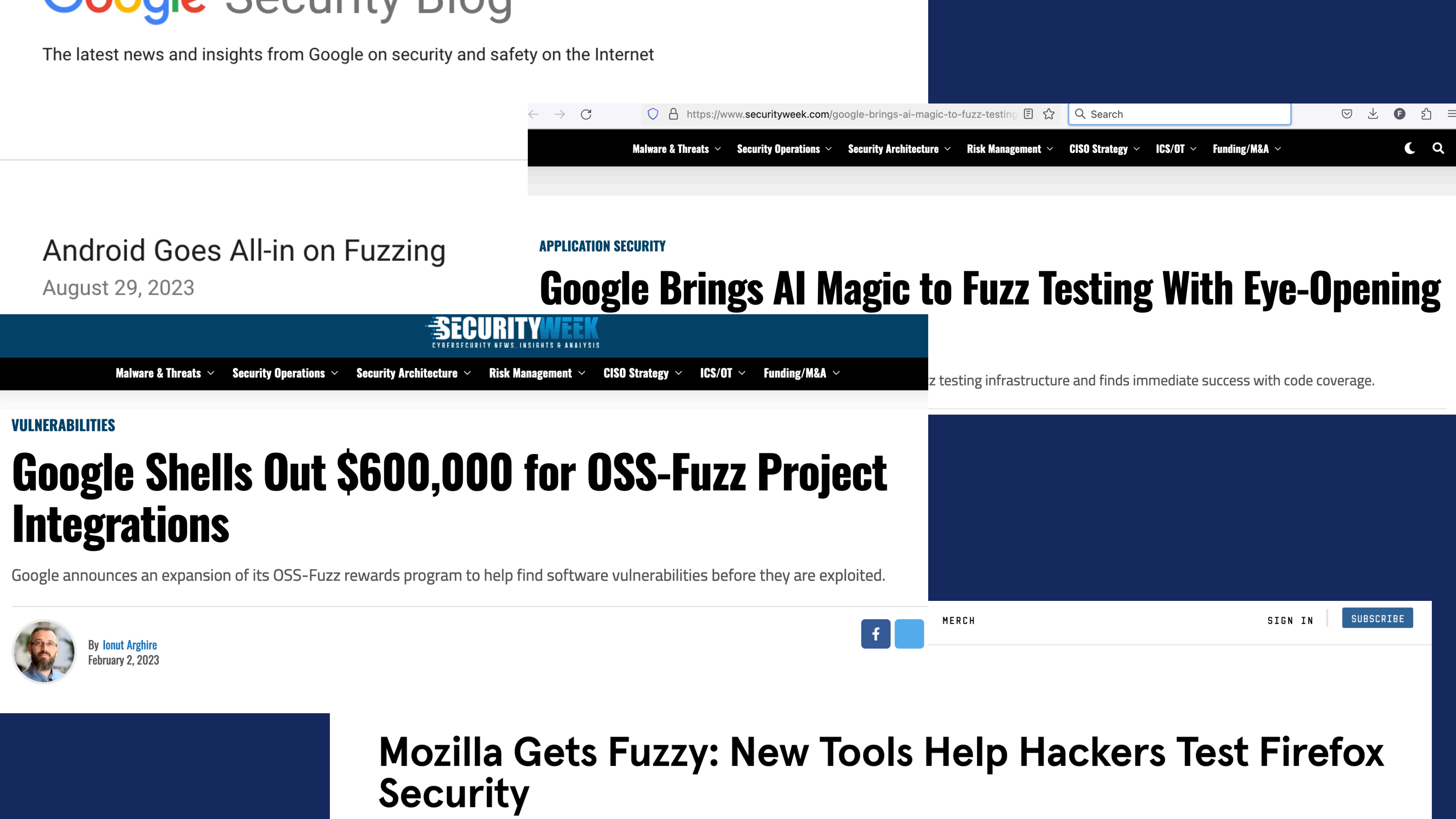
In light of the previous malfunctions, today's devastating $1.2 billion failure raised fresh questions abou the reliability of the nation's premier military launcher. The unmanned rocket system was developed in large part to give the Air Force "assured access to space" in the wake of the 1986 Challenger space shutt accident

How
space software is
verified and validated?

Software testing is prevalent in V&V

Word cloud for
ECSS-Q-ST-80C standard

Software testing
is expensive:
can we automate it?



Google Security Blog

The latest news and insights from Google on security and safety on the Internet

Android Goes All-in on Fuzzing

August 29, 2023

APPLICATION SECURITY

Google Brings AI Magic to Fuzz Testing With Eye-Opening

z testing infrastructure and finds immediate success with code coverage.

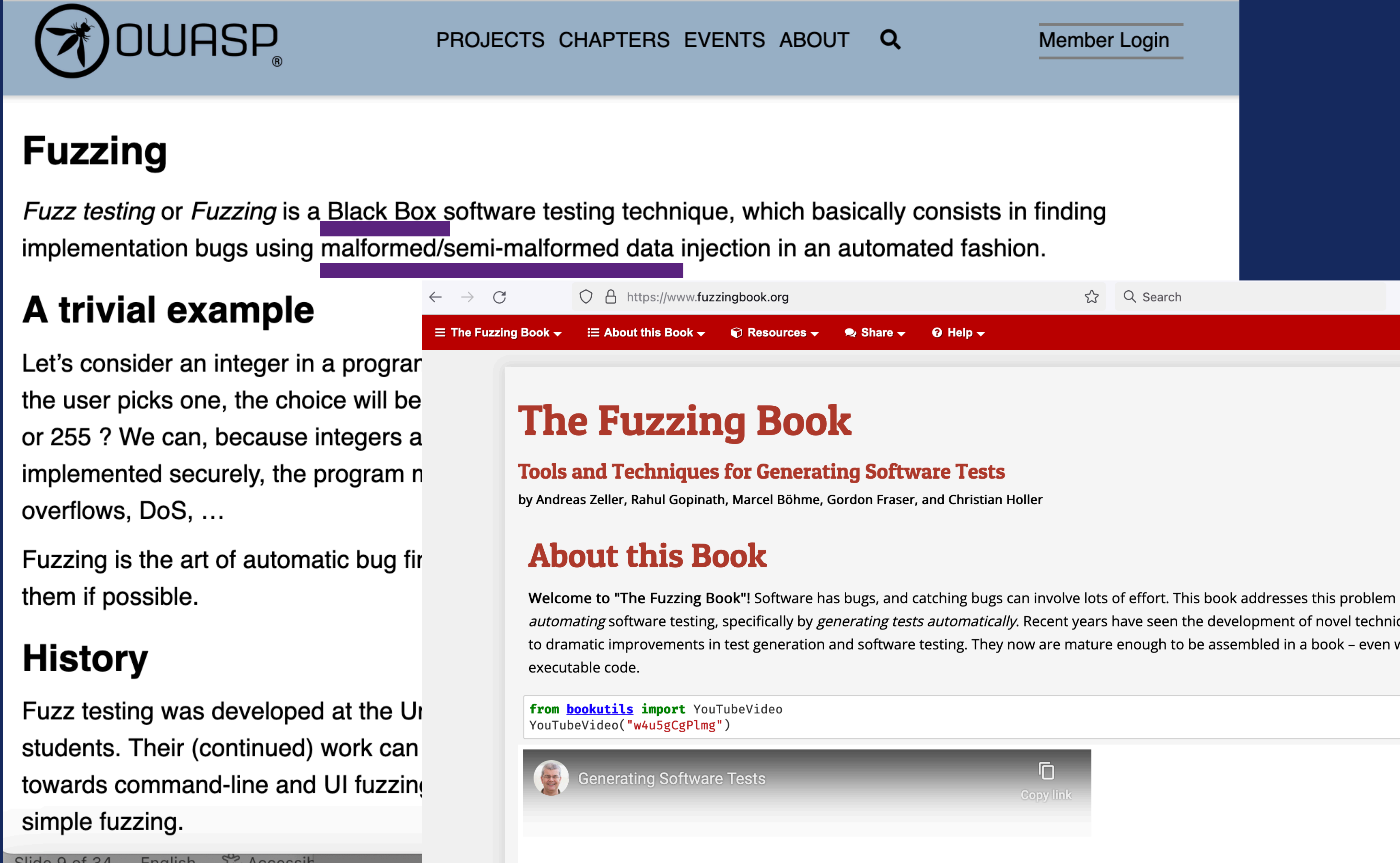
VULNERABILITIES

Google Shells Out $600,000 for OSS-Fuzz Project Integrations

Google announces an expansion of its OSS-Fuzz rewards program to help find software vulnerabilities before they are exploited.

By Ionut Arghire
February 2, 2023

Mozilla Gets Fuzzy: New Tools Help Hackers Test Firefox Security



Fuzzing

Fuzz testing or *Fuzzing* is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion.

A trivial example

Let's consider an integer in a progran
the user picks one, the choice will be
or 255 ? We can, because integers a
implemented securely, the program n
overflows, DoS, …

Fuzzing is the art of automatic bug fir
them if possible.

History

Fuzz testing was developed at the Ur
students. Their (continued) work can
towards command-line and UI fuzzin
simple fuzzing.

