Automotive Fuzzing

2025.08

자동차융합대학

GENERAL MOTORS
GM TECHNICAL CENTER KOREA

KMU 국민대학교
KOOKMIN UNIVERSITY

CONTENTS

01 **Brute Force Vulnerability Detection**
- Fuzzing Overview

02 **Automotive Fuzzing**
- Challenges and Insights

KMU 국민대학교 KOOKMIN UNIVERSITY

01

Brute Force Vulnerability Detection

Techniques for vulnerability detection

Dynamic Analysis → **Static Analysis**

Fuzzing

Symbolic Execution
Taint Analysis

Static rule-based detection

Lower coverage
Lower false positive
Higher false negative

Higher coverage
Higher false positive
Lower false negative

Fuzzing

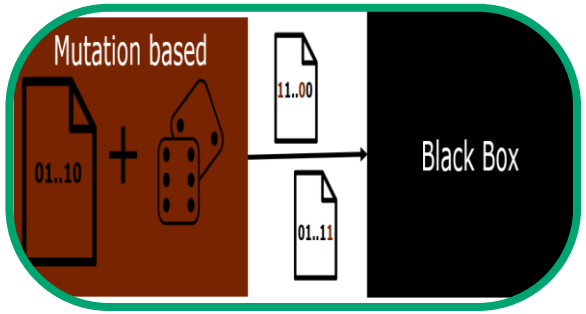
- Fuzzy + Testing ➔ Fuzzing
- *Automatically generate* test cases
- Many *slightly anomalous test cases* are input into a target
- Application is *monitored* for errors.
- Inputs are generally either *file based* (pdf, png, wav, etc.) or *network based* (http, SNMP, etc.)

Fuzzing Evolution Timeline

Dumb Fuzzing

Random input generation without format awareness

Barton Miller et al. '89

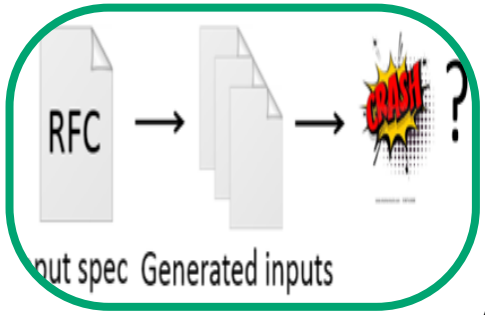


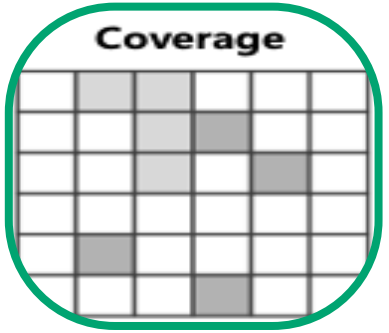
Mutation based Fuzzing

Modify existing inputs

Generation based Fuzzing

Generate from protocol/format





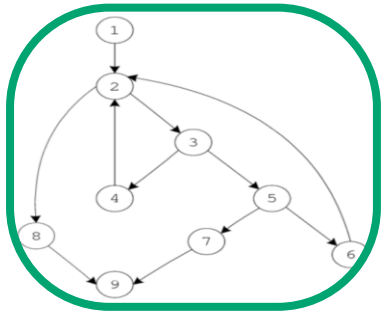
Coverage guided Fuzzing

Feedback from code coverage

American Fuzzy Lop (AFL)

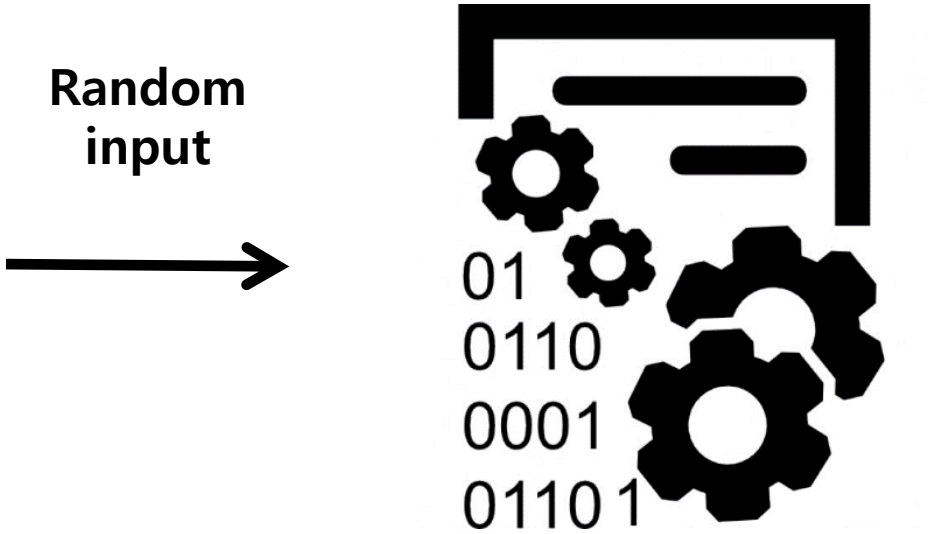
Dataflow guided Fuzzing

Leverages internal dataflow to refine fuzzing



Blackbox Fuzzing

Random input



Test Program

Barton Miller et al. ‘89

Blackbox Fuzzing

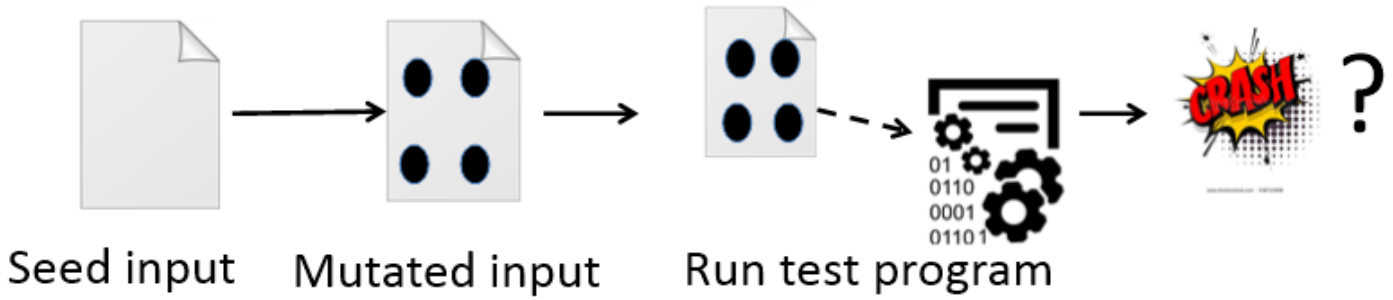
- **Given a program *simply feed random inputs* and see *whether it exhibits incorrect behavior* (e.g., crashes)**

- **Advantage: *easy, low programmer cost***

- **Disadvantage: inefficient**
  - Inputs often require structures; random inputs *are likely to be malformed*
  - Inputs that trigger an incorrect behavior is a *very small fraction*, probably of *getting lucky is very low.*

Mutation-based Fuzzing

- Take a ***well-formed input, randomly perturb*** (flipping bit, etc.)
- ***Little or no the structure knowledge*** of the inputs is assumed.
- Anomalies are ***added to existing valid inputs***
  - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Mutation-based Fuzzing

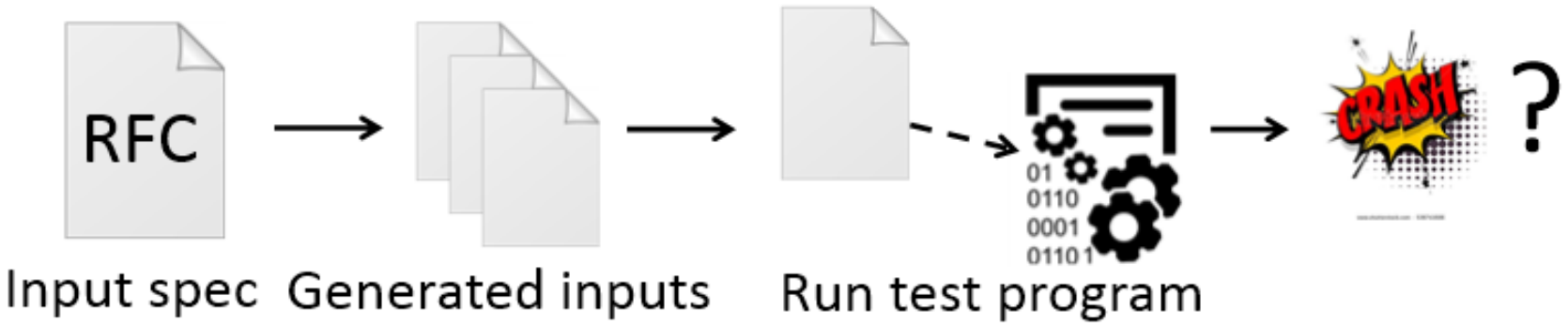
- Super easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus (seed)
- May fail for protocols with checksums, those which depend on challenges

Generation-based Fuzzing

- **Test cases are generated from some description of *the input format*: RFC, documentation, etc.**
  - Using specified protocols/file format information.

