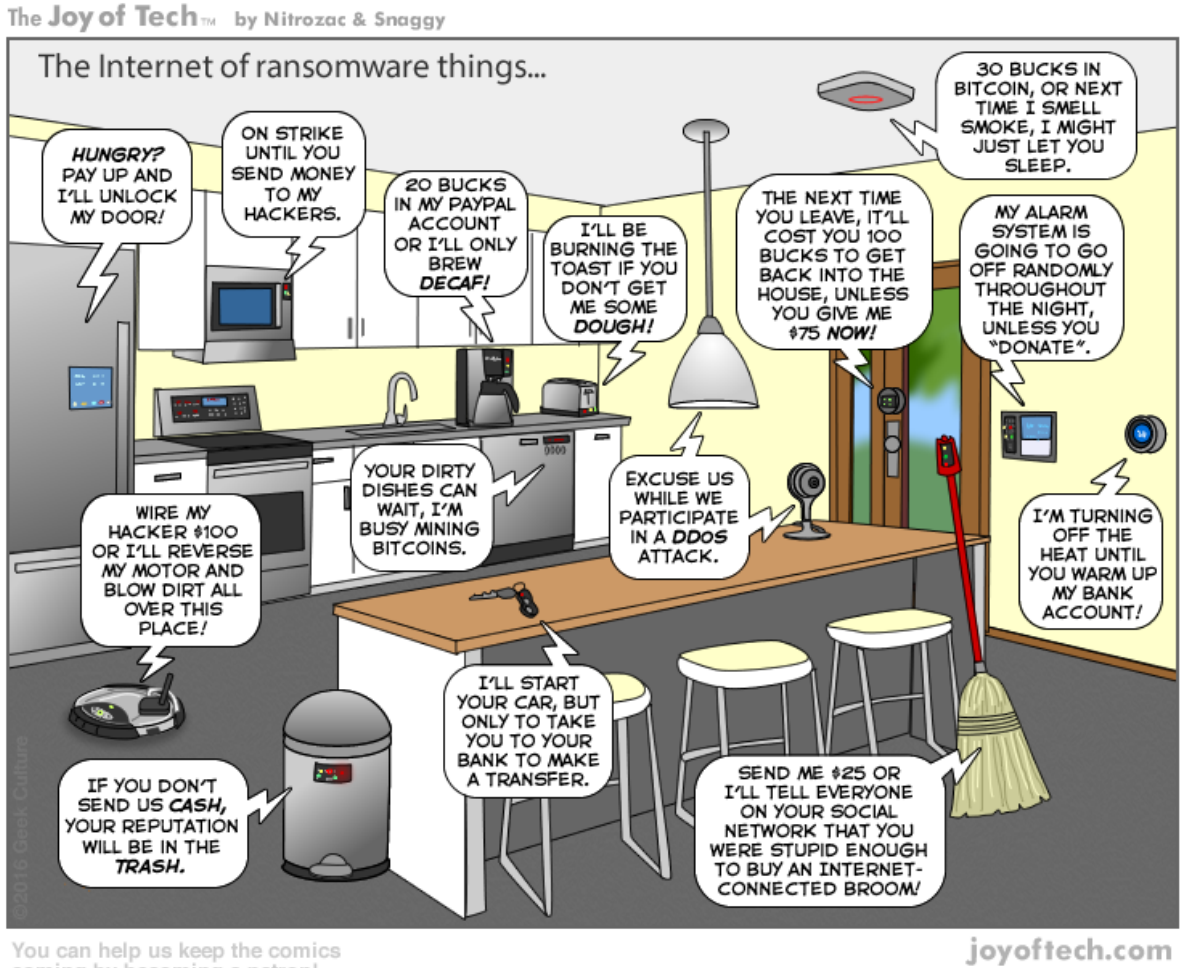
MINI-TUTORIAL ON AUTOMATED DISCOVERY OF SECURITY VULNERABILITIES USING FUZZING

BOSCH

Agenda

1. Motivation
2. Definition of Fuzzing and where it is in the security landscape
3. Targets that can be fuzzed
4. Fuzzing types: black-box, grey-box, and white-box
5. Fuzzing types: source code fuzzing and protocol fuzzing
6. Fuzzing metrics and coverage information
7. Fuzz test optimization: sanitizers, seeds, dictionaries, and parallelization
8. Challenges and good practices

Motivation



One of the first bugs discovered by fuzzing (30 years old)

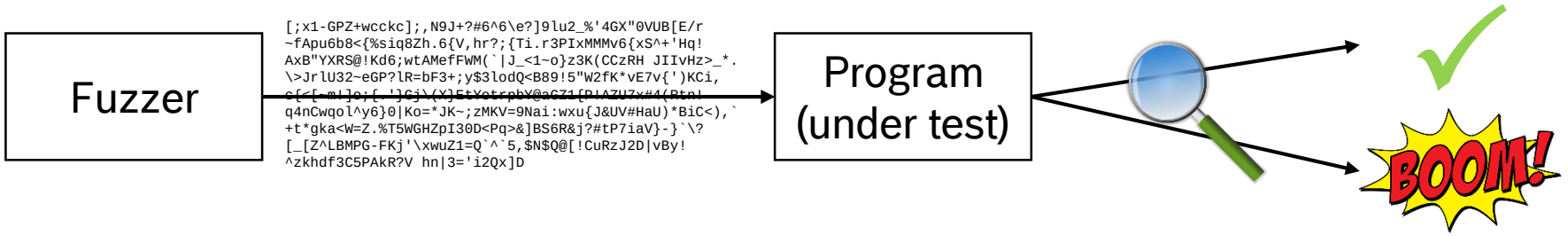
Example Vulnerability

```
while ((cc = getch()) != c)
{
  buf[i++] = cc;
  ...
}
```