The Fuzzing Book

Tools and Techniques for Generating Software Tests

by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler

About this Book

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

```
from bookutils import YouTubeVideo
YouTubeVideo("w4u5gCgPlmg")
```

Different Fuzzing Techniques

Inputs structure and constraints

- model-based (grammar-based) approaches
- model-less approaches

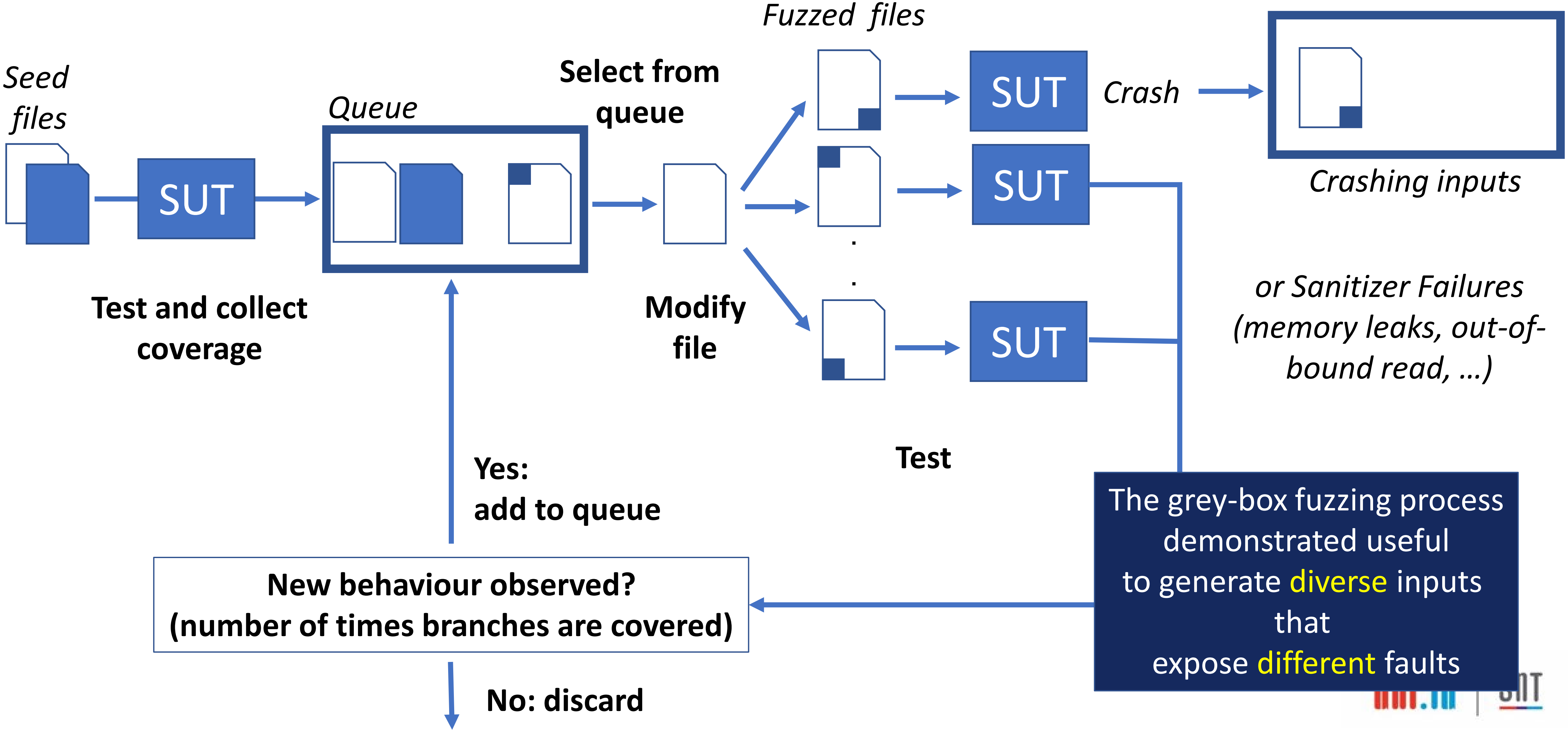
Test target

- Whole-program
- API functions

Internal program structure

- black-box approaches
  - ignore internal structure
- greybox approaches
  - rely on data that is easy to collect and process (e.g., code coverage)
- whitebox approaches
  - reason about the code semantics (e.g., symbolic execution)

Grey-box Fuzzing: An Evolutionary Testing Approach



Seed files

SUT

Test and collect coverage

Queue

Select from queue

Modify file

Fuzzed files

SUT

SUT

SUT

Crash

Crashing inputs

or Sanitizer Failures (memory leaks, out-of-bound read, …)

Test

Yes: add to queue

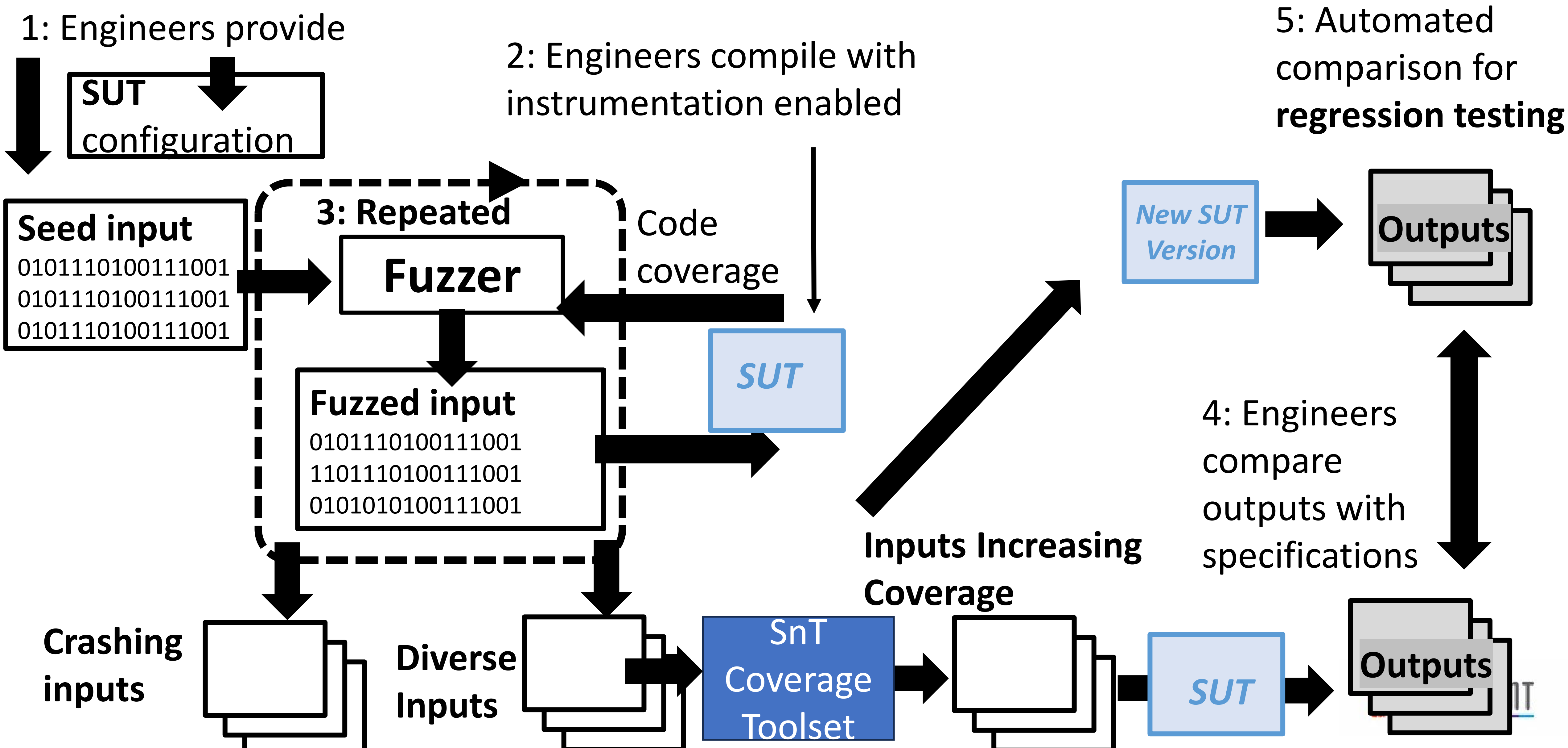
New behaviour observed?
(number of times branches are covered)

No: discard

The grey-box fuzzing process demonstrated useful to generate diverse inputs that expose different faults

Can we rely on fuzzing to automate functional testing?

Proposed Methodology: System-level Functional Testing



1: Engineers provide

SUT configuration

Seed input
0101110100111001
0101110100111001
0101110100111001

2: Engineers compile with instrumentation enabled

3: Repeated

Fuzzer

Fuzzed input
0101110100111001
1101110100111001
0101010100111001

Code coverage

SUT

Crashing inputs

Diverse Inputs

SnT Coverage Toolset

Inputs Increasing Coverage

New SUT Version

Outputs

5: Automated comparison for **regression testing**

4: Engineers compare outputs with specifications

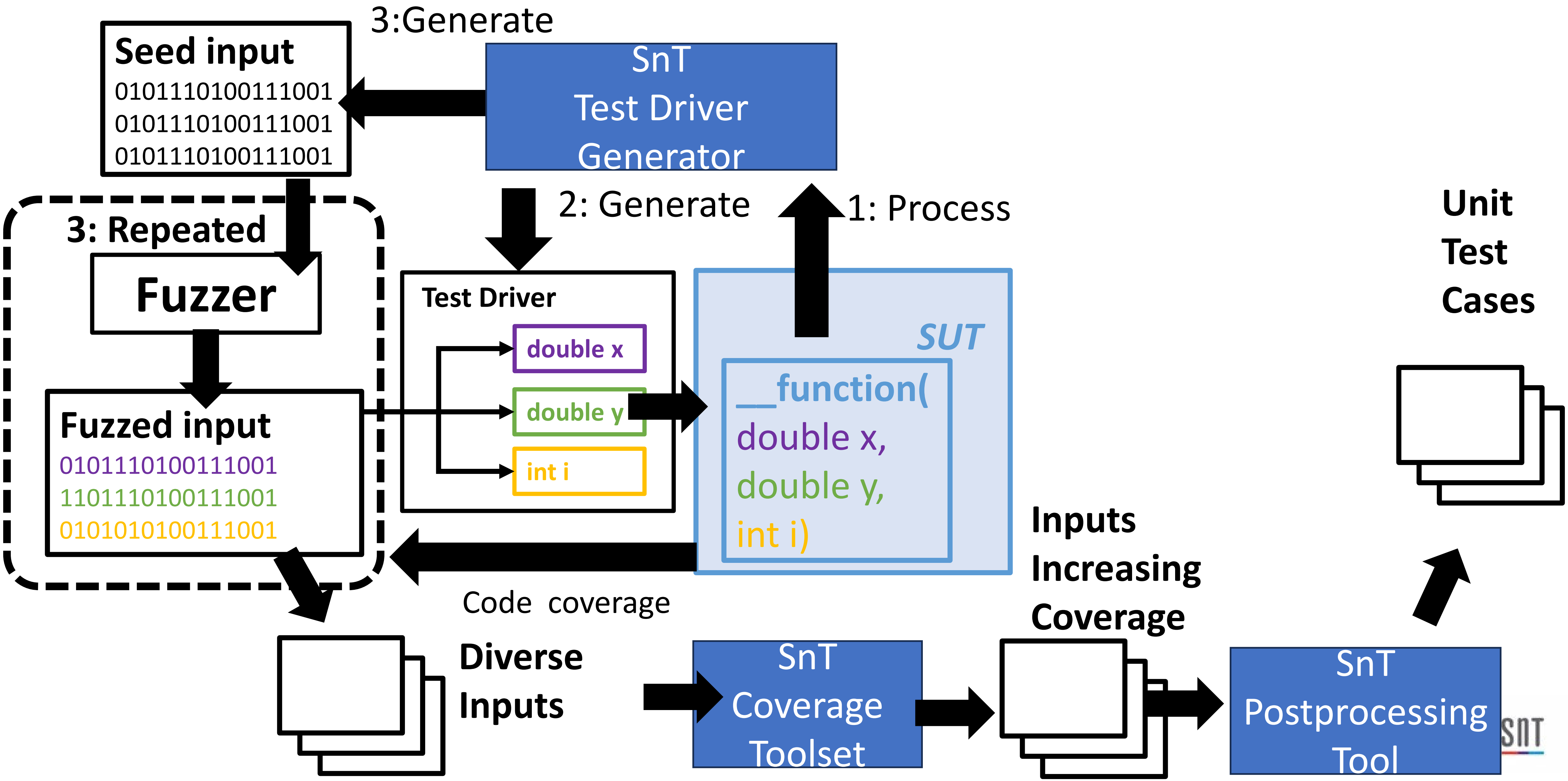
SUT

Outputs

Automated
system-level testing would
be beneficial,
but we should maximize
coverage with unit test
cases, first

Can we rely on fuzzing to generate unit test cases?