- **Anomalies are added to *each possible spot* in the inputs**

- **Knowledge of protocol should give *better results than random fuzzing***



Mutation-based vs. Generation-based

■ Mutation-based fuzzer

- Pros: easy to set up and automate, little to no knowledge of input format required
- Cons: limited by initial corpus, may fall for protocols with checksums and other hard checks

■ Generation-based fuzzer

- Pros: completeness, can deal with complex dependencies (e.g., checksum)
- Cons: writing generators is hard, performance depends on the quality of the spec.

How much fuzzing is enough?

- **Mutation-based fuzzers may generate an *infinite number of test cases*. When has the fuzzer run long enough?**

- **Generation-based fuzzers may generate a *finite number of test cases*. What happens when they are all run, and no bugs are found?**

How much fuzzing is enough?

■ **Properties of Security Assessment (test)**

- Assessment resources (test engineer, time, etc.) are finite.
- The security assessment does not guarantee that the target project is perfectly secure. Instead, it ensures the target project is secure in specific scenarios (test case).

Need to improve test efficiency
(speed, coverage, etc.)

Code coverage

- **Some of the answers to these questions lie in code coverage.**

- **Code coverage is a metric that can be used to determine *how much code has been executed*.**

- **Coverage data can be obtained using a variety of profiling tools.**
  - E.g., gcov, lcov

Code Coverage – Line Coverage

■ **Line/block coverage**

- measures how many lines of source code have been executed.

■ **For the code on the right, how many test cases (values of pair (a, b)) Needed for full (100%) line coverage?**

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```


Code Coverage – Branch Coverage

■ **Branch coverage**

- Measures how many branches in code have been taken (conditional jmps)

■ **For the code on the right, how may test cases needed for full branch coverage?**

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```


Code Coverage – Path Coverage

■ **Path coverage**

- Measures how many paths have been taken

■ **For the code on the right, how many test cases needed for full path coverage?**

```
if( a > 2 )
    a = 2;
if( b >2 )
    b = 2;
```


Benefits of code coverage

■ **Can answer the following questions**

- How good is an initial file?
- *Am I getting stuck somewhere?*
- How good is fuzzer X versus fuzzer Y?
- Am I getting benefits by running multiple fuzzers?

Problems of code coverage

- Does full line coverage guarantee find the bug?
- Does full branch coverage guarantee find the bug?
- Does full path coverage guarantee find the bug?

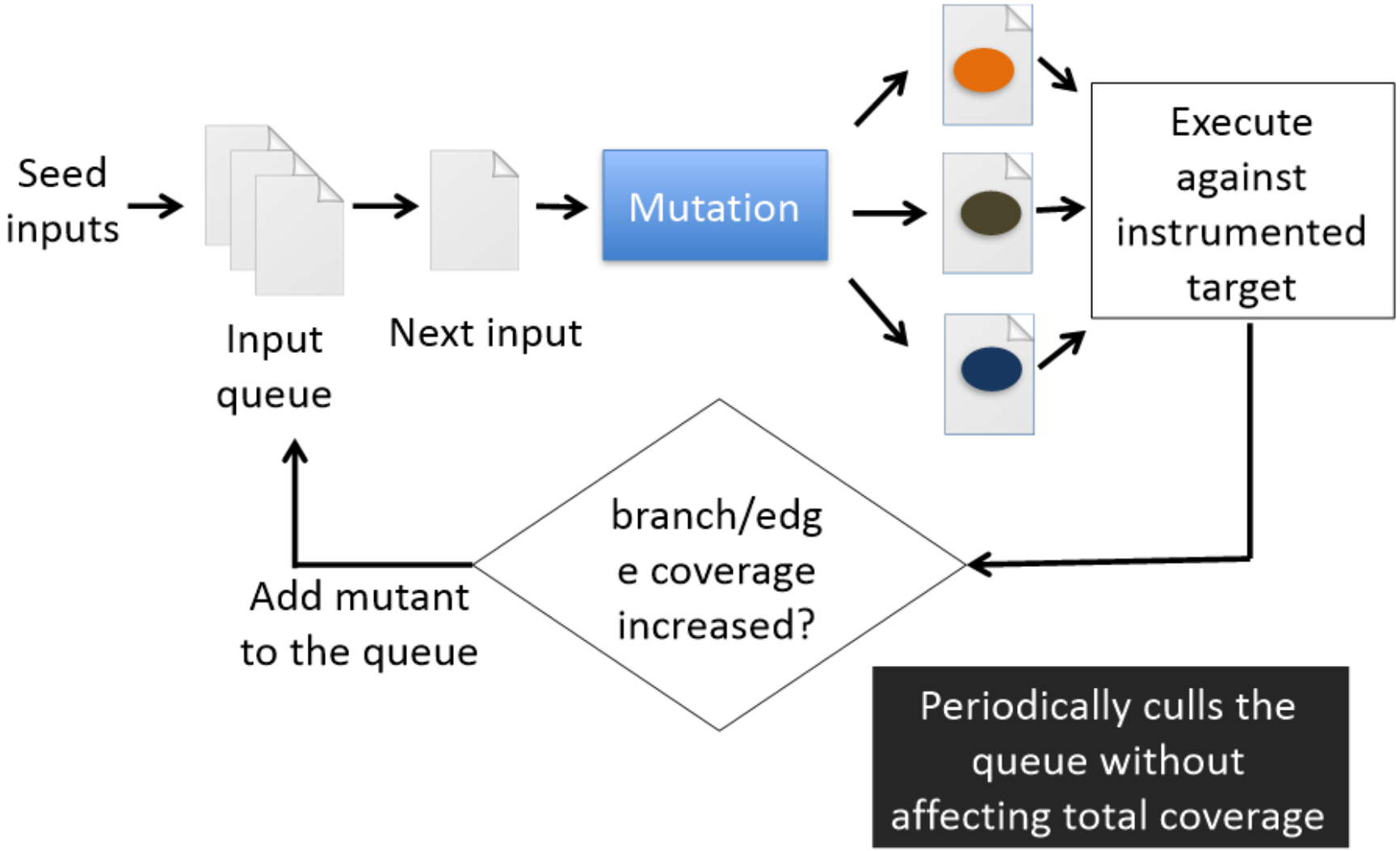
```
void mySafeCopy(char *dst, uint32 dstLen, char* src, uint32 srcLen)
{
        if (dstLen <= 0 || srcLen <= 0) return;
        if (dst && src) strcpy(dst, src);
}
```

Coverage-guided Graybox Fuzzing

■ Special type of mutation-based fuzzing

- Run mutated inputs on instrumented program and measure code coverage
- Search for *mutants that result in coverage increase*
- Often use genetic algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases.
  - E.g., AFL, Libfuzzer