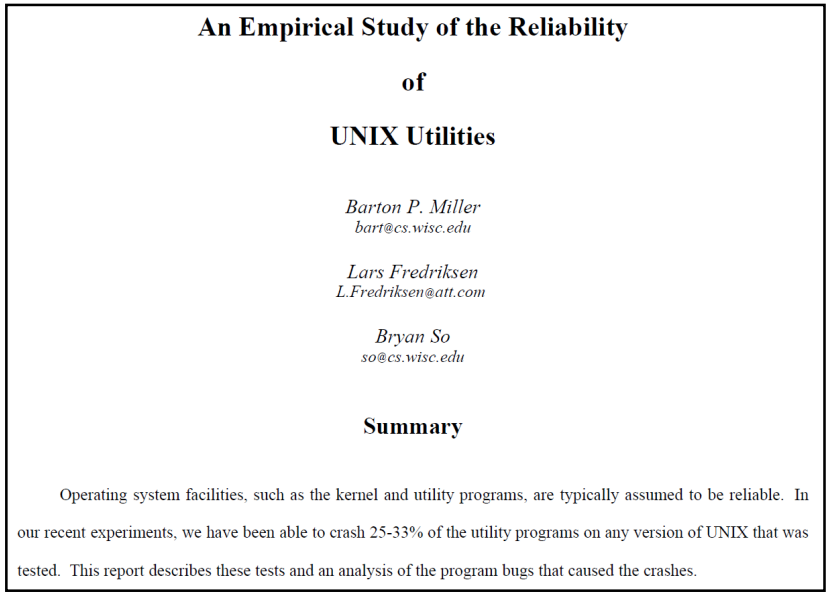
1. No check on the length of buffer `buf`
2. Write own code on the stack
3. 

What is Fuzzing?

"Fuzzing is a dynamic technique. To find bugs, it must trigger the code that contains these bugs.", Hui Peng, *T-Fuzz: fuzzing by program transformation*, 2018



- Fuzzing was coined in 1989, when Miller *et al.* used a random testing tool to investigate the reliability of UNIX tools.
- Fuzzing automatically generates
  - unexpected, malformed or random data
  - and provides this data as input to a software under test.
  - Software under test is monitored, e.g. for crashes or hangs.



An Empirical Study of the Reliability

of

UNIX Utilities

Barton P. Miller
bart@cs.wisc.edu

Lars Fredriksen
L.Fredriksen@att.com

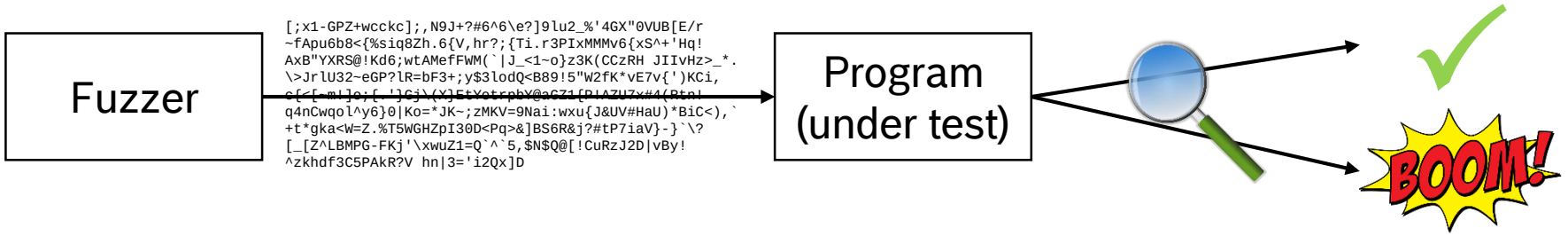
Bryan So
so@cs.wisc.edu

Summary

Operating system facilities, such as the kernel and utility programs, are typically assumed to be reliable. In our recent experiments, we have been able to crash 25-33% of the utility programs on any version of UNIX that was tested. This report describes these tests and an analysis of the program bugs that caused the crashes.

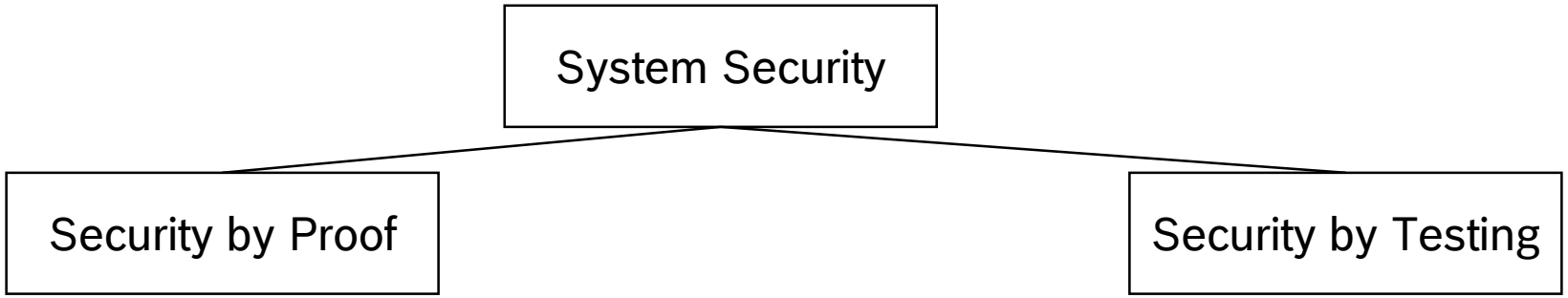
What is Fuzzing?

"Fuzzing is a dynamic technique. To find bugs, it must trigger the code that contains these bugs.", Hui Peng, *T-Fuzz: fuzzing by program transformation*, 2018



- Overall goal is to validate a robust program behavior.
  - When a program accepts data from an untrusted input (or faulty in general), unwanted and observable behavior should be avoided.
  - In more detail, fuzzing metrics, such as code coverage and time, can be maximized.
- Fuzz testing can detect bugs which can lead to vulnerabilities.
- The generated input that triggers a bug is saved, and thus provides a reproducible test case.
- Therefore, fuzzing is limited to discovering symptoms for exploitable bugs.