Proposed Methodology: Unit-level Functional Testing



3:Generate

SnT Test Driver Generator

Seed input
0101110100111001
0101110100111001
0101110100111001

2: Generate

1: Process

3: Repeated

Fuzzer

Fuzzed input
0101110100111001
1101110100111001
0101010100111001

Test Driver

double x
double y
int i

SUT

__function(
double x,
double y,
int i)

Code coverage

Diverse Inputs

SnT Coverage Toolset

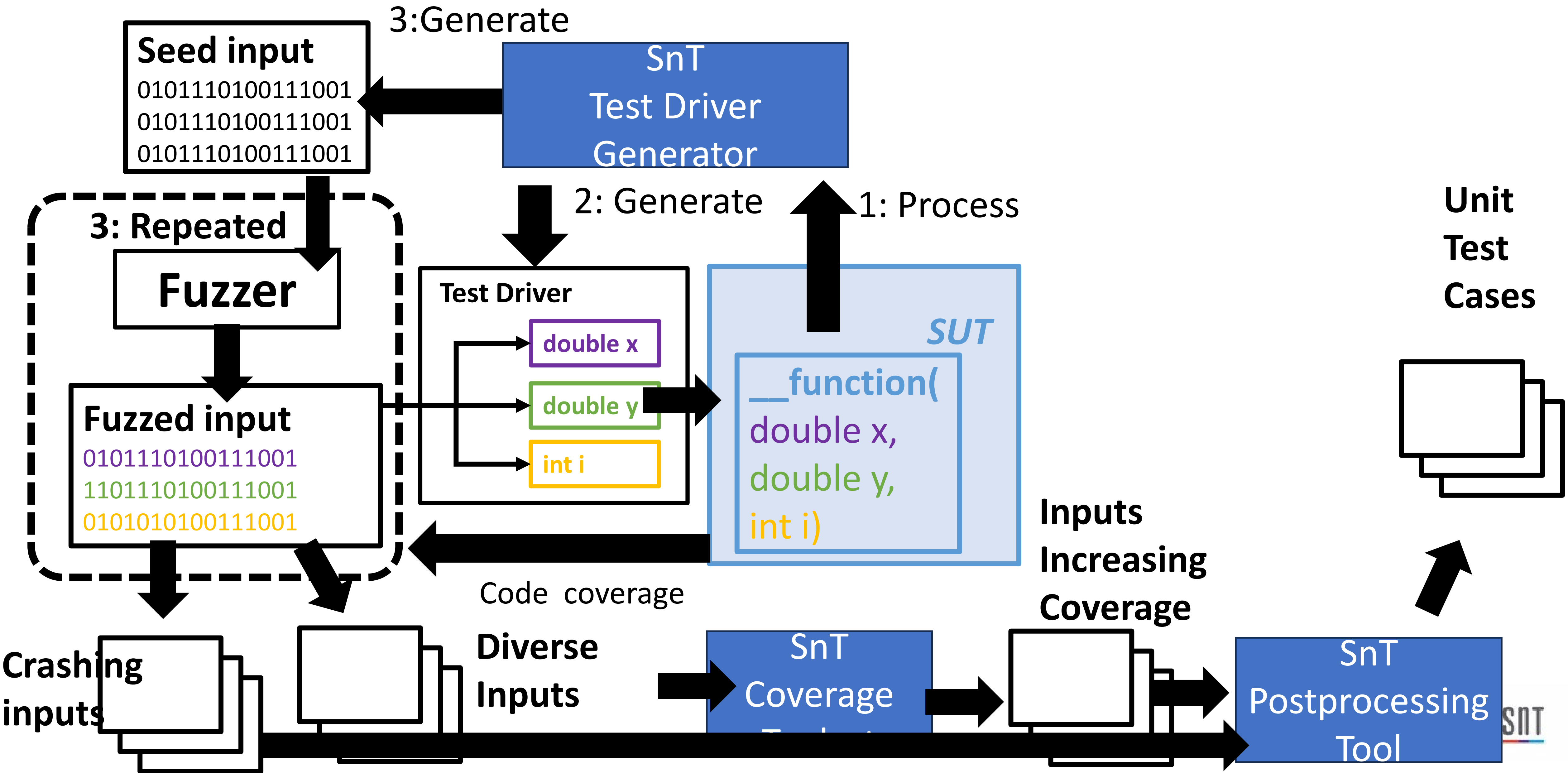
Inputs Increasing Coverage

SnT Postprocessing Tool

Unit Test Cases

```
/*Variables with the data in the file genertae by the fuzzer*/
const char input_data_pVal={0xFF,0xFF,0xFF,0xFF};
const char input_data_pErrCode={0xFF,0xFF,0xFF,0xFF};
/*Variables with the values observed after function execution, for regression*/
const char expected_pVal={0xFF,0xFF,0xFF,0xFF};
const char expected_pErrCode={0xFF,0xFF,0xFF,0xFF};
const char expected_return={0x00,0x00,0x00,0x00};
/*Test case*/
int main(int argc, char** argv){
  T_POS pVal;
  int pErrCode;
  int _return;
  /*Initialize inputs*/
  memcopy(&pVal, input_data_pVal, sizeof(pVal));
  memcopy(&pErrCode, input_data_pErrCode, sizeof(int));
  /* Invoke the function under test*/
  _return = T_POS_IsConstraintValid(&pVal, &pErrCode);
  /* Print output values of the function under test*/
  printf_struct("pVal (T_POS)=", &pVal, sizeof(pVal));
  printf("pErrCode (int) = %d\n", pErrCode);
  printf("return (flag) = %d\n", _return);
  /* Generated assertions enabling  regression testing*/
  assert( 0==compare((char*)&pVal, sizeof(pVal)));
  assert( 0==compare((char*)&pErrCode, sizeof(pErrCode)));
  assert( 0==compare((char*)&_return, sizeof(_return)));
  return 0;
}
```

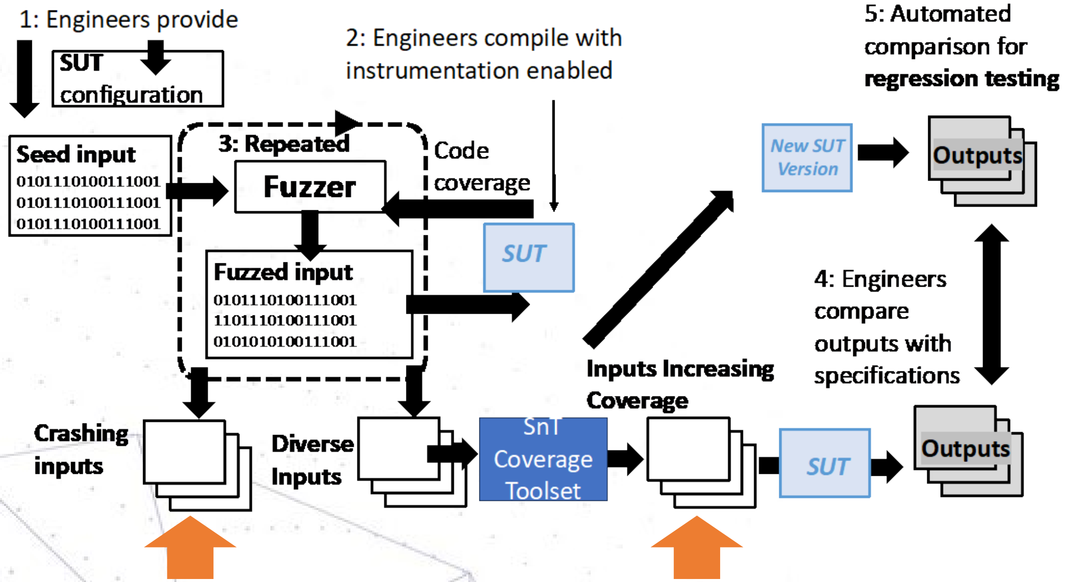
Proposed Methodology: Unit-level Functional Testing

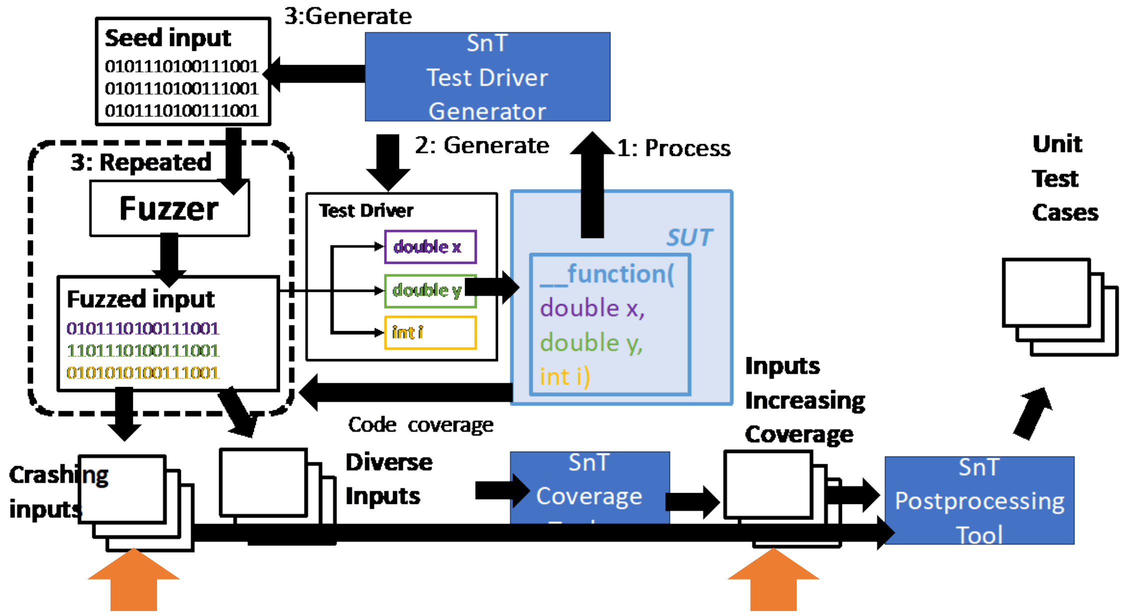


Is it feasible?