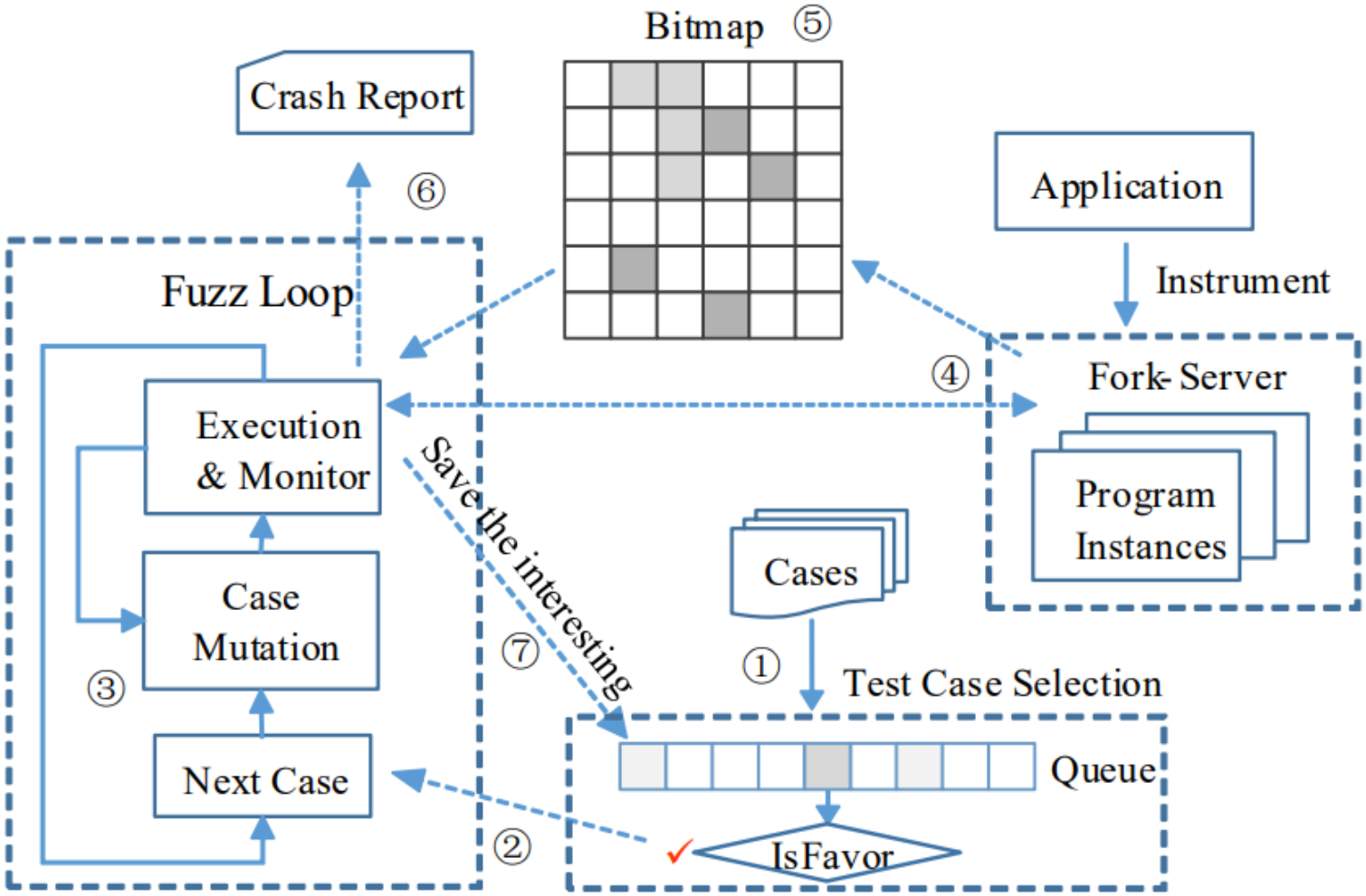
American Fuzzy Lop (AFL)



AFL

- **Instrument the binary at compile-time → gray-box fuzzing**
- **Regular mode: instrument assembly**
- **Recent addition: LLVM compiler instrumentation mode (faster than regular mode)**
- **Provide 64K counters representing all edges in the app**
- **Hash table keeps track of # of execution of edges**
  - 8 bits per edges (# of executions: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+)
  - Imprecise (edges may collide) but very efficient → CollAFL
- **AFL-Fuzz is the driver process, the target app runs as separate process(es).**

AFL Fuzzing Process



Wang, Yanhao, et al. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization."

AFL Fuzzing Process (cont'd)

- 2가지 단계로 구성
  - Instrumentation Phase
    - Preliminary analysis
    - Runtime execution feedback
  - Fuzzing Loop Phase
    - Input prioritization
- 용어 정리
  - ***Queue*** : sample queue
  - ***CrashSet*** : a set of crashes
  - ***CovFb, AccountingFb*** : coverage/accounting feedback map
  - ***accCov, accAccounting*** : global accumulated structures to hold all covered transitions and their maximum hit counts for each feedback map
  - ***TopCov, TopAccounting*** : a hash map with edges as keys and inputs as values to maintain the best input for each edge

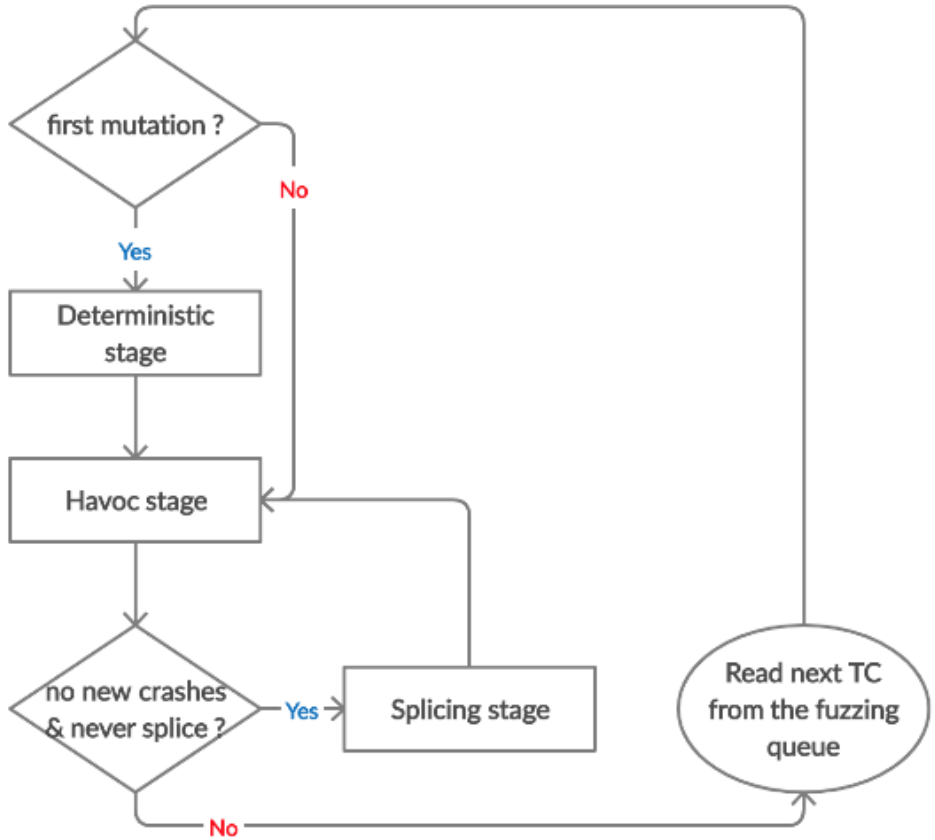
Favored 값이 지정된 seed는 100%, 아닌 경우 1% 확률로 퍼징

Algorithm 1 Fuzzing algorithm with coverage accounting

```
1:  function FUZZING(Program, Seeds)
2:      P ← INSTRUMENT(Program, CovFb, AccountingFb)        ▷ Instr. Phase
3:      // AccountingFb is FunCallMap, LoopMap, or InsMap
4:      INITIALIZE(Queue, CrashSet, Seeds)
5:      INITIALIZE(CovFb, accCov, TopCov)
6:      INITIALIZE(AccountingFb, accAccounting, TopAccounting)
7:      // accAccounting is MaxFunCallMap, MaxLoopMap, or MaxInsMap
8:      repeat                                              ▷ Fuzzing Loop Phase
9:          input ← NEXTSEED(Queue)
10:         NumChildren ← MUTATEENERGY(input)
11:         for i = 0 → NumChildren do
12:             child ← MUTATE(input)
13:             IsCrash, CovFb, AccountingFb ← RUN(P, child)
14:             if IsCrash then
15:                 CrashSet ← CrashSet ∪ child
16:             else if SAVE_IF_INTERESTING(CovFb, accCov) then
17:                 TopCov, TopAccounting ←
18:                     UPDATE(child, CovFb, AccountingFb, accAccounting)
19:                 Queue ← Queue ∪ child
20:             end if
21:         end for
22:         CULL_QUEUE(Queue, TopCov, TopAccounting)
23:     until time out
24: end function
```

AFL Fuzzing Process – Mutation Scheduling

- **The first time to mutate this test case ?**
- **If yes, Deterministic stage**
  - bitflip, byteflip, arithmetic inc/dec, interesting values,
  - auto extras, user extras.
- **If no, Havoc stage**
  - bitflip, byteflip, arithmetic inc/dec, interesting values,
  - random byte, delete bytes, insert bytes, overwrite bytes
- **AFL mutate all the TCs in the queue but discovers no crashes or path && this TC has not entered splicing stage for this time ?**
- **If yes, Splicing stage**
  - cross over
- **If no, read next TC from the fuzzing queue**



Fuzzing Challenges

- **How to seed a fuzzer?**
  - Seed inputs must cover different branches
  - Remove duplicate seeds covering the same branches
  - Small seeds are better → Why?