Where is Fuzzing in the Security Landscape? (not complete)



- Systems are *provably secure*
  - specific attacks are *impossible*
  - always *behave as designed*
- Requires (expensive) mathematical proof

- Systems are *tested*
  - *Low probability* of successful attacks
  - Remaining attacks have *high complexity*
  - Cost-efficient if *automated*
- *No guarantee* of absence of bugs

What can be fuzzed?

Anything can be fuzzed that consumes untrusted, complex inputs.

- (Crypto-) Functions
- Parsers of any kind
- Media codecs
- Network protocols
- Compression
- Formatted output
- Compilers and interpreters
- Regular expression matchers
- Text processing
- Databases
- Browsers, text editors
- OS Kernels, drivers, supervisors, VMs

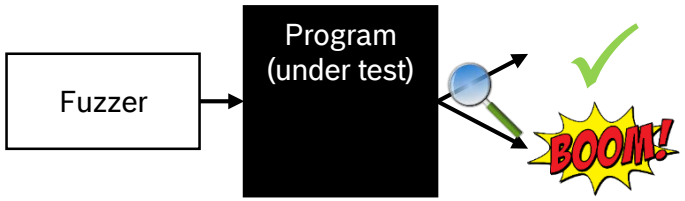
What can a fuzzer detect?

- The easiest is when a program crashes.
  - NULL dereferences, uncaught exceptions, div-by-zero, ...
- Additionally, with sanitizers, a fuzzer can detect
  - use-after-free, buffer overflows
  - uses of uninitialized memory, memory leaks
  - data races, deadlocks
  - int/float overflows, bitwise shifts by invalid amount
- Resource usage bugs
  - Memory exhaustion, hangs or infinite loops, infinite recursion (stack overflows)

Fuzzing Types
Example: Fuzz some source code, i.e. no protocol

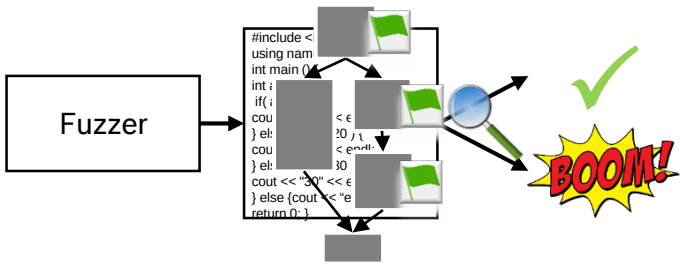
Black-box fuzzing ■

- Only requires the software under test to execute
- Assuming no source code
- Observes whether the program crashed (if at all)



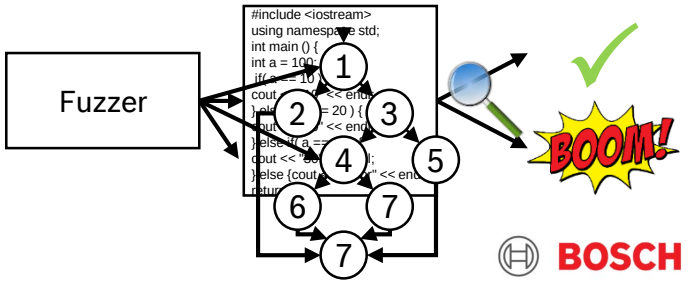
Grey-box fuzzing ■

- mixture of black-box and white-box fuzzing
- lightweight instrumentation
  - trace the program structure during monitoring



White-box fuzzing □

- Heavy-weight program analysis
- available source code
  - maybe more sophisticated reasoning
- Observe (and modify) semantics of a program's source code (including the binary)



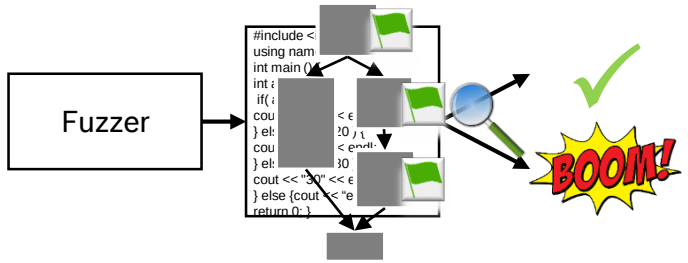
Fuzzing Types
Protocol and source code fuzzing

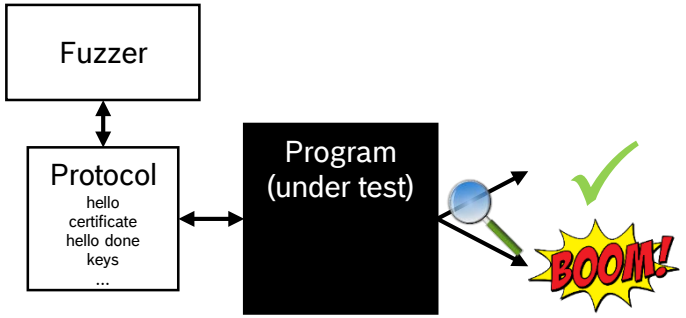
Source code fuzzing

- Focuses on bugs inside a program
- program states are secondary
- Good measure is e.g. code coverage