Feasibility study

Focus on coverage and crash detection for flight and ground segment



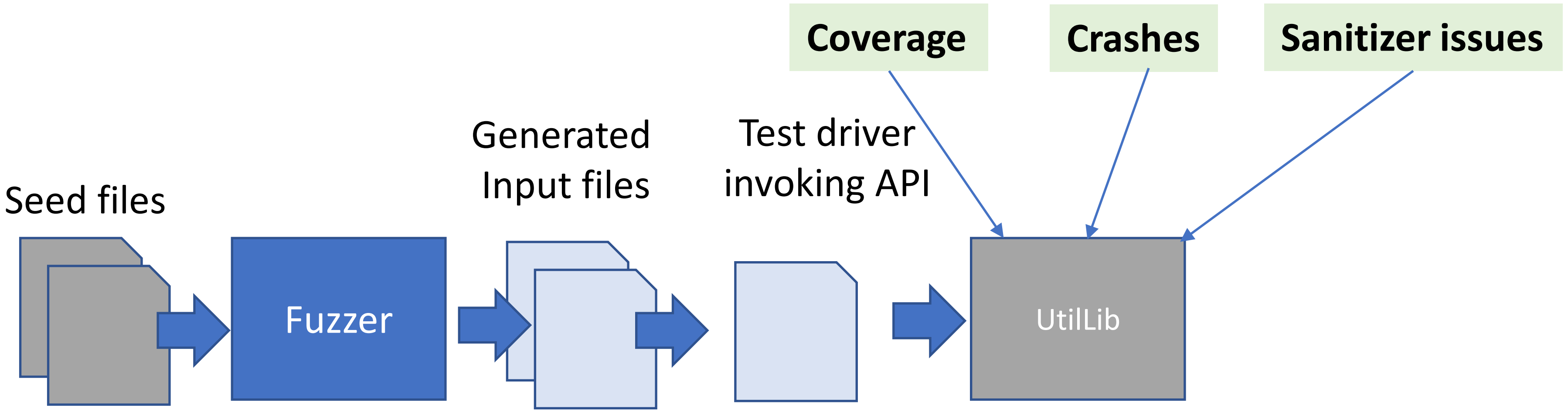


- RQ1: Can fuzzing automate functional testing at unit- and system-level?
- RQ2: How do fuzzing options contribute to fuzzing results?
- RQ3: How do different fuzzers compare for functional testing?
- RQ4: How does fuzzing perform for code sanitization purposes?

Subjects

Subjects: Unit Testing (1)

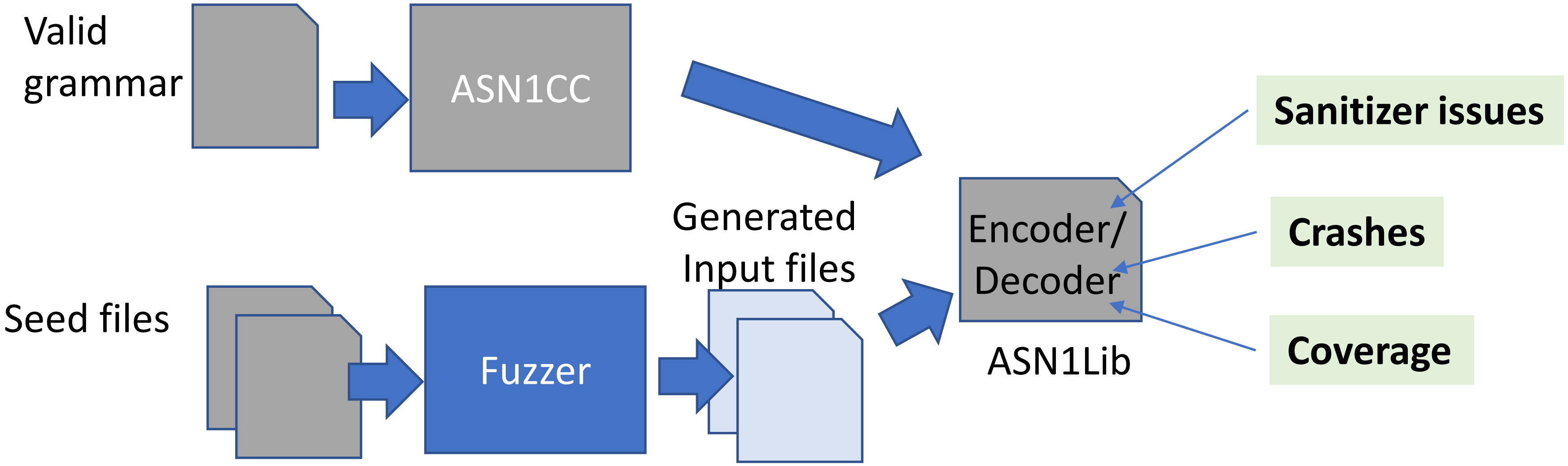
Commercial Utility Library



Subjects: Unit Testing (2)

ASN1Lib

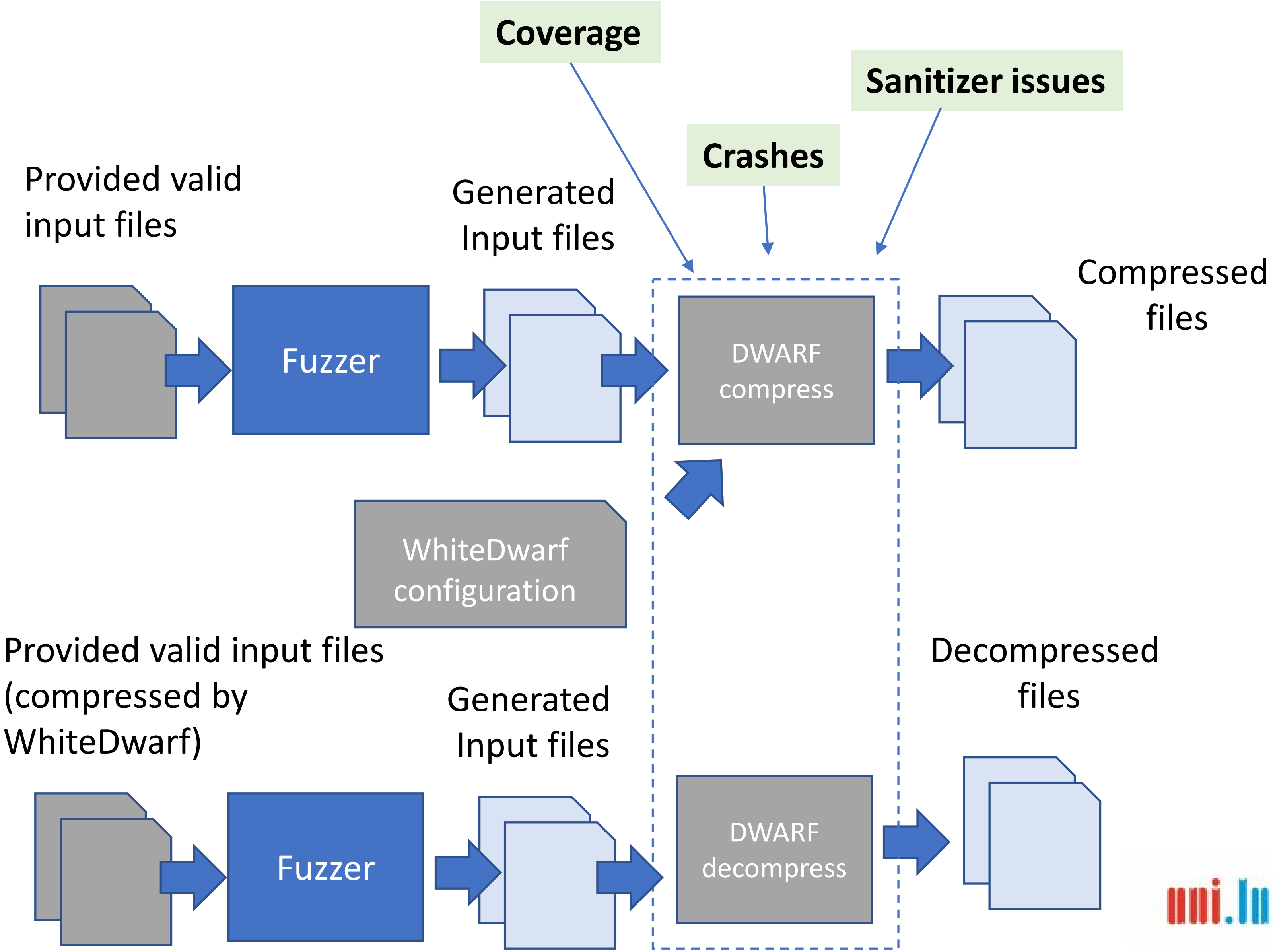
- Data serialization and deserialization



Subjects: System Testing (1)