- **Some branches might be very hard to get past as the # of inputs satisfying the conditions are very small**
  - Manually / automatically transform / remove those branches

Fuzzing Challenges (cont'd)

- **Hard to fuzz code**

```
void test (int n) {
  if (n==0x12345678)
     crash();
}
```

needs 2^32 or 4 billion attempts
In the worst case

Fuzzing Challenges (cont'd)

- The probability to generate feasible inputs is very low

```
1  int main() {
2    unsigned v, w, x, y, z = input();
3    if(x == 3 && y == 4 && z*z < 40000) {
4      if (z > 195) {
5        ......
6        // crash 1
7      }
8      if(15 <= z+v && z+v <= 25) {
9        ......
10       // crash 2
11     }
12   }
13 }
```

Line 3:

$$x==3 \;\&\&\; y==4 \;\&\&\; z*z < 40000$$

seed 1: (v, x, y, z) = (100, 3, 4, 20)

The probability for a mutated input satisfying the condition at Line 3:

$$\underset{<10^{-9}}{x}\quad \underset{<10^{-9}}{y}\quad \underset{<10^{-7}}{z}$$

$$1/2^{32} * 1/2^{32} * 200/2^{32} < 10^{-25}$$

Dataflow guided Fuzzing

- **A.K.A., Hybrid fuzzing**
  - *Combine* coverage-guided fuzzing *with taint-analysis or symbolic execution*, and so on.

- ***Intercept the data flow*, analyze the inputs of comparisons**
  - Incurs extra overhead

- **Modify the test inputs, observe the effect on comparisons**

- **Prototype implementations in T-fuzz, and VUzzer.**

Fuzzing rule of thumb

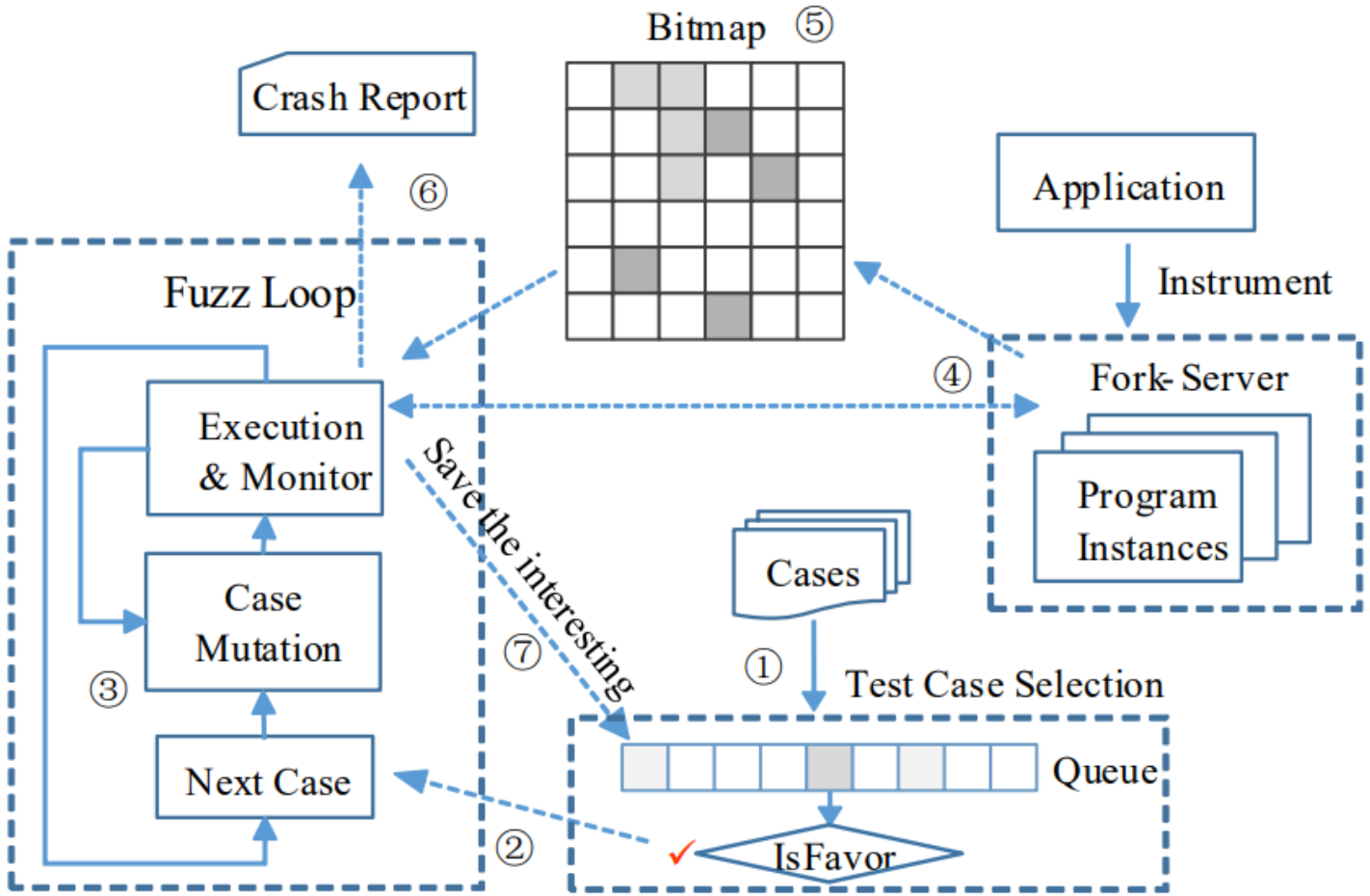
- **Input-format knowledge is very helpful.**
- **Generational tends to beat random, better specs make better fuzzers.**
- **Each implementation will vary different fuzzers find different bugs.**
  - More fuzzing with is better → EnFuzz
- **The longer you run, the more bugs you may find.**
  - But it reaches a plateau and saturates after a while
- **Best results come from guiding the process.**
- **Notice where you are getting stuck, use profiling (gcov, lcov)!**

Improvements for fuzzer

■ Instrumentation

- Execution Feedback
  - Where feedback ?
    - ✓ Source code, Binary Code
  - What feedback?
    - ✓ Edge coverage (Branch coverage)
    - ✓ Condition/Decision, MCDC …
  - How new path ?
    - ✓ Hybrid
    - ✓ Symbolic Execution, Taint Analysis

- In-memory (Persistent) Fuzzing
  - When fork ?
    - ✓ Enable to skip start up code
    - ✓ Need to skip initial I/O routine
  - How reproduce ?
    - ✓ Hard to reproduce error