Protocol fuzzing

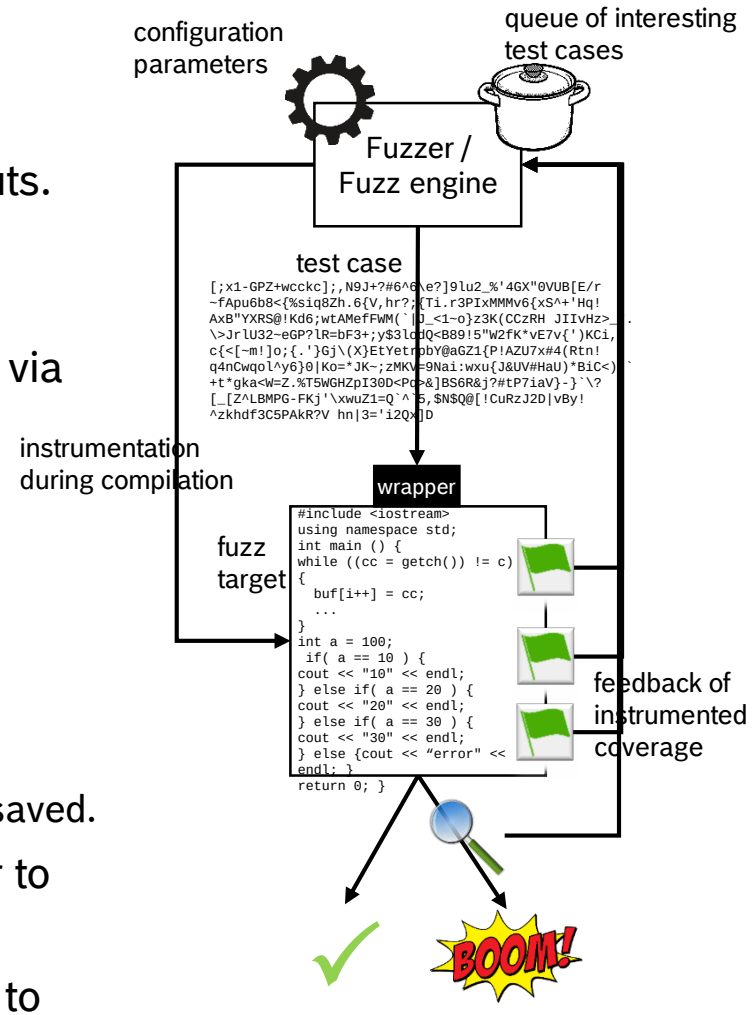
- Focuses on communication of a program
- Messages are delayed, intercepted, replayed, randomized, forged, etc.
- Fuzzer can be a MitM
- Protocol fuzzing is a black box test





Terminology

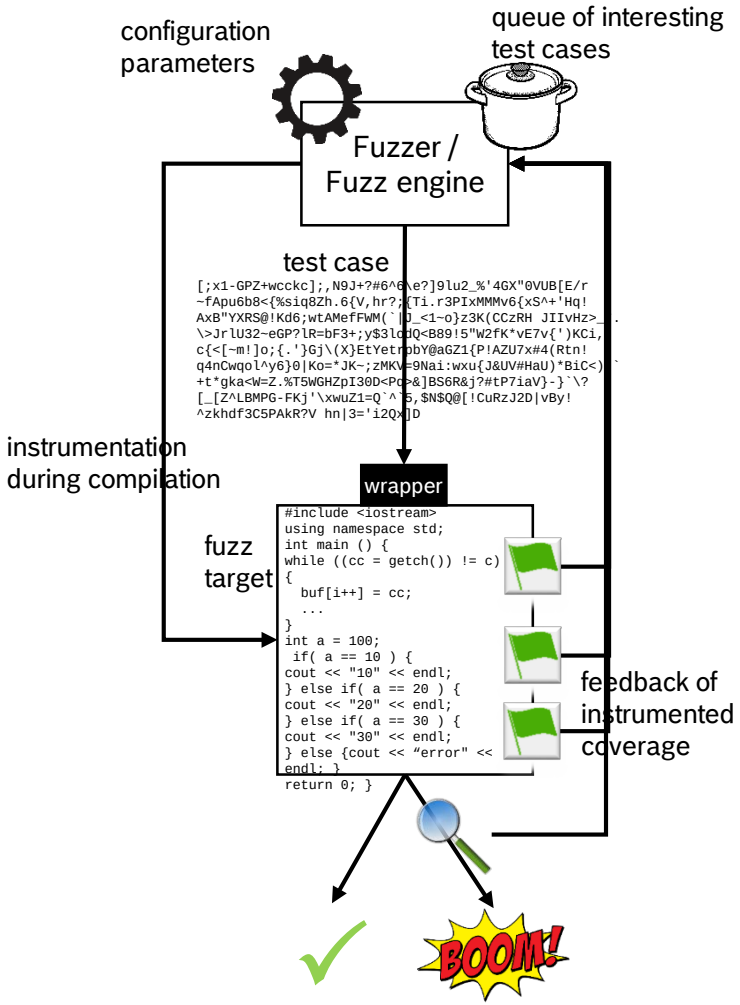
- "**Fuzzing**" or "**fuzz testing**" is the overall term.
- "**Fuzzer**" or "**fuzzing engine**" are programs that automatically generate inputs.
  - Not connected to any software under test, nor any instrumentation done.
  - Capabilities to instrument code, generate test cases and run programs under test.
- "**Fuzz target**" is a software program or function that is intended to be tested via fuzzing.
  - fuzz target consumes some untrusted input, which is then generated by a fuzzer.
- "**Fuzz Test**" is the combined version of a fuzzer and a fuzz target.
  - A fuzz test is executable.
- "**Glue code**" or "**wrapper**" or "**harness**" connects a fuzzer to a fuzz target.
- "**Test case**" is one specific input and test run from a fuzz test.
  - Usually for reproducibility, interesting runs (finding new code paths or crashes) are saved.
- "**Instrumentation**" is used for metrics, which are then fed back to the fuzzer to generate more interesting test cases.
- "**Configuration parameters**", such as "dictionary" or "seed" help the fuzzer to generate relevant input more quickly.