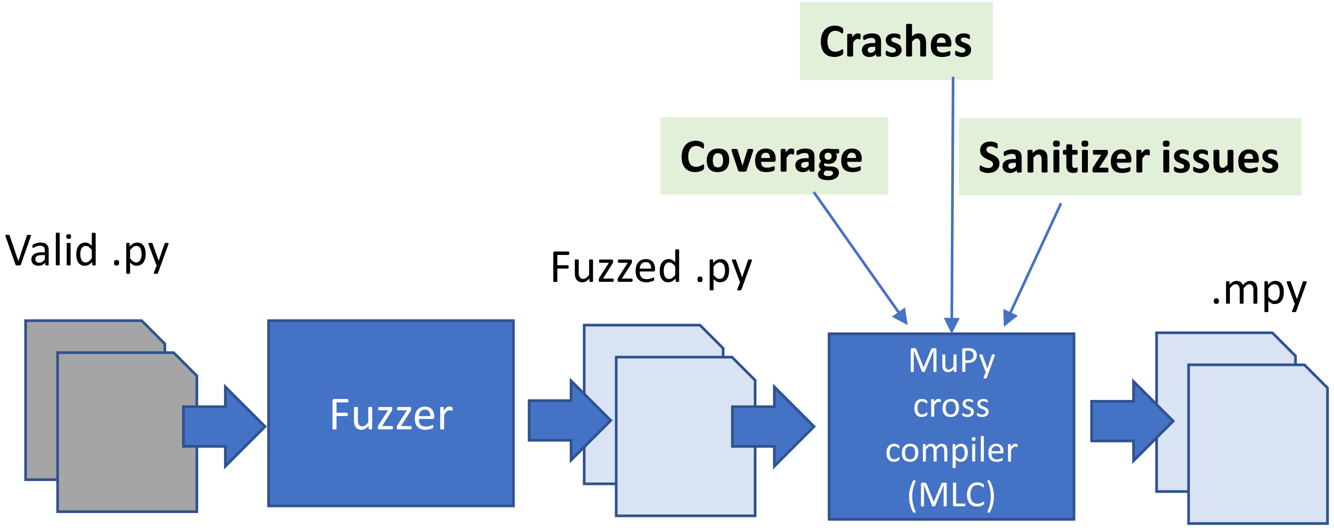
WhiteDwarf

- Data compression tool for CCSDS compression algorithms used in ESA missions (e.g., CSSDS 121.0-B-1, 122.0-B-1 and 123.0-B-1)



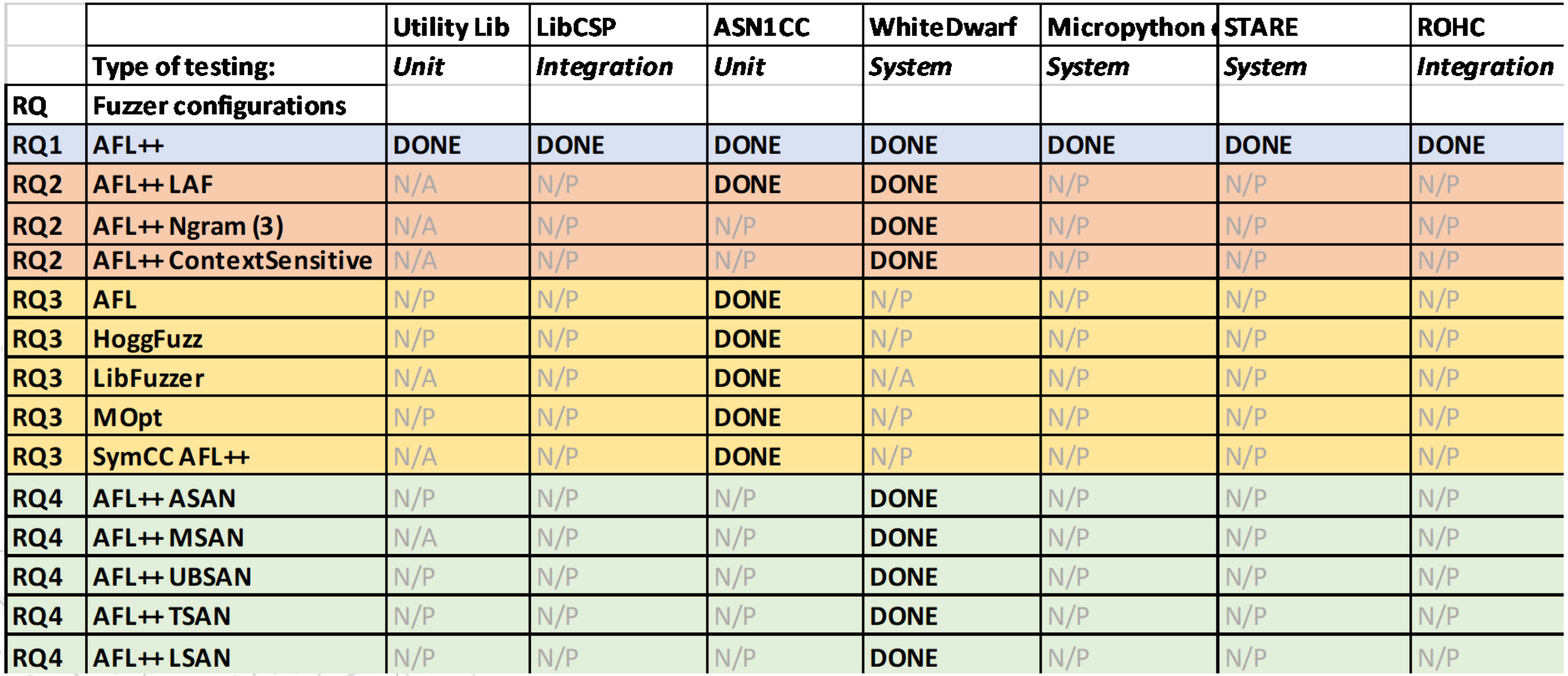
Subjects: System Testing (2)

Micropython Cross Compiler for Leon



Experiment Design

Experiments table



		Utility Lib	LibCSP	ASN1CC	WhiteDwarf	Micropython	STARE	ROHC
	Type of testing:	*Unit*	*Integration*	*Unit*	*System*	*System*	*System*	*Integration*
RQ	Fuzzer configurations							
RQ1	AFL++	DONE	DONE	DONE	DONE	DONE	DONE	DONE
RQ2	AFL++ LAF	N/A	N/P	DONE	DONE	N/P	N/P	N/P
RQ2	AFL++ Ngram (3)	N/A	N/P	N/P	DONE	N/P	N/P	N/P
RQ2	AFL++ ContextSensitive	N/A	N/P	N/P	DONE	N/P	N/P	N/P
RQ3	AFL	N/P	N/P	DONE	N/P	N/P	N/P	N/P
RQ3	HoggFuzz	N/P	N/P	DONE	N/P	N/P	N/P	N/P
RQ3	LibFuzzer	N/A	N/P	DONE	N/A	N/P	N/P	N/P
RQ3	MOpt	N/P	N/P	DONE	N/P	N/P	N/P	N/P
RQ3	SymCC AFL++	N/A	N/P	DONE	N/P	N/P	N/P	N/P
RQ4	AFL++ ASAN	N/P	N/P	N/P	DONE	N/P	N/P	N/P
RQ4	AFL++ MSAN	N/A	N/P	N/P	DONE	N/P	N/P	N/P
RQ4	AFL++ UBSAN	N/P	N/P	N/P	DONE	N/P	N/P	N/P
RQ4	AFL++ TSAN	N/P	N/P	N/P	DONE	N/P	N/P	N/P
RQ4	AFL++ LSAN	N/P	N/P	N/P	DONE	N/P	N/P	N/P

- RQ1: Can fuzzing automate functional testing at unit- and system-level?
- RQ2: How do fuzzing options contribute to fuzzing results?
- RQ3: How do different fuzzers compare for functional testing?
- RQ4: How does fuzzing perform for code sanitization purposes?

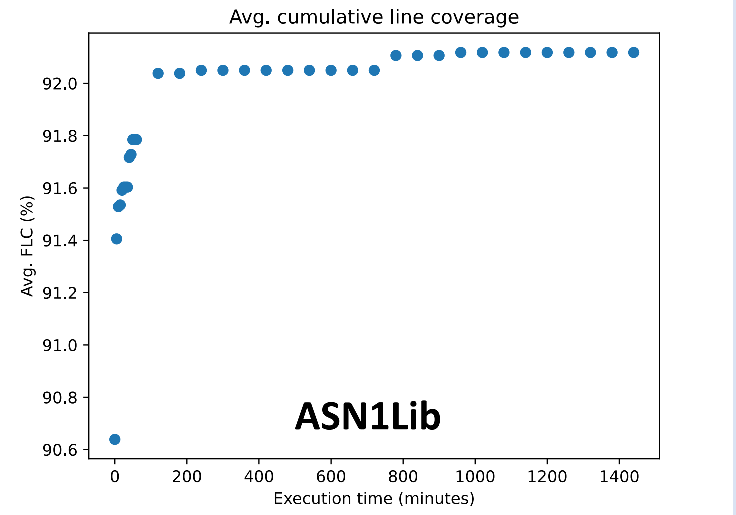
Metrics

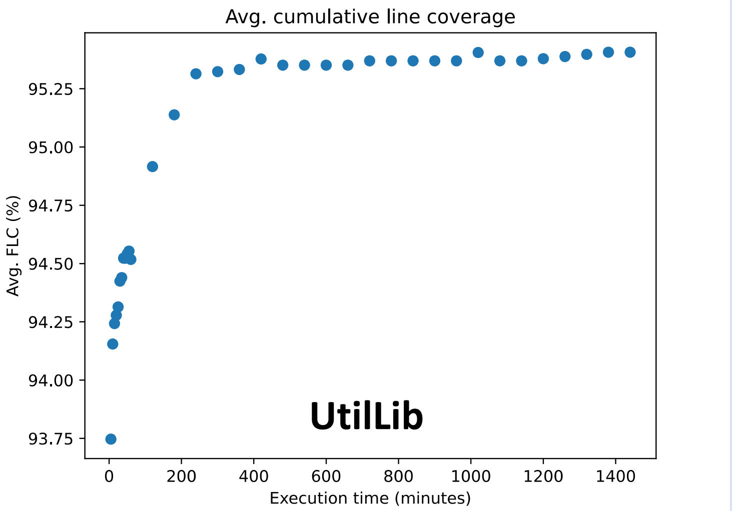
- Function Line/Branch Coverage (FLC/FBC):
  - Percentage of executable lines/branches covered, belonging to functions reached during testing
  - Avoid sides effect due to functions not reachable because of specific SUT configurations
  - For unit testing, focus on the functions under test:
    - FLC/FBC matches the line/branch coverage for the functions for which we generated a test driver (i.e., the targets of our testing process)
- Cumulative number of crashes
  - Natural
  - Or triggered by sanitizers

Results

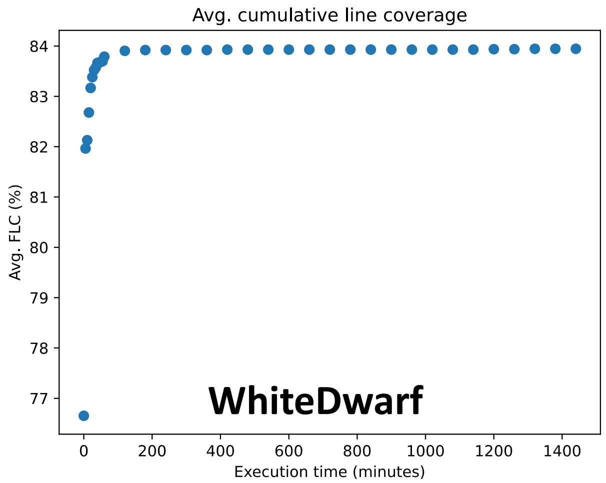
RQ1: Can fuzzing automate functional testing at unit- and system-level?