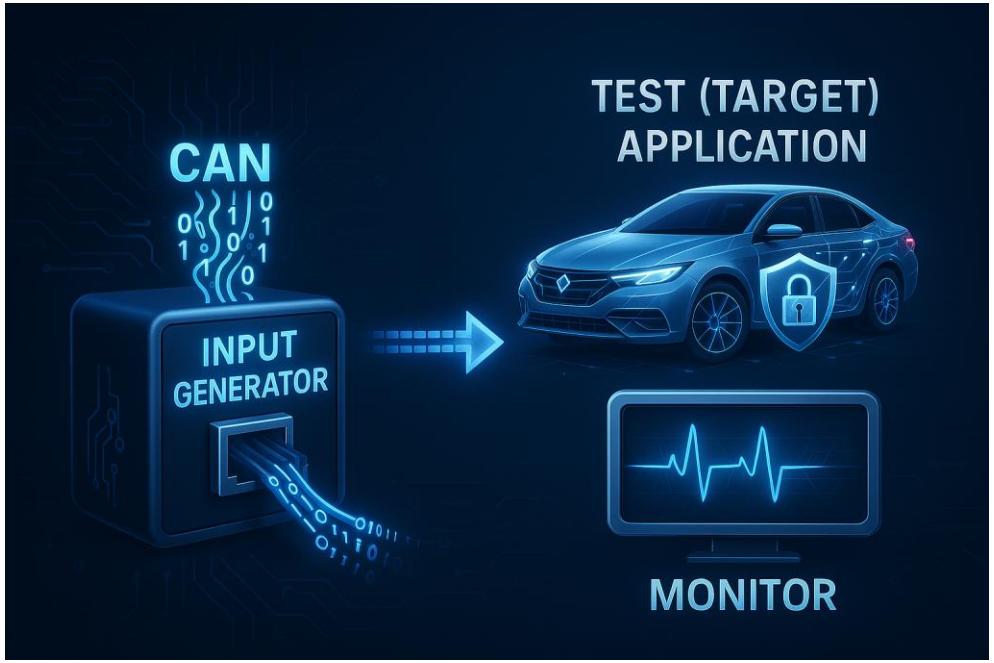
02

Automotive Fuzzing

Automotive Fuzzing Goal

■ **The ultimate goal is to build a reliable and repeatable fuzzing framework that accelerates vulnerability discovery on real vehicle.**

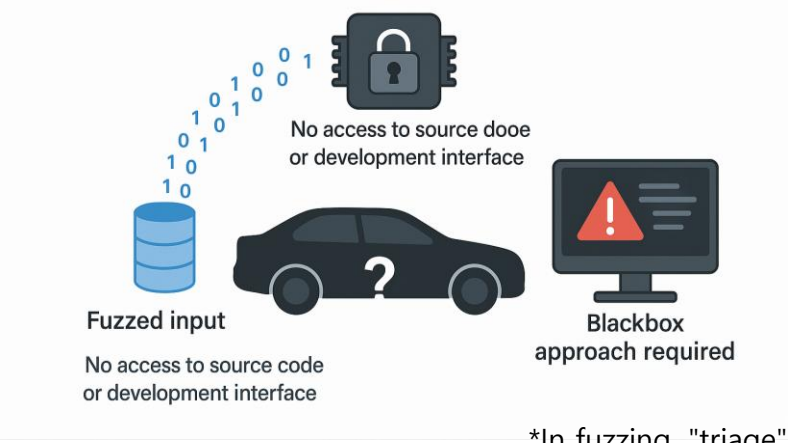
- Enable autonomous and scalable fuzzing directly on real vehicles
- Support adaptive testcase refinement using observed CAN responses
- Provide fine-grained failure analysis without halting the vehicle



Why Blackbox Fuzzing

■ **Automotive fuzzing requires a unique black-box strategy due to limited access to internal ECU mechanics.**

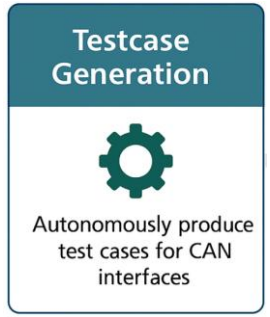
- No access to source code or dev ECUs → black-box fuzzing is required.
- Blackbox fuzzing for vehicle is not easy
  - Coverage-guided fuzzing cannot be performed due to the lack of source code.
- Can't request *triage in detail
  - It is impossible to connect directly to the ECU.
  - The only way testers obtain information from cars is through OBD-II ports or harnessed CAN lines.



*In fuzzing, "triage" refers to the process of prioritizing discovered bugs or vulnerabilities and evaluating their importance or severity.

3 Major challenges in Automotive fuzzing

1. **SMART testcase generation (Feedback)**
   - How to generate effective test cases when coverage-guided fuzzing is not possible.



2. **FAIL detection (Monitoring)**
   - How to monitor the target and detect issues.

3. **Reset (Automated)**
   - Even if the target crashes, fuzzing must continue. How to automatically reset, reboot, and initialize the target?

Insights #1 – Testcase Generation

1. **Generate testcases only for the available UDS services**

2. **Transmit a testcase complying with the message sequences**

3. **Transmit multiple frames when a testcase is large**

→ For effective fuzzing, Fuzzer should consider the above UDS CAN features.

■ Testcase Generation Basis

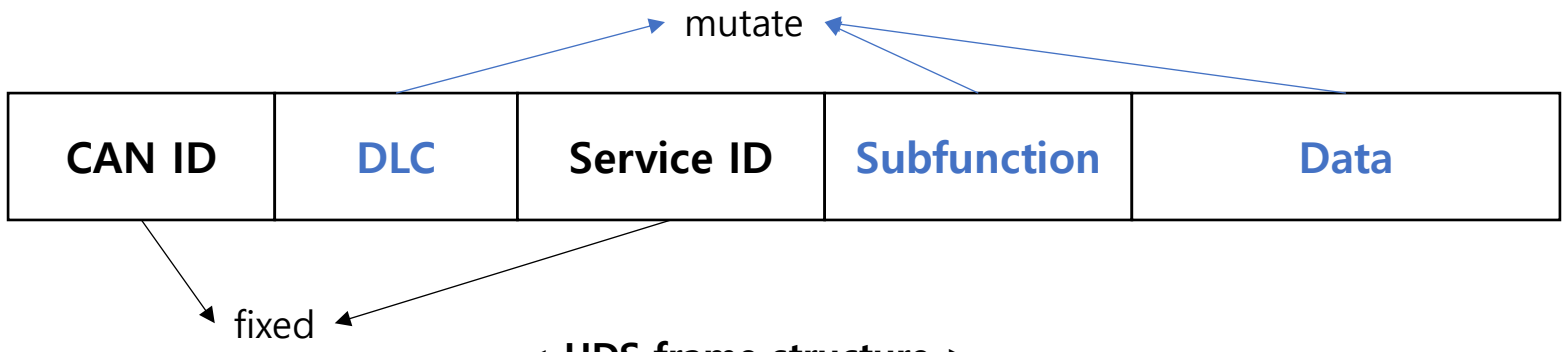
- Fuzzer sets the target ECU and generates testcases for each UDS service.

  → Fixing CAN ID and service ID

  → Mutating other fields.

- Mutating CAN ID/Service ID could be a strategy, but it is not effective.

  → Most ECUs filter wrong CAN ID and service ID



< UDS frame structure >

■ Generate Testcase only for Available Services

- There are total 26 UDS services.
- Fuzzer doesn't need to generate testcases for all services. Usually only several services are available in the ECU.
- It is efficient to generate testcases only for the available services in the target ECU and test them.