Christopher Huth | 2020-04-02

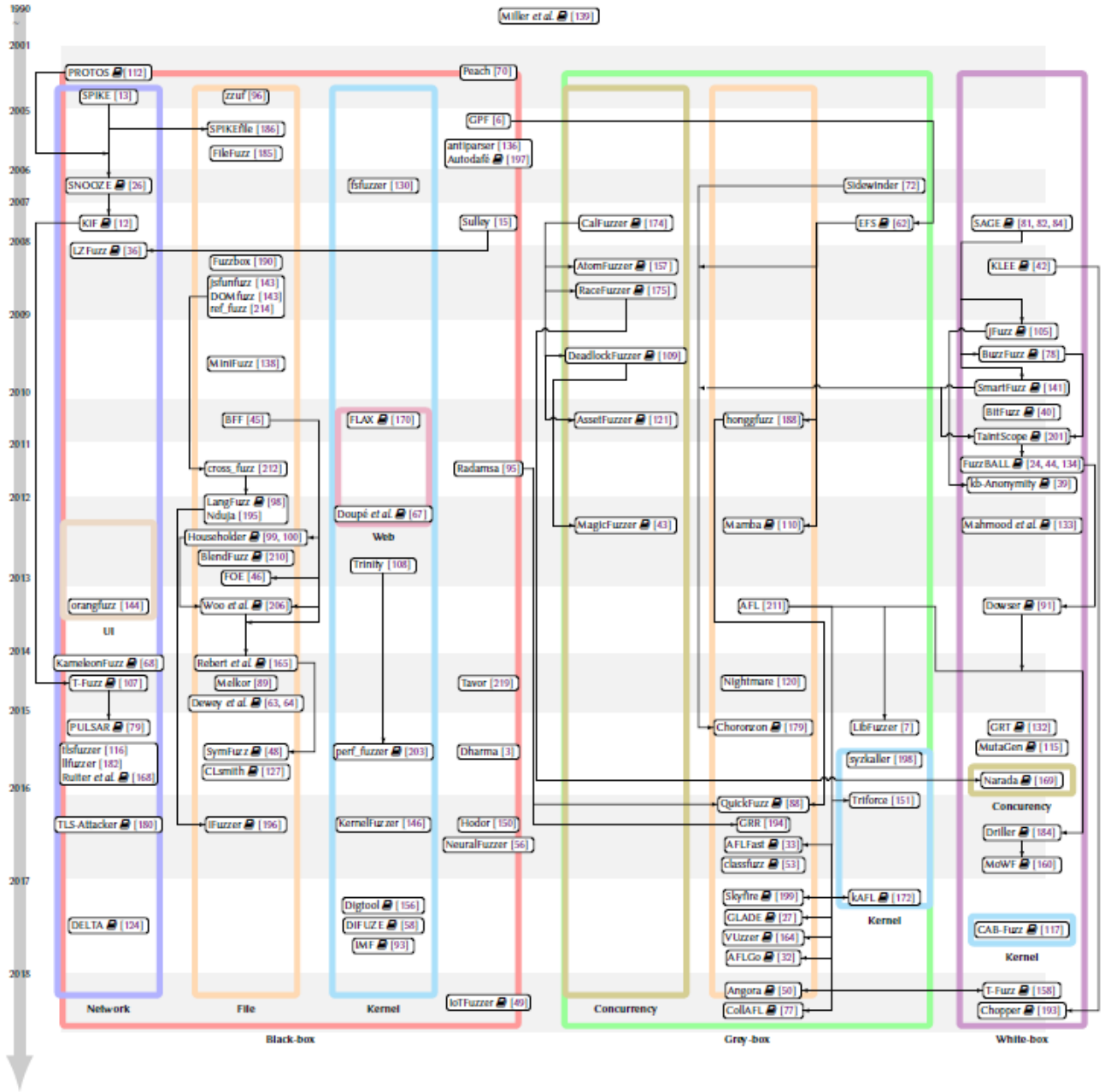
Metrics

- Comparing different fuzzers, or fuzzing runs, is hard
- Best possible metric is the **number of** (possibly exploitable) **bugs** identified by crashing inputs.
- Other metrics are:
  - “unique” crashes
  - Execution Speed
  - Total runtime / timeout
  - Code Coverage
    - or criticality regarding safety / security of software under test
    - or amount of interfaces
  - Corpus Size
- When comparing, use
  - benchmark programs (e.g. LAVA, CGC, “old” software)
  - same platform and configuration (dictionary, seed)
  - mean of multiple runs for varying durations



Tools (Far from complete)

- Fuzzing: Art, Science, and Engineering,
  VALENTIN J.M. MANES, KAIST CSRC, Korea, HYUNGSEOK HAN, KAIST, Korea, CHOONGWOO HAN, Naver Corp., Korea, SANG KIL CHA∗, KAIST, Korea, MANUEL EGELE, Boston University, USA, EDWARD J. SCHWARTZ, Carnegie Mellon University/Software Engineering Institute, USA, MAVERICK WOO, Carnegie Mellon University, USA



Let’s fuzz

- We look at two of the most popular fuzzers, AFL++ (a community-driven successor of AFL) and libFuzzer
  - Both are coverage-guided grey-box source code fuzzers
- As an example target software we use a snapshot from WOFF2 (font compression) and a wrapper from fuzzer-test-suite https://github.com/google/fuzzer-test-suite/tree/master/woff2-2016-05-06
  - We compile with afl-clang-fast for AFL++ and clang for libfuzzer to reuse the same wrapper

- https://github.com/AFLplusplus/AFLplusplus
- https://llvm.org/docs/LibFuzzer.html

Let’s fuzz

- Wrapper from https://github.com/google/fuzzer-test-suite/blob/master/woff2-2016-05-06/target.cc

```
// Copyright 2016 Google Inc. All Rights Reserved.
// Licensed under the Apache License, Version 2.0 (the "License");
#include <stddef.h>
#include <stdint.h>

#include "woff2_dec.h"

// Entry point for LibFuzzer.
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  std::string buf;
  woff2::WOFF2StringOut out(&buf);
  out.SetMaxSize(30 * 1024 * 1024);
  woff2::ConvertWOFF2ToTTF(data, size, &out);
  return 0;
}
```