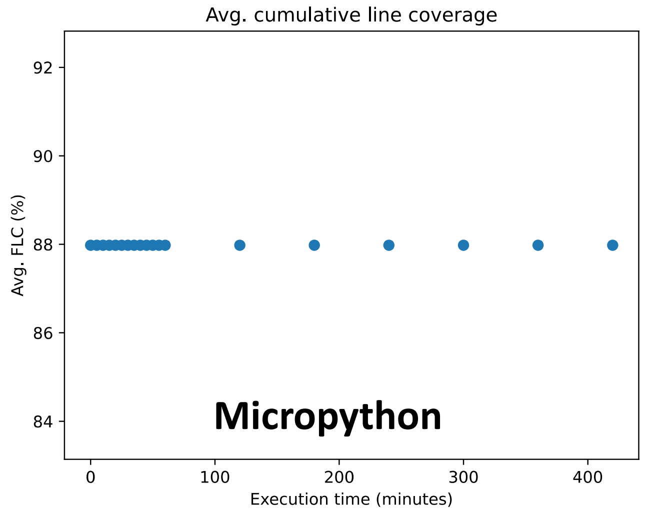
Fuzzing for Unit Testing





Fuzzing for System Testing

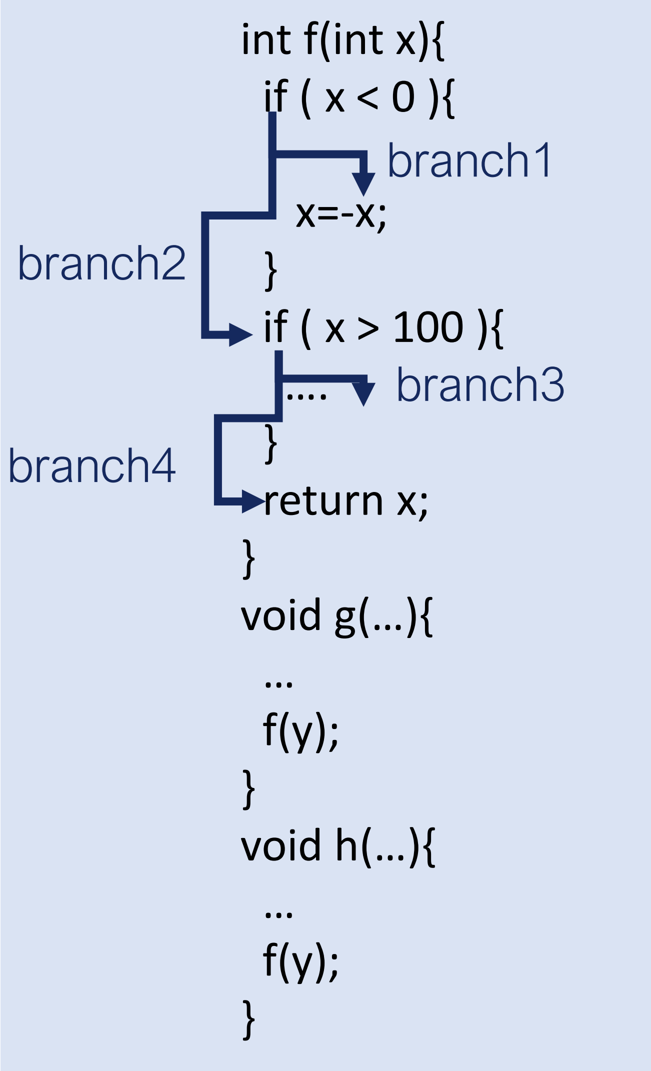




- Fuzzing enables reaching high FLC for unit testing
- Fuzzing is effective for system-level testing of software that processes binary data.
- For software that processes text files, grammar-based fuzzers should be used.

RQ2: How do fuzzing options contribute to fuzzing results?

- **CTX**: takes calling context into consideration
  - an input that
    exercises branch 1, when function f is invoked by function g,
    differs from
    an input that exercises branch 1, when function f is invoked by function h
- **NGRAM2**: consider edge pairs to achieve path coverage
  - four inputs, respectively exercising:
    - branch1 and branch 4
    - branch2 and branch 4
    - branch1 and branch 3
    - branch2 and branch 3
  - are considered diverse
- **LAF**: split expressions into branches
  - replaces “if (x >= 0)”
  - with “if (x > 0 or x == 0)”



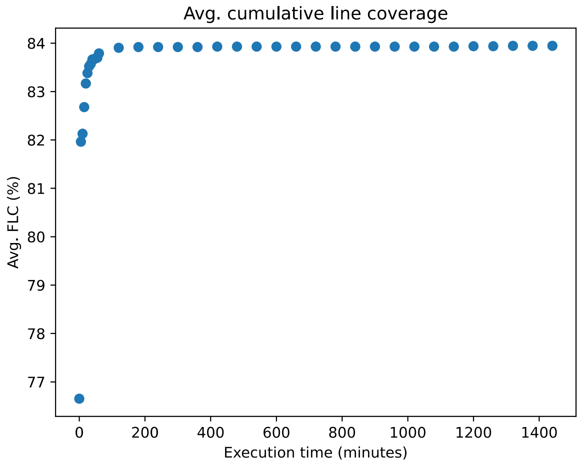
```
int f(int x){
 if ( x < 0 ){
   x=-x;
 }
 if ( x > 100 ){
  ....
 }
 return x;
}
void g(...){
 ...
 f(y);
}
void h(...){
 ...
 f(y);
}
```

RQ2: How do fuzzing options contribute to fuzzing results?

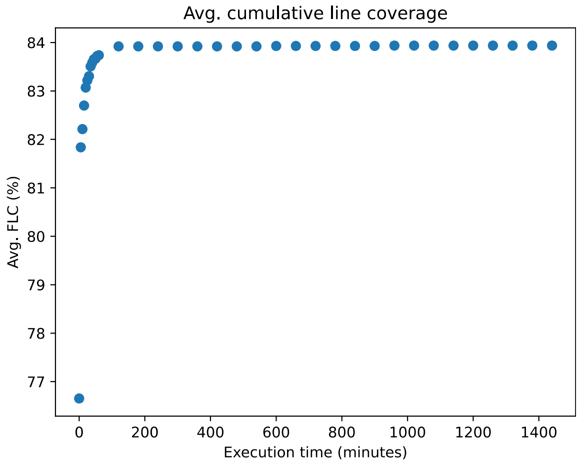
Fuzzing for System Testing

WhiteDwarf

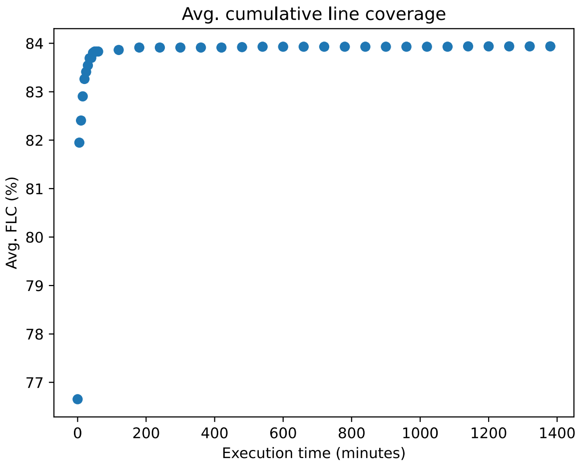
AFL++



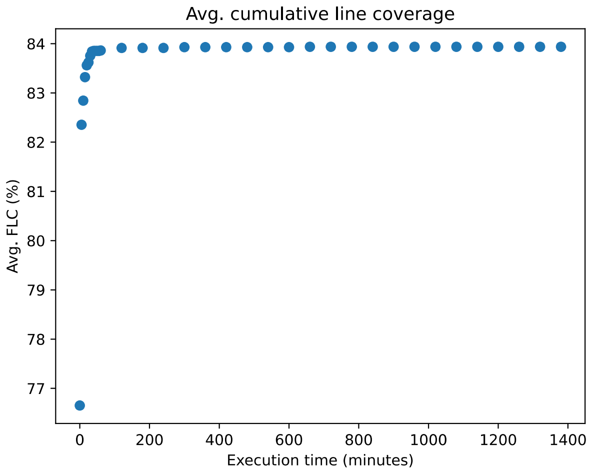
AFL++ LAF



AFL++ CTX

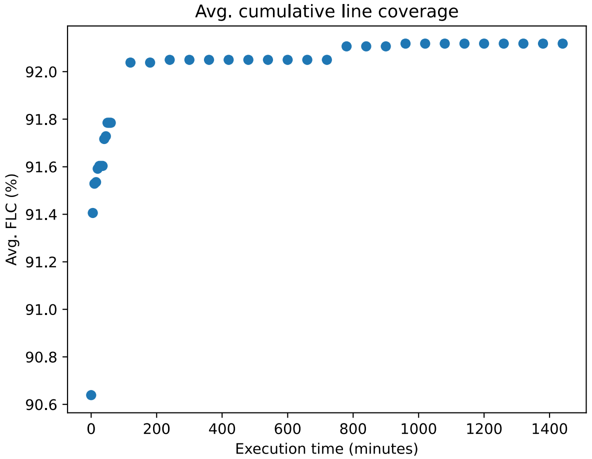


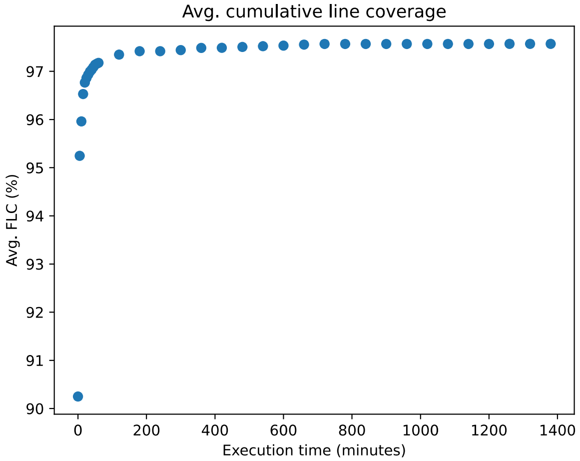
AFL++ NGRAM2



Fuzzing for Unit Testing

ASN1Lib



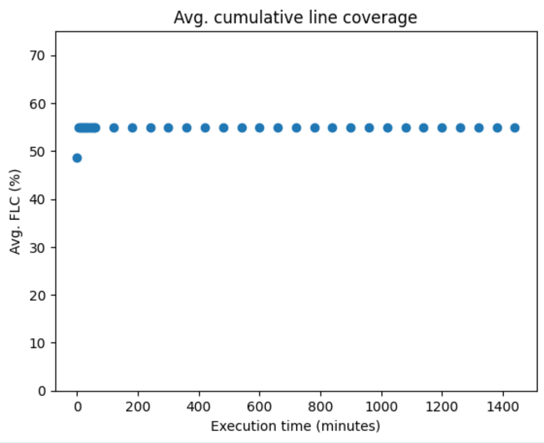


- For system-level, CTX and NGRAM lead to high coverage quicker
- For unit-level, LAF increases effectiveness (+5 percentage points wrt default AFL++)