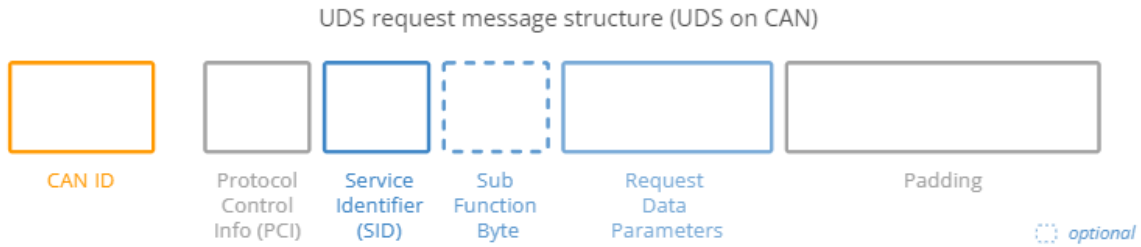
No	Service ID	Service Name
1	0x10	DiagnosticSessionControl
2	0x11	ECUReset
3	0x27	SecurityAccess
..	…	…
24	0x36	TransferData
25	0x37	RequestTransferExit
26	0x38	RequestFileTransfer

< UDS Services >

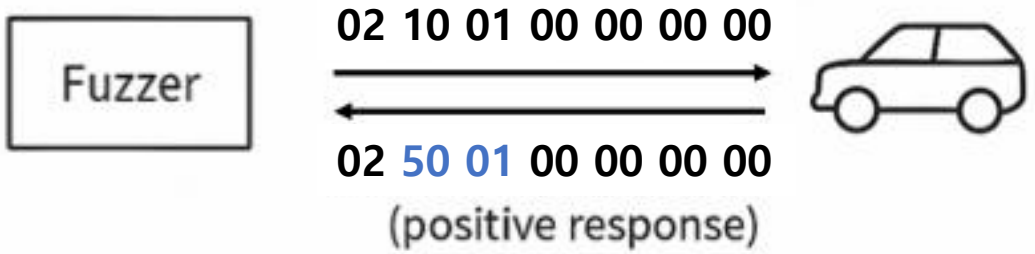
■ Check Available UDS Services

- **Before start fuzzing, Fuzzer should check the available services on the target ECU.**
- **Steps**
  1. Send a valid message of each service
  2. Check the response
  3. Decide the availability of the service depends on the response
     a. Positive response → Available
     b. No response → Unavailable (retry may be required)
     c. Negative Response → Depends on the NRC (Negative Response Codes)

■ Check Availability Cases

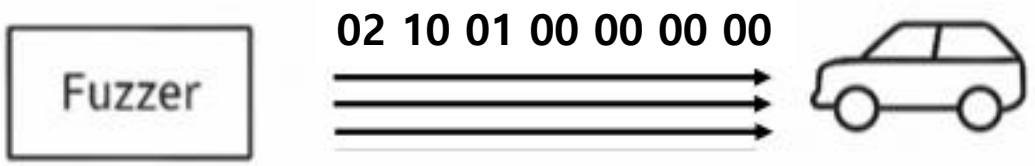


Positive response → Available



Diagnostic Session Control is available!

No Response → Unavailable



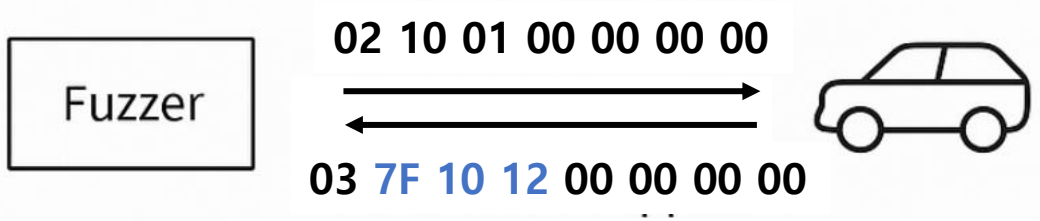
Diagnostic Session Control is unavailable!

■ Check Availability Cases

UDS SID 0x7F - Negative Response Codes (NRC)

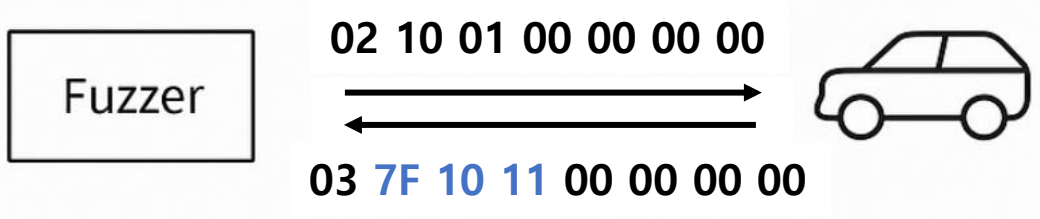
UDS NRC	Description
0x10	General reject
0x11	Service not supported
0x12	Sub-function not supported
0x13	Invalid message length/format

NRC → Fuzzer should decide depending



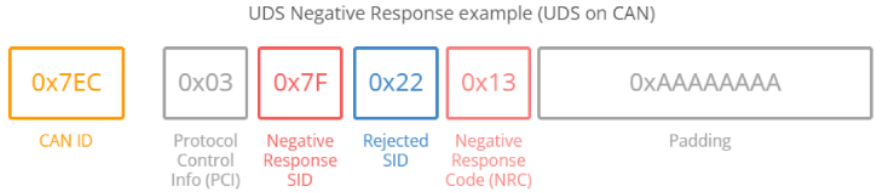
(NRC 0x12: SubFunctionNotSupported)

Diagnostic Session Control is available!



(NRC 0x11: ServiceNotSupported)

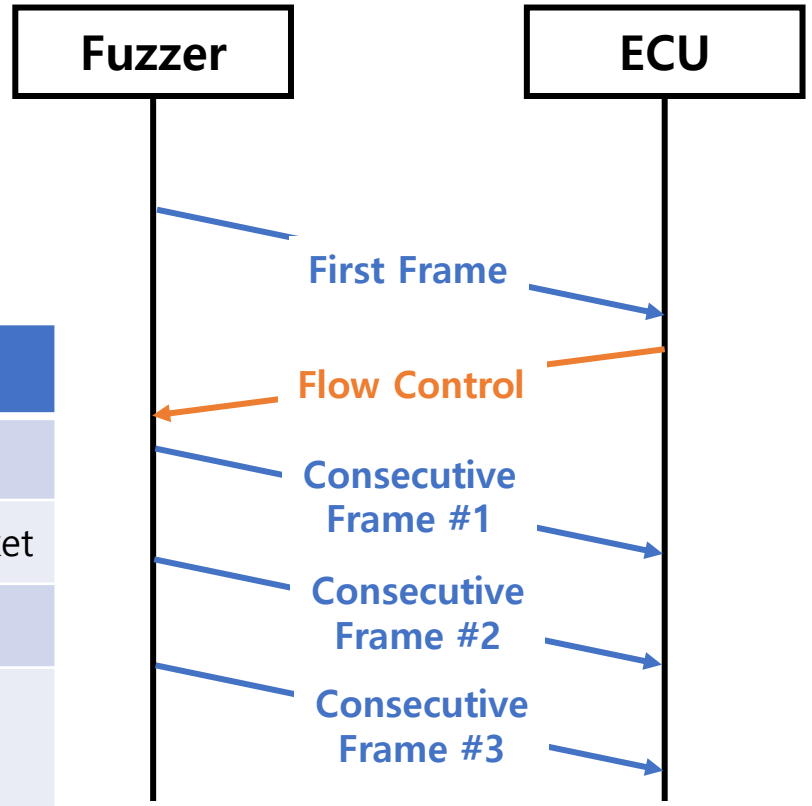
Diagnostic Session Control is unavailable!



■ Frame Types

- If testcase payload exceeds 8 bytes, the testcase should be transmitted by ISO-TP.
  - ISO-TP(ISO-15765-2)
    - ✓ The protocol for transmitting messages that exceed the 8bytes

Type	Description
Single Frame	Complete payload of up to 6, 7 bytes
First Frame	First frame of multi-frame packet with the length of the full packet
Consecutive Frame	A frame containing subsequent data
Flow Control Frame	The response of the first frame with the parameters for the consecutive frame

