Polar: Function Code Aware Fuzz Testing of ICS Protocol

Zhengxiong Luo[1], Feilong Zuo[1], **Yu Jiang[1]**, Jian Gao[1], Xun Jiao[2], Jiaguang Sun[1]

[1]School of Software, Tsinghua University

[2]Department of Computer Science and Engineering, Villanova University

Outline

- **Introduction**
  - Background
  - Motivation
- **Polar**
  - **System Design**
  - **Evaluation**
- **Conclusion**

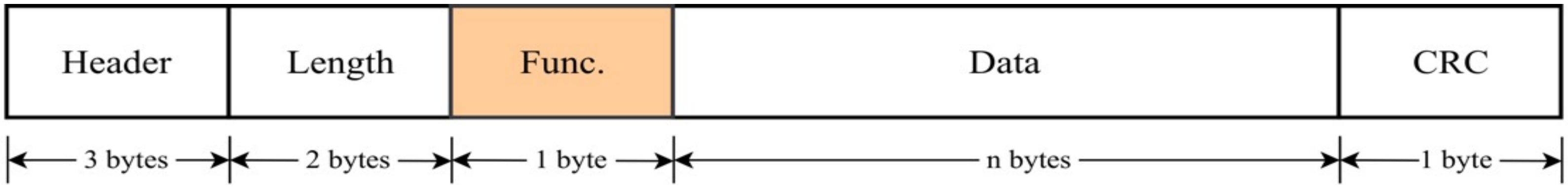
Industry Control System

- Industrial Control System(ICS) is a general term referring to a system of electronic components that control the physical operations of machines[1].
- ICS is widely used to support critical infrastructure, such as power system, transportation, etc.

[1] Jayne Caswell et al. Survey of Industrial Control Systems Security.

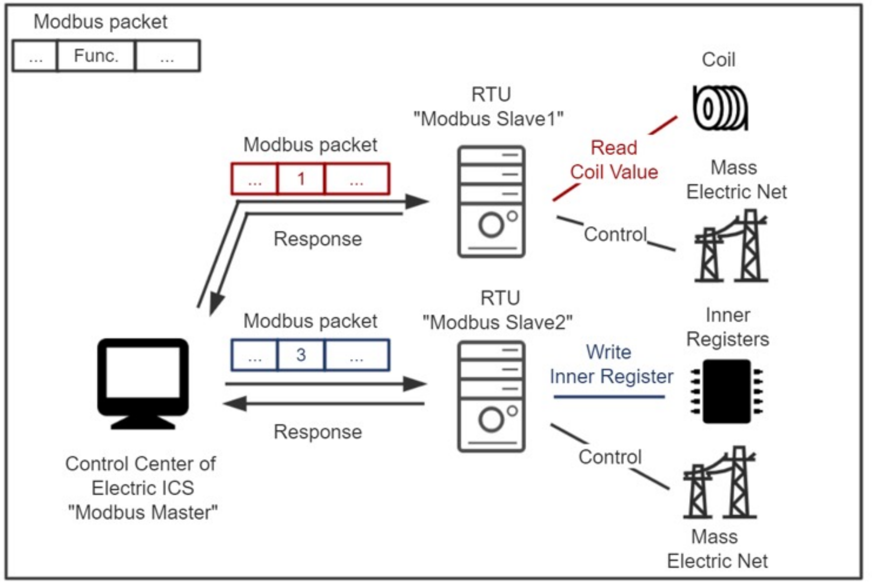
Industry Control System Protocol

- ICS protocol plays a vital role in communications among system components and devices.
- Unlike the common internet protocols, ICS protocols are designed to acquire measurements and status and to control devices. Therefore, ICS protocol packet usually carries a special field, called the function code field, to specify what is received and what should be responded.
- One simple format for example:



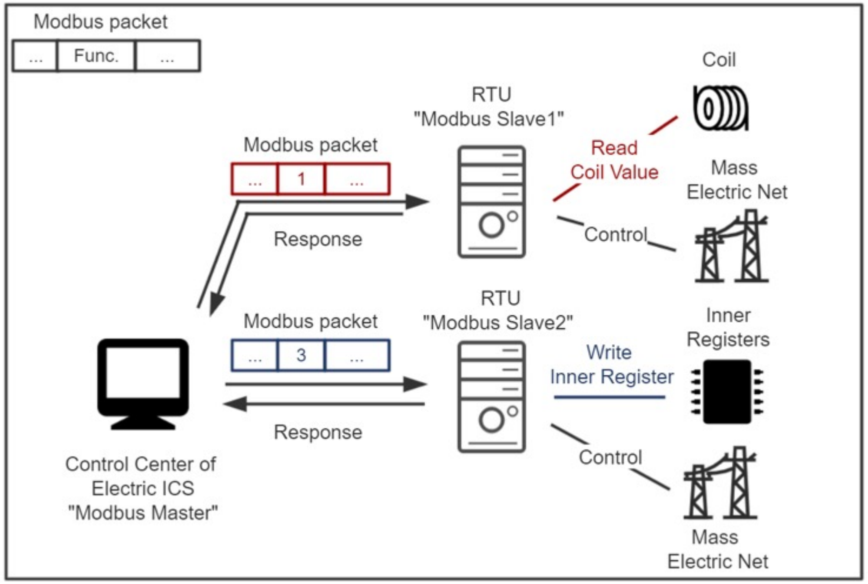
Function Code in ICS Protocol

- Simple example
  - Electrical ICS running Modbus protocol
  - Different values (e.g. 1 and 3) in function code field refer to different orders.