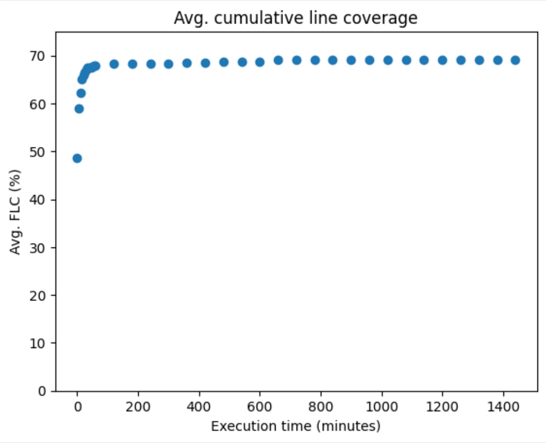
RQ3: How do different fuzzers compare for functional testing?

Fuzzing for Unit Testing

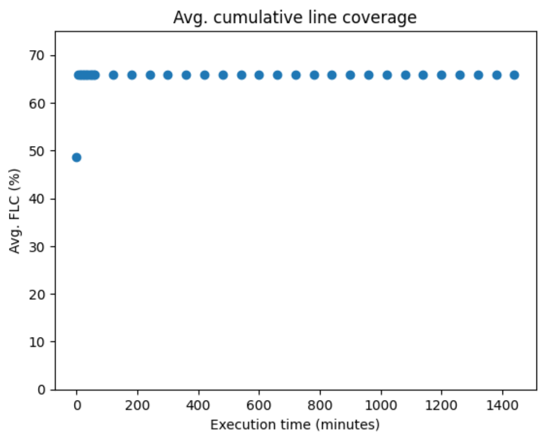
AFL++



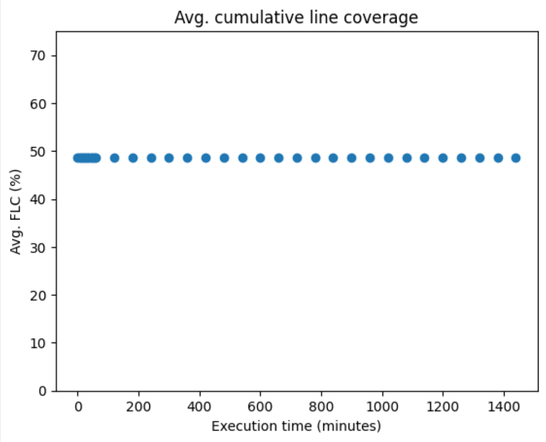
AFL++ LAF



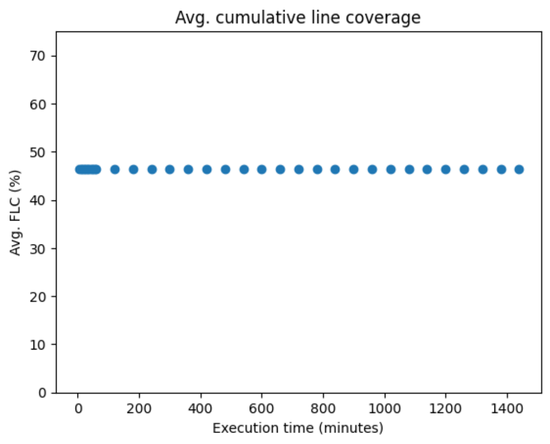
HonggFuzz



SymCC

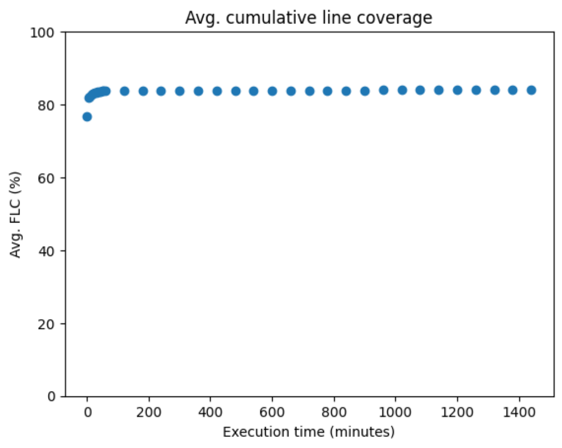


LibFuzzer

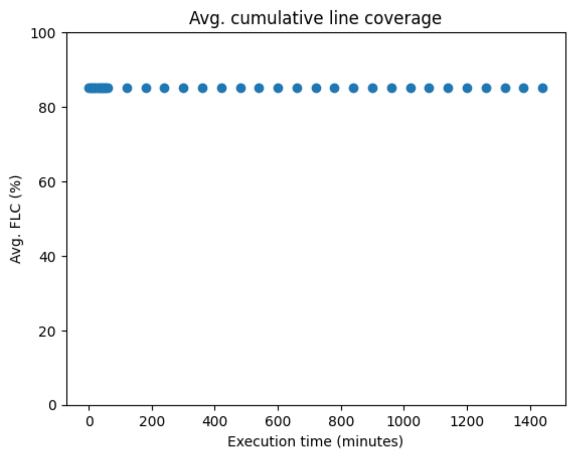


Fuzzing for System Testing

AFL++ NGRAM



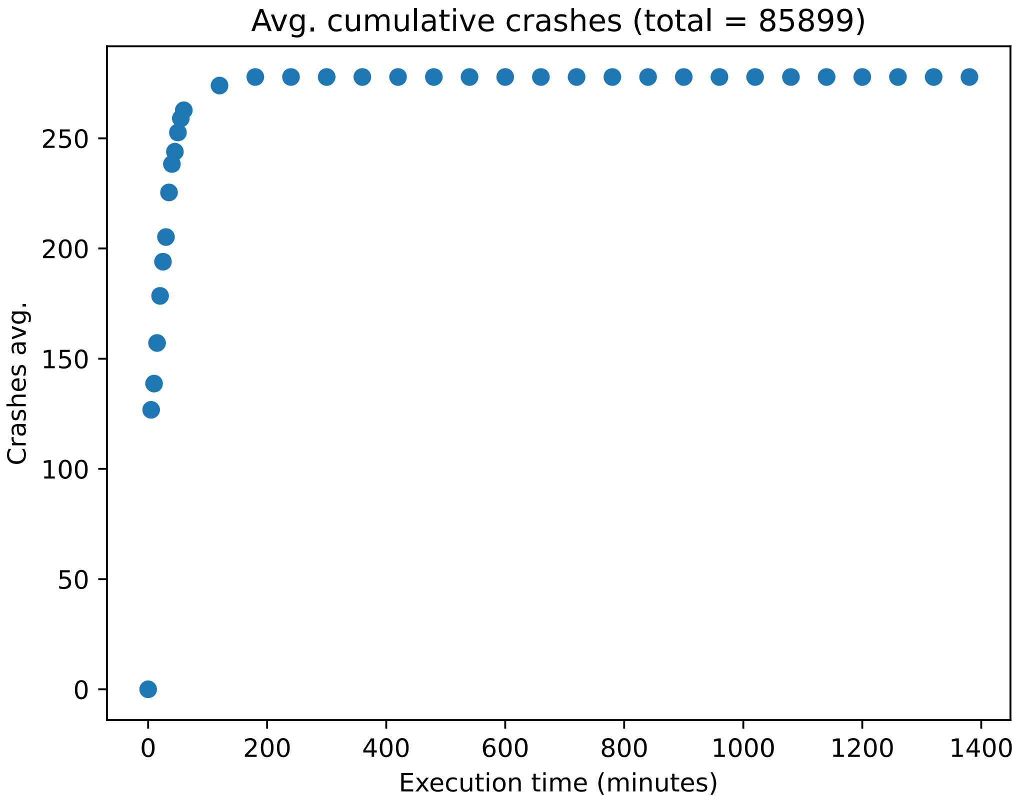
HonggFuzz



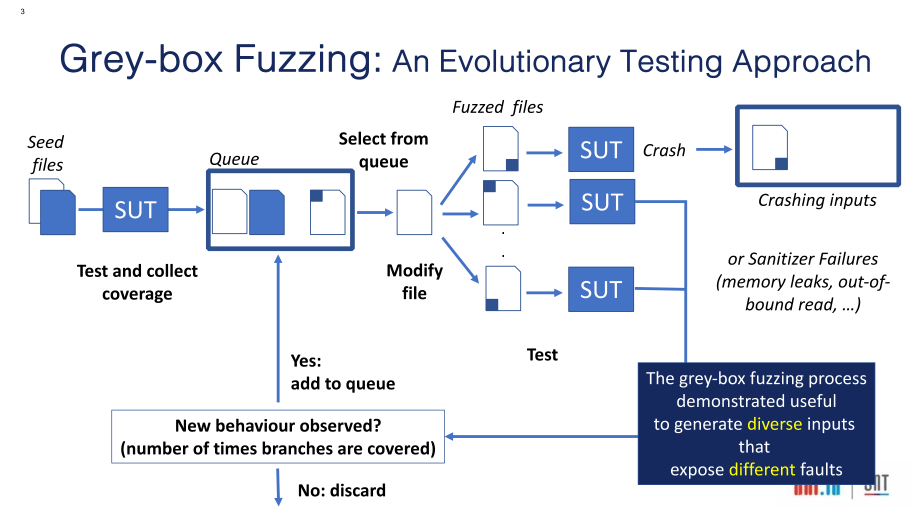
- AFL++ with LAF is the best for unit testing
- HongFuzz is the best for system testing