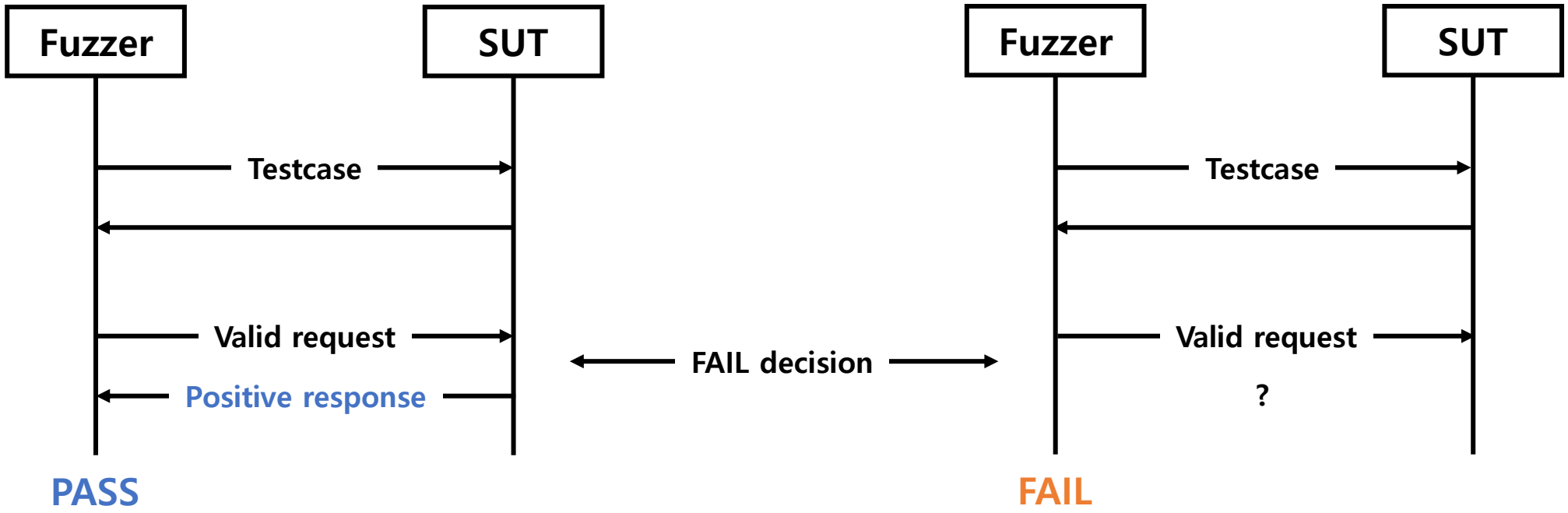


Insights #2 – FAIL detection

*SUT: System Under Test

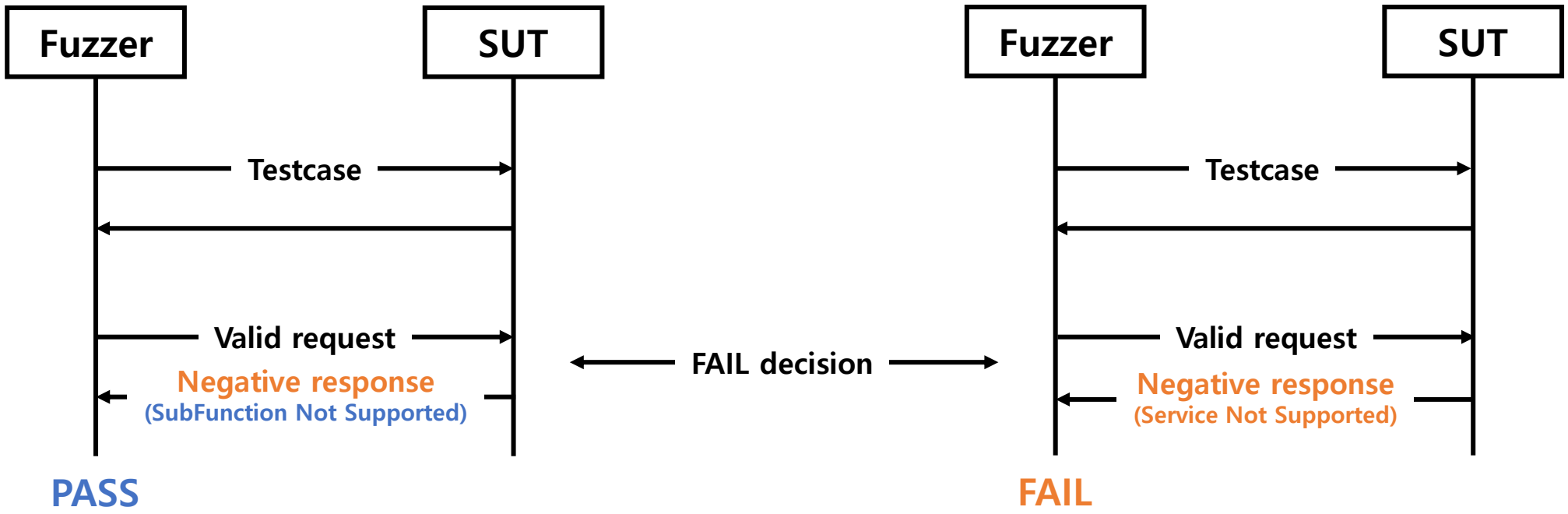
■ No response

- The fuzzer sends a valid request after testcase transmission to check the state of the SUT.
- Here, the valid request is the same message used during the check availability process.
- If there is no response during a 50-millisecond, the fuzzer reports a FAIL.



■ Negative Response

- If there is a negative response to the valid request, the fuzzer reports a FAIL depending on the NRC.

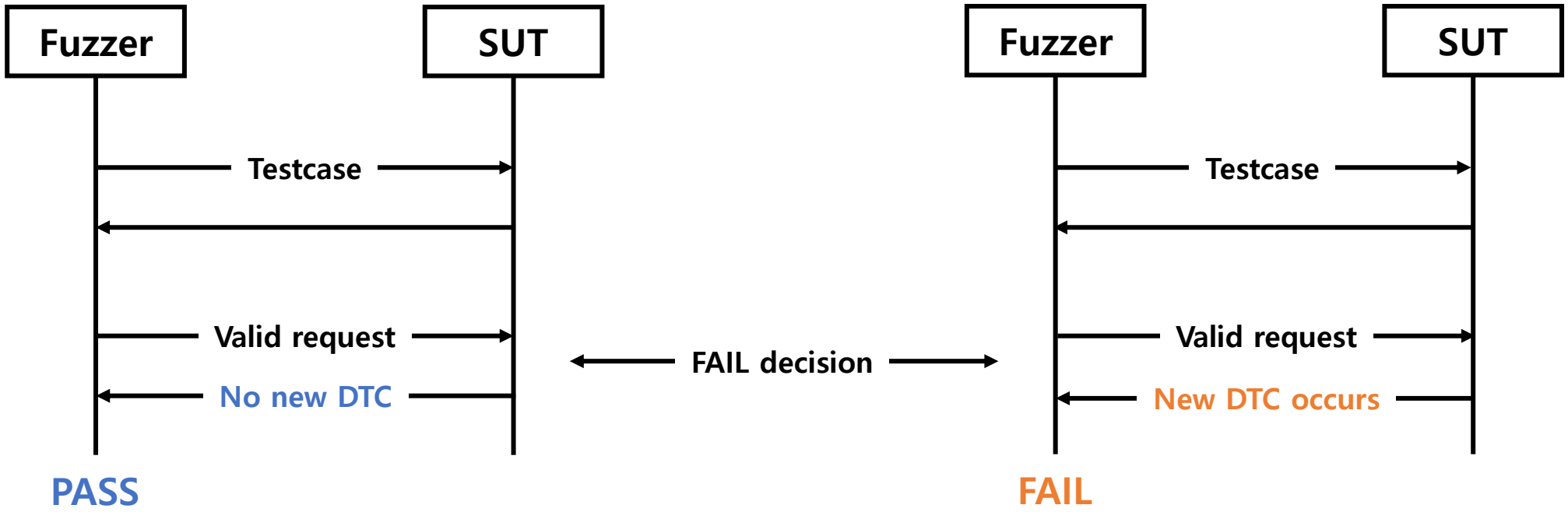


■ Diagnostic Trouble Code (DTC)