- In short, LLVMFuzzerTestOneInput ‘replaces’ the main function of the software under test
  - Indicated by `-fsanitize=fuzzer` during compilation and linking
  - test case provided via `data` and `size`
  - test case injected into software by function `woff2::ConvertWOFF2ToTFF`

Let's fuzz

- Clone and build from https://github.com/AFLplusplus/AFLplusplus

- Clone https://github.com/google/fuzzer-test-suite

```
$ export FUZZING_ENGINE=afl
$ export CC=afl-clang-fast
$ export CXX=afl-clang-fast
$ ./build.sh
```

- Download clang (or build from sources) https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md
- clone https://github.com/google/fuzzer-test-suite

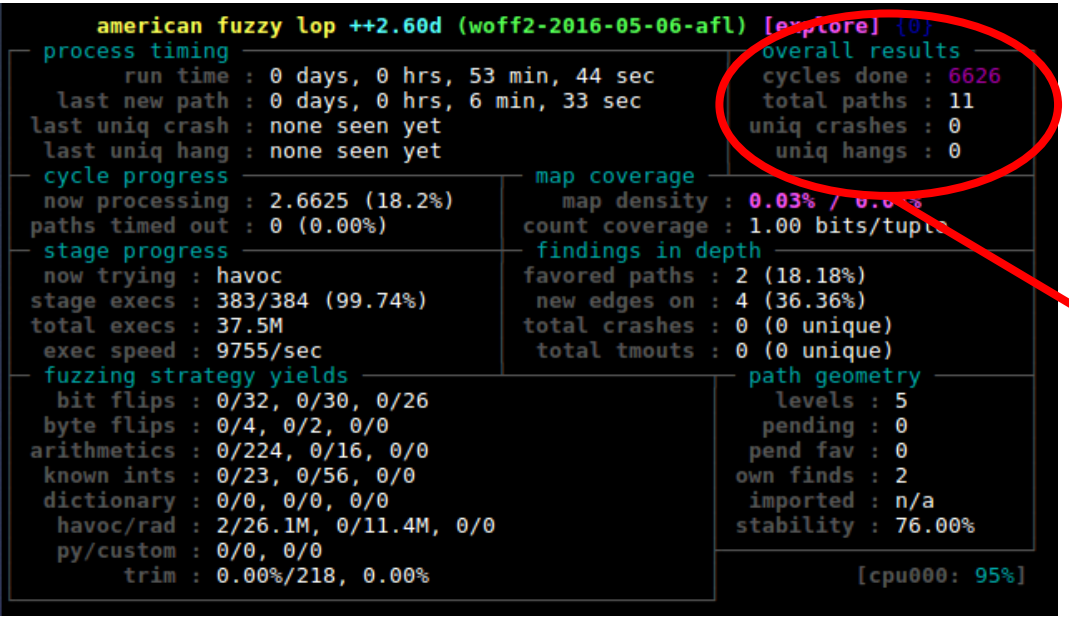
```
$ export FUZZING_ENGINE=libfuzzer
$ export CC=clang
$ export CXX=clang++
$ ./build.sh
```

The build script then clones the WOFF2 snapshot and compiles (and instruments) the source code.

Let's fuzz


```
$ afl-fuzz -i seeds/ -o CORPUS-
  woff2-2016-05-06-afl/ ./woff2-
  2016-05-06-afl
```

```
./woff2-2016-05-06-fsanitize_fuzzer
```




```
INFO: Seed: 1534267175
INFO: Loaded 1 modules   (9611 inline 8-bit counters): 9611 [0x93a710, 0x93cc9b),
INFO: Loaded 1 PC tables (9611 PCs): 9611 [0x6e67e8,0x70c098),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 15 ft: 16 corp: 1/1b exec/s: 0 rss: 37Mb
#5      NEW    cov: 16 ft: 17 corp: 2/10b exec/s: 0 rss: 38Mb L: 9/9 MS: 3 ShuffleBytes-CopyPart-CMP- DE: "\x01\x00
\x00\x00\x00\x00\x00\x00\x00"-
#8      REDUCE cov: 16 ft: 17 corp: 2/6b exec/s: 0 rss: 38Mb L: 5/5 MS: 3 ChangeBinInt-PersAutoDict-EraseBytes- DE:
 "\x01\x00\x00\x00\x00\x00\x00\x00"-
#45     REDUCE cov: 16 ft: 17 corp: 2/5b exec/s: 0 rss: 38Mb L: 4/4 MS: 2 ChangeByte-EraseBytes-
#3002   REDUCE cov: 17 ft: 18 corp: 3/9b exec/s: 0 rss: 41Mb L: 4/4 MS: 2 ShuffleBytes-CMP- DE: "wOF2"-
#3059   NEW    cov: 18 ft: 19 corp: 4/17b exec/s: 0 rss: 41Mb L: 8/8 MS: 2 CopyPart-CrossOver-
#3071   NEW    cov: 19 ft: 20 corp: 5/34b exec/s: 0 rss: 41Mb L: 17/17 MS: 2 ChangeByte-InsertRepeatedBytes-
#3377   REDUCE cov: 19 ft: 20 corp: 5/33b exec/s: 0 rss: 41Mb L: 16/16 MS: 1 EraseBytes-
#3744   REDUCE cov: 19 ft: 20 corp: 5/31b exec/s: 0 rss: 42Mb L: 14/14 MS: 2 ChangeByte-EraseBytes-
#4060   REDUCE cov: 19 ft: 20 corp: 5/29b exec/s: 0 rss: 42Mb L: 12/12 MS: 1 EraseBytes-
#5282   REDUCE cov: 20 ft: 21 corp: 6/41b exec/s: 0 rss: 43Mb L: 12/12 MS: 2 ChangeBinInt-ChangeBinInt-
```

Coverage information

Both fuzzers then try to maximize coverage by mutating interesting test cases.