Function Code in ICS Protocol

- Simple example
  - Electrical ICS running Modbus protocol
  - Different values (e.g. 1 and 3) in function code field refer to different orders.



- To meet the demand of the developing industry, ICS protocol is becoming more open.
- This openness has increased the susceptibility to attack, primarily due to greater awareness of ICS protocols.

ICS Protocol Vulnerability

```c
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0], *pl; // p points to the received package
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */
    ......
      hbtype = *p++; // /*Type of the p */
    n2s(p, payload); // The length of the package is payload
    pl = p; // p -> message content
    unsigned char *buffer, *bp; int r;
    buffer = OPENSSL_malloc(1 + 2 + payload + padding); // 3 bytes for type and length
      bp = buffer;
    ......
    *bp++ = TLS1_HB_RESPONSE; // type
    s2n(payload, bp); // length is payload
    memcpy(bp, pl, payload);
```

Heartbleed vulnerability

```c
if(1+2+payload+16>s->s3->rrec.length)
 return 0;
```

memcpy(bp, pl, payload);

Industry Control System Incidents

- Frequent accidents arising from ICS protocol gravely threaten the ICS, resulting in enormous property loss and social infrastructure damage.

- **Protecting ICS Protocol from attacks is essential!**

Attack	Year
Venezuela Blackout	2019
Saudi Arabia TRISIS	2017
Ukraine CRASHOVERRIDE	2016
Ukraine BLACKENERGY3	2015
German Steel Mill Cyber Attack	2014
DragonFly	2014
Havex Malware	2013
Telvent Canada Attack	2012

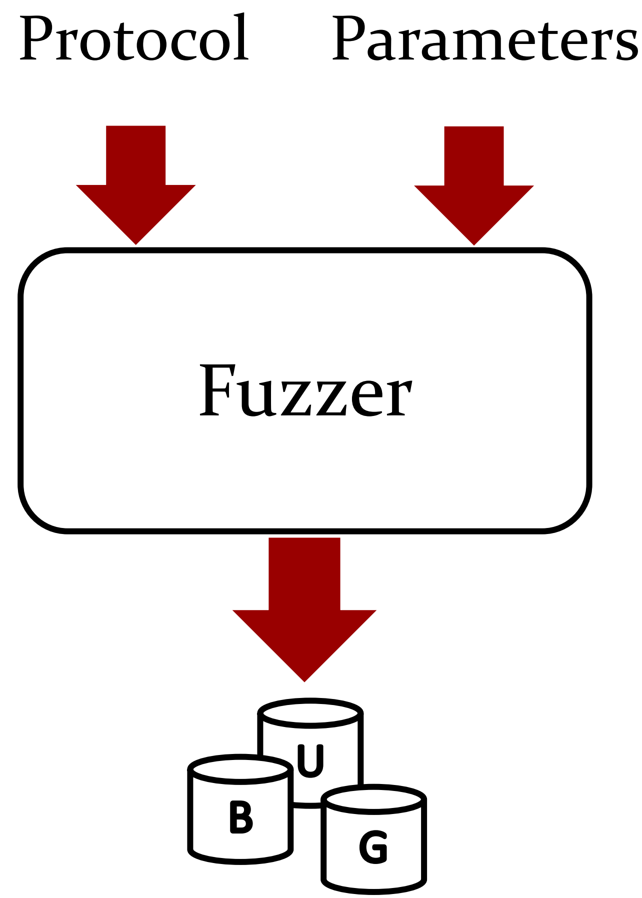
Fuzz Testing for ICS Protocol

DNP3 Modbus PROFI NET

PEACH FUZZER synopsys

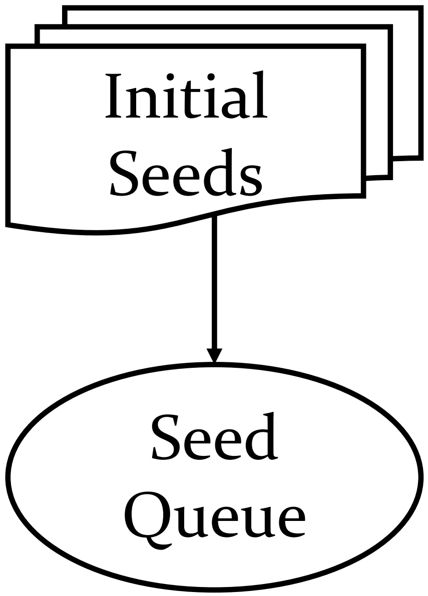
american fuzzy lop (2.52b)

Heartbleed Shellshock