RQ4: How does fuzzing perform for code sanitization purposes? (Systel-level)

AFL++ with LSAN

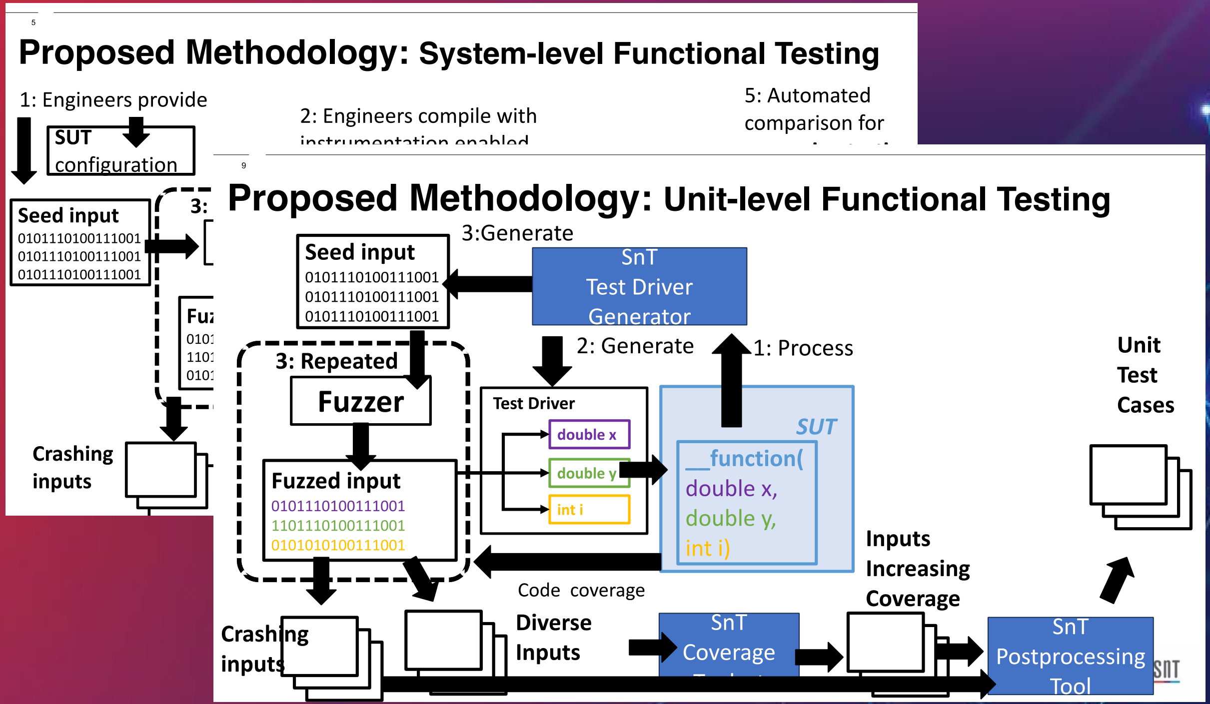


- Tested sanitizers:
    - ASAN. Address SANitizer. It detects memory corruption vulnerabilities like use-after-free, NULL pointer dereference, and buffer overruns.
    - UBSAN. Undefined Behavior SANitizer.
    - MSAN. Memory Sanitizer. It reports uninitialized memory.
    - TSAN. Thread Sanitizer. It discovers thread race conditions.
    - CFISAN. Control Flow Integrity SANitizer.
    - LSAN. Leak SANitizer. It reports memory leaks in a program.
- LSAN has identified errors in in the allocation of memory for WhiteDwarf



Grey-box Fuzzing: An Evolutionary Testing Approach

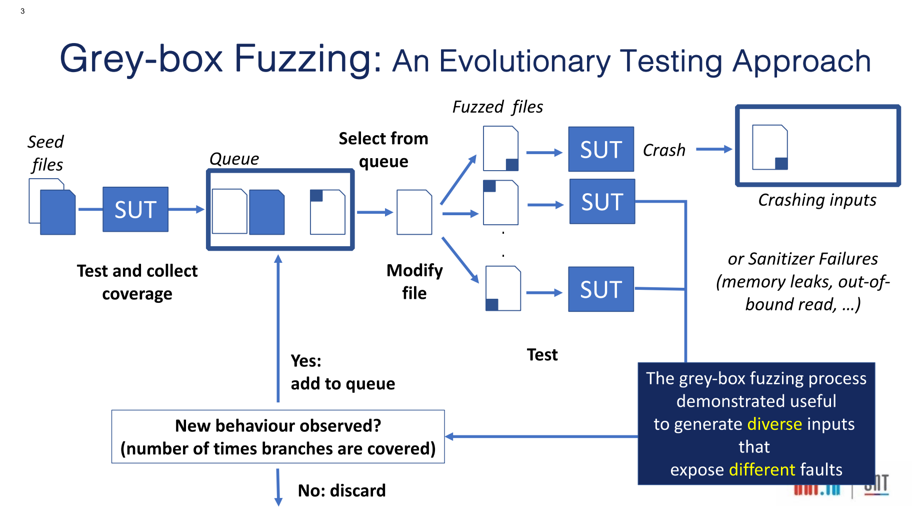
The grey-box fuzzing process demonstrated useful to generate diverse inputs that expose different faults



Proposed Methodology: System-level Functional Testing

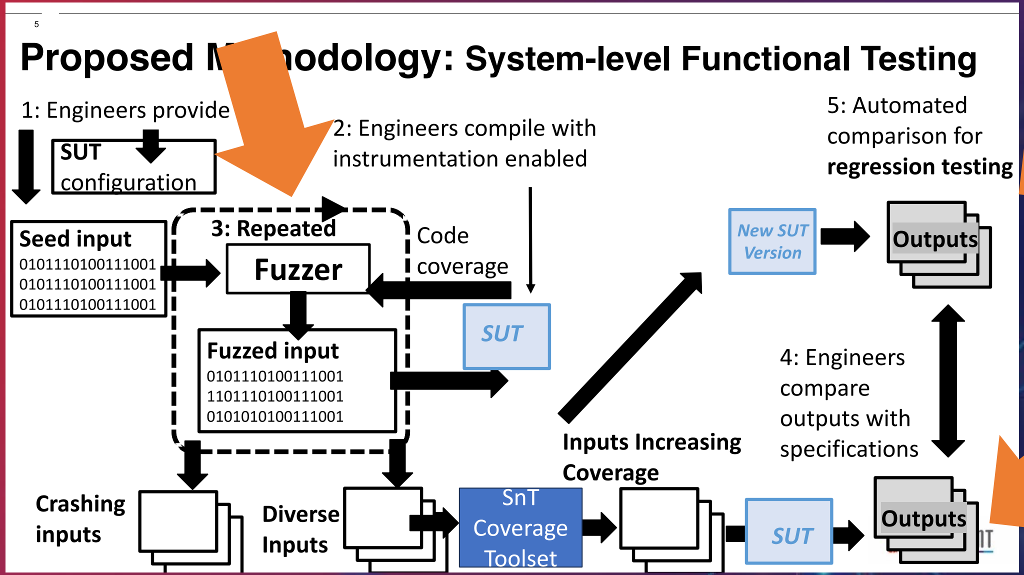
Proposed Methodology: Unit-level Functional Testing

- Feasibility results:
  - Demonstrated that fuzzing is applicable to space software and effective not only for robustness testing
    but also to achieve very high coverage in unit testing
  - The fuzzer options LAF and NGRAM/CTX should be used for unit and system-level testing, respectively
  - AFL++ with LAF performs best for unit,
    Honggfuzz for system-testing
  - Fuzzing demonstrated effective for leak detection



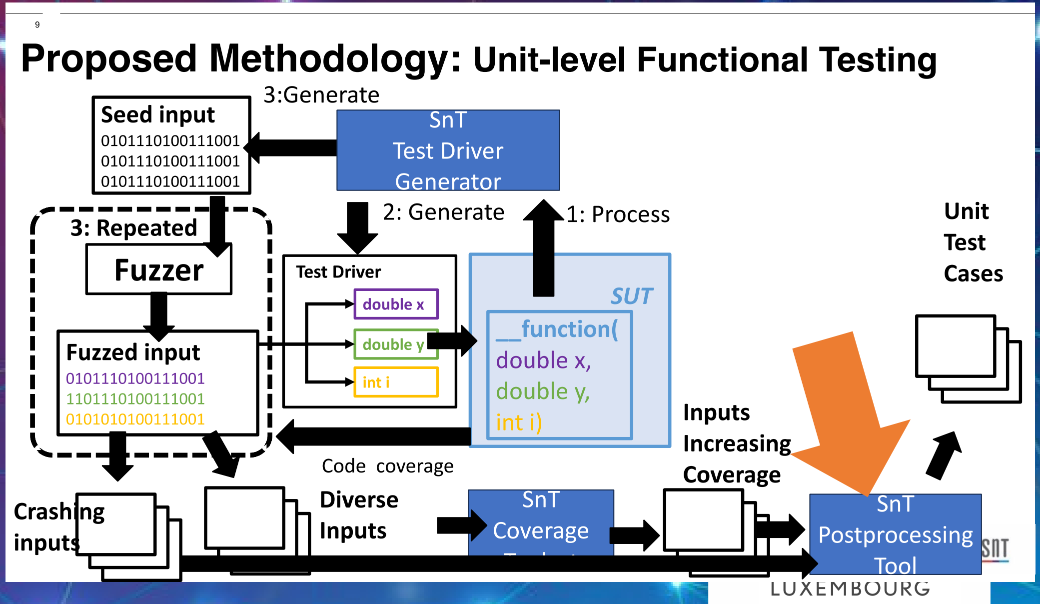
Grey-box Fuzzing: An Evolutionary Testing Approach

The grey-box fuzzing process demonstrated useful to generate diverse inputs that expose different faults



Proposed M odology: System-level Functional Testing

- Future work:
  - Finalize a unit testing tool leveraging fuzzers
  - Assess grammar-based fuzzing tools for standard interfaces (e.g., TC/TM)
  - Integrate strategies to prioritize the inspection of outputs
  - Facilitate regression testing at system-level



Proposed Methodology: Unit-level Functional Testing

EXPRO / TEC-Standardisation
07/2022 – 07/2023

Applicability of Fuzz Testing to Space Software

A-Prof. Fabrizio Pastore
University of Luxembourg
fabrizio.pastore@uni.lu

ADCSS 2023 - November 15th, 2023