BOSCH

Let’s fuzz

AFL++ fuzzes for an unlimited amount of time.

libFuzzer fuzzes until a crash is found.

Both fuzzers save a reproducible crashing file.

For our WOFF2 example, both can find a multi-byte-write-heap-buffer-overflow.

A crash looks like:

```
ERROR: AddressSanitizer: heap-buffer-overflow
WRITE of size 6707 at 0x62300000534d thread T0
    #0 0x4a95d3 in __asan_memcpy
    #1 0x62fa5c in woff2::Buffer::Read(unsigned char*, unsigned long) src/./buffer.h:86:7
    #2 0x62fa5c in woff2::(anonymous namespace)::ReconstructGlyf src/woff2_dec.cc:500
    #3 0x62fa5c in woff2::(anonymous namespace)::ReconstructFont src/woff2_dec.cc:917
    #4 0x62fa5c in woff2::ConvertWOFF2ToTTF src/woff2_dec.cc:1282
```

https://github.com/google/fuzzer-test-suite/tree/master/woff2-2016-05-06

Optimization: Sanitizers (heartbleed example)

Try running the fuzzer:

```
./openssl-1.0.1f-fsanitize_fuzzer
```

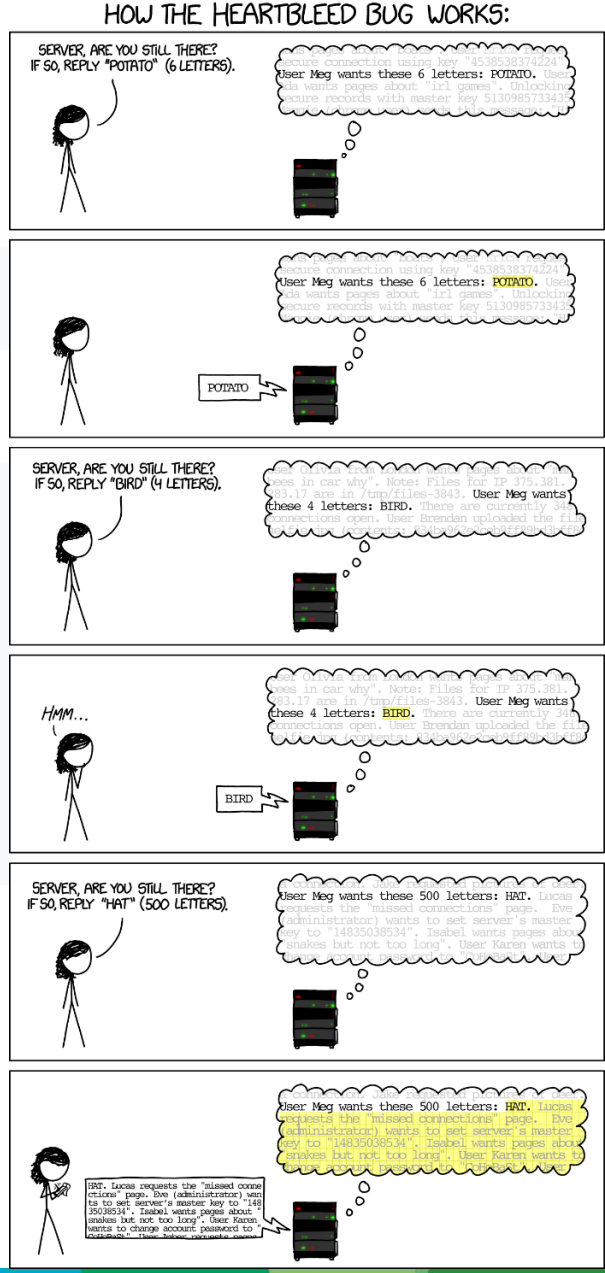
You whould see something like this in a few seconds:

```
==5781==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x629000009748 at pc 0x0000004a9817...
READ of size 19715 at 0x629000009748 thread T0
    #0 0x4a9816 in __asan_memcpy (heartbleed/openssl-1.0.1f+0x4a9816)
    #1 0x4fd54a in tls1_process_heartbeat heartbleed/BUILD/ssl/t1_lib.c:2586:3
    #2 0x58027d in ssl3_read_bytes heartbleed/BUILD/ssl/s3_pkt.c:1092:4
    #3 0x585357 in ssl3_get_message heartbleed/BUILD/ssl/s3_both.c:457:7
    #4 0x54781a in ssl3_get_client_hello heartbleed/BUILD/ssl/s3_srvr.c:941:4
    #5 0x543764 in ssl3_accept heartbleed/BUILD/ssl/s3_srvr.c:357:9
    #6 0x4eed3a in LLVMFuzzerTestOneInput FTS/openssl-1.0.1f/target.cc:38:3
```

Sanitizers ‘provoke’ a crash on certain behaviour, to make certain bug types detectable for fuzzers, e.g. for a reading buffer overflow.

https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md

https://xkcd.com/1354/



Optimization: Seeds

Seeds are initial (small and valid) test cases, so that the fuzzer does not have to start from thin air.

In our example the build.sh downloads the Roboto font as seed.


```
$ afl-fuzz -i seeds/ -o CORPUS-
  woff2-2016-05-06-afl/ ./woff2-
  2016-05-06-afl
```

```
$ ./$woff2-2016-05-06-fsanitize_fuzzer
  CORPUS seeds
```

Optimization: Dictionaries

Dictionaries help the fuzzer by replacing part of the test case by a dictionary entry, rather than e.g. random. Dictionary entries should be often used symbols and words by the target software.

There are multiple pre-built dictionaries available, e.g. for SQL, XML, JSON, ...


```
$ -x dict=DICTIONARY_FILE
```


```
$ -dict=DICTIONARY_FILE
```

Optimization: Parallelization

Run first fuzzer as ‘master’ **–M**

```
$ ./afl-fuzz -i seeds -o sync_dir
  -M fuzzer01 [...]
```

then, start up secondary instances

```
$ ./afl-fuzz -i seeds -o sync_dir
  -S fuzzer02 [...]
$ ./afl-fuzz -i seeds -o sync_dir
  -S fuzzer03 [...]
```

Each fuzzer will keep its state in a separate subdirectory in `sync_dir`, and the master syncs all fuzzing instances.