- DTC occurrence → FAIL

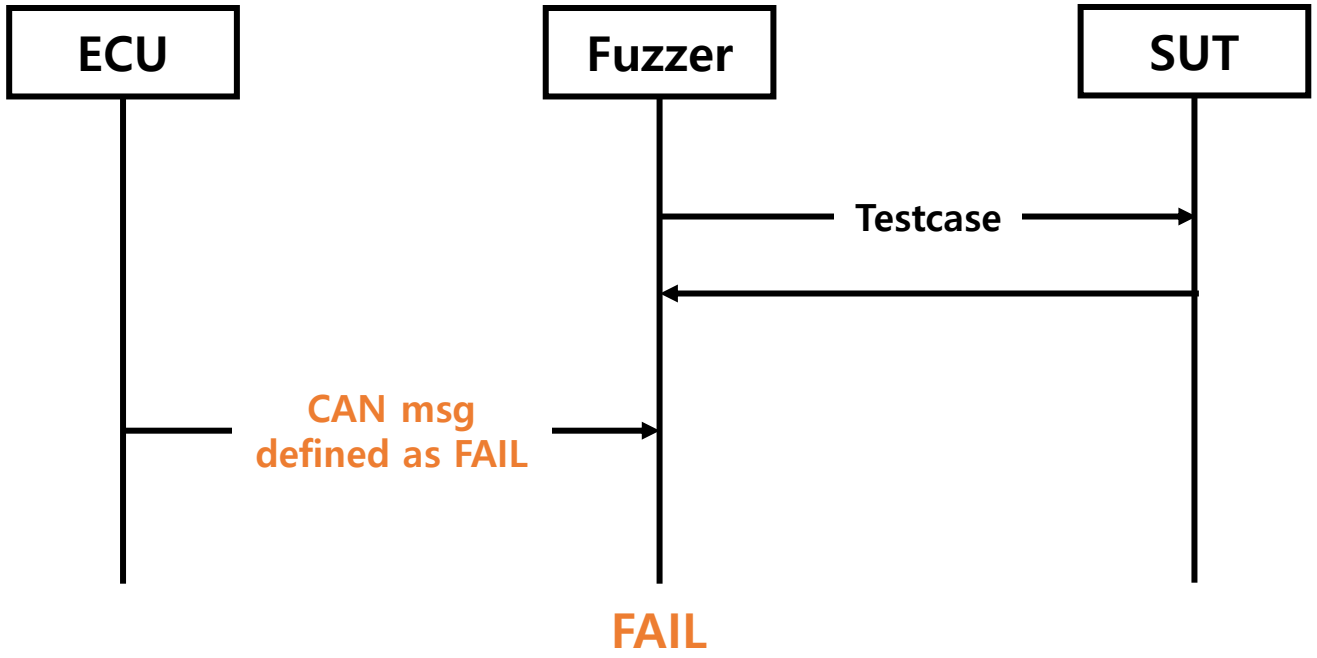
P0130

OBD-II DTC code example



■ User-specified CAN Message Occurrence

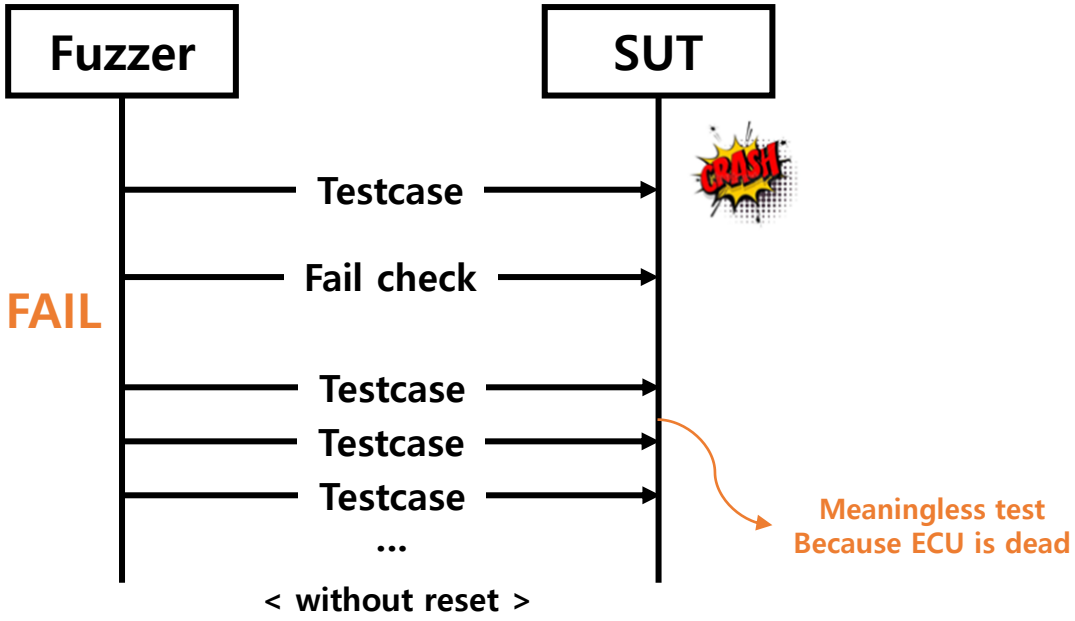
- The tester can specify a certain CAN message occurrence as a FAIL. If the certain CAN message occurs, the fuzzer reports a FAIL.
  - The fuzzer must be able to monitor the CAN bus on which the CAN message is transmitted.

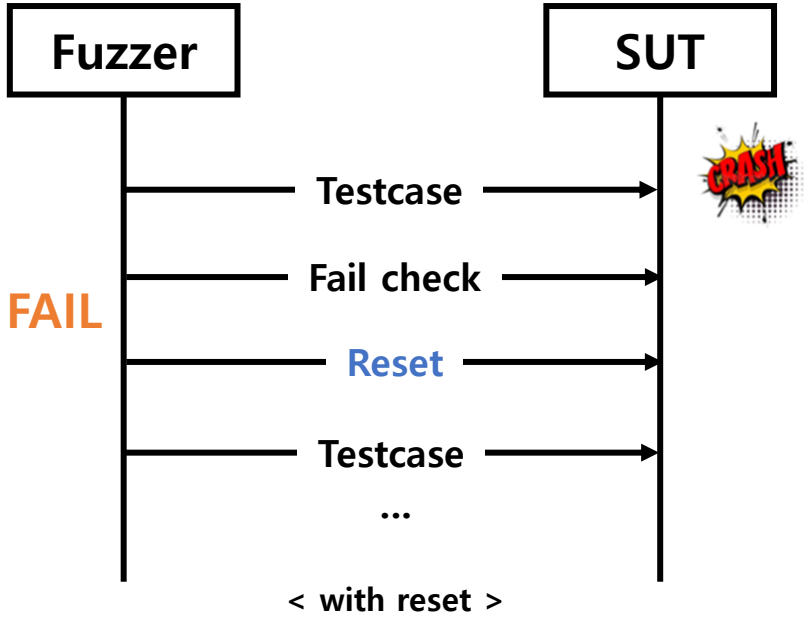


e.g. When we perform a fuzzing test on the Head Unit ECU, we can define the occurrence of the following CAN messages as a FAIL, such as Brake Pressed or Accelerator Pedal Pressed.

■ Fuzzing Automation

- When a FAIL occurs, the SUT must be initialized to continue fuzzing.
- The tester can manually initialize the SUT, but it's a very tough task to perform each time a FAIL occurs.
  - Since testers must continuously monitor the SUT while fuzzing is in progress, automatic SUT reset is required.





■ How to Perform a Reset?

- **Use Two ECU Services**
  - 0x11 and 0x14 can be used for resetting.
  - By using these services, you can trigger different types of resets.

- **Reset Types (Based on Subfunction)**
  - 0x01 (Hard Reset): A full system reset (recommended based on experience).
  - 0x03 (Soft Reset): Only restarts the application program.

Reset ECU

ECUReset (0x11) – 02 11 01 00 00 00 00 00 00

▶ Reset ECU

Byte Value	Name	Description
0x01	hardReset	Perform Power-on/Start-up
0x03	softReset	Restart the application program

ClearDiagnosticInformation (0x14) – 04 14 FF FF 00 00 00

▶ Clear all DTC related data

- **Clear Diagnostic Information**
  - When a new DTC is detected and reported as a FAIL, use a specific service to clear all DTC-related data.

Key Takeaways

1. Fuzzing is one of the efficient dynamic analysis techniques used to detect vulnerabilities.

2. Fuzzing has evolved in the following order:
   - Dumb fuzzing → Mutation-based fuzzing → Generation-based fuzzing → Coverage-guided fuzzing → Dataflow-guided(or hybrid) fuzzing.

3. Automotive fuzzing poses challenges due to the inability to perform coverage-guided fuzzing or detailed triage, and testers can only gather information through OBD-II ports or CAN lines.

4. In automotive fuzzing, there are challenges such as Testcase Generation, FAIL Detection, and SUT Reset.

Thank You

- **Lab:** https://mose.kookmin.ac.kr
- **Email:** sh.jeon@kookmin.ac.kr