Run multiple libfuzzer processes in parallel with a shared corpus directory.

$JOBS is by default half of available CPU cores

```
$ ./$woff2-2016-05-06-fsanitize_fuzzer
  CORPUS -workers=$JOBS CORPUS
```

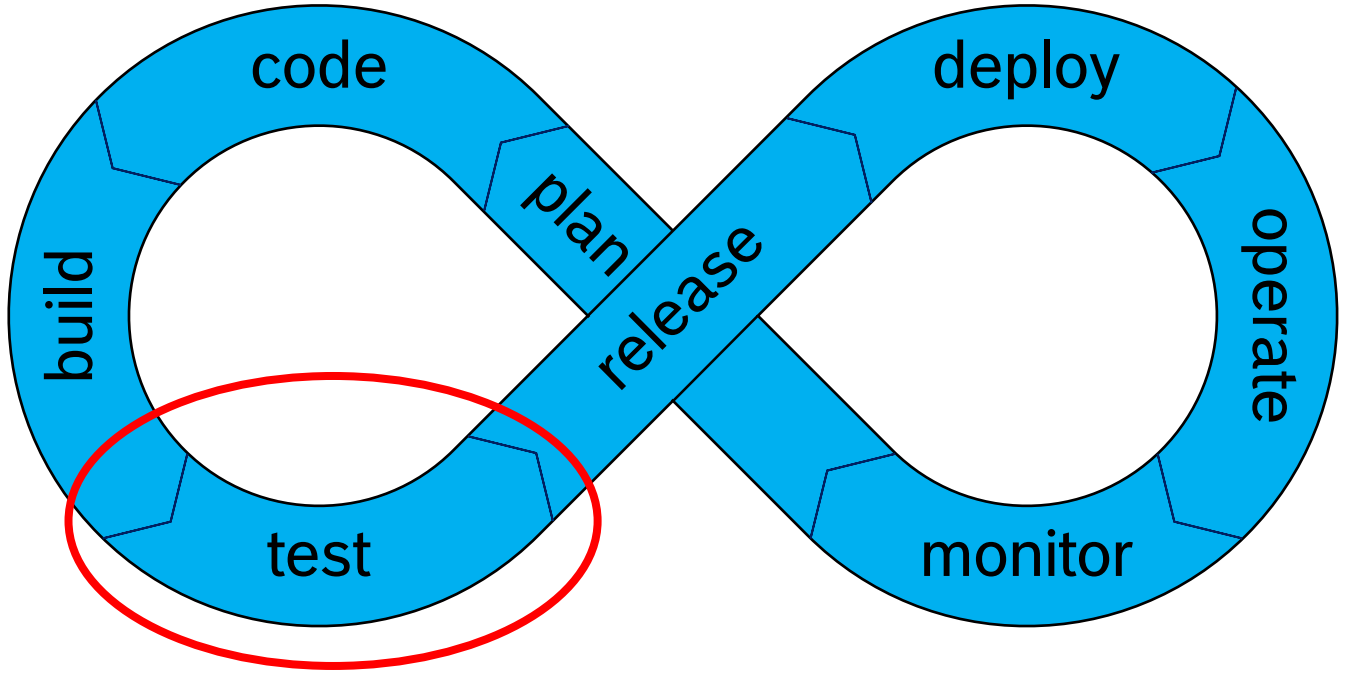
Challenges

- Find a suited fuzz target.
  - E.g. an API function, which consumes untrusted input under the control of a potential attacker.
- Write a fuzz test.
  - Connecting the software under test to the fuzzing engine is manual work.
  - E.g. wrapper connecting a single function to the fuzzing engine, fuzz a complete running system, or concentrate on parts of the system.
- Fuzzing results should be observable.
  - E.g. crashes in black-box fuzzing could be hard to detect.
  - Instrumentation can be hard (different compilers, debug vs. productive software)
- Speed up your fuzzing, as it relies on thousands of test case executions.
  - Keeping the current test case corpora at a relevant minimum.
  - Parallelize your fuzz tests while working on the same test corpora (synchronize and do regular clean-ups).
- Provide a useful structure of the input.
  - Grammar, dictionary, or seed
- No fixed value for timeout.
  - Typical tests vary from hours over days to weeks.

Good Practices

- Don’t fuzz everything.
  - The fuzz target should consume input which is under control of a potential attacker.
- Fuzz at least for a realistic threat, optionally dig deeper.
  - E.g. HW system level testing: Fuzz components at least over the bus, optionally fuzz component’s internals
  - E.g. SW component level testing: Fuzz components over interface, optionally fuzz internal methods
- Validate your programmed fuzz test before fuzzing.
  - The fuzz test should consume the generated input, the fuzz test should not crash for valid inputs, and code coverage tracking of the fuzzer should work.
  - Design your tests for testability and observable results
- Use different fuzzers, potentially working on the same test corpora, in parallel.
- Provide an input structure for the fuzzer.
  - Such as a grammar, dictionary or seed. Typically, there exist prebuilt structures for fuzzing engines, which should be used and refined.

Fuzzing in DevOps



- Integrate fuzzing in your test stack.
- Best, fuzz independently, e.g. over night or over weekend.
- In the test pyramid, fuzz tests can be small (unit), medium (integration), and large (system) tests.

Thank you!

More:
Automated security testing to provide more protection from the start
Automated software testing by Bosch

https://www.bosch.com/stories/automated-security-testing/

BOSCH

Dr.-Ing.
Christopher Huth

Corporate Sector Research and Advance Engineering
Security, Privacy, Safety

christopher.huth@de.bosch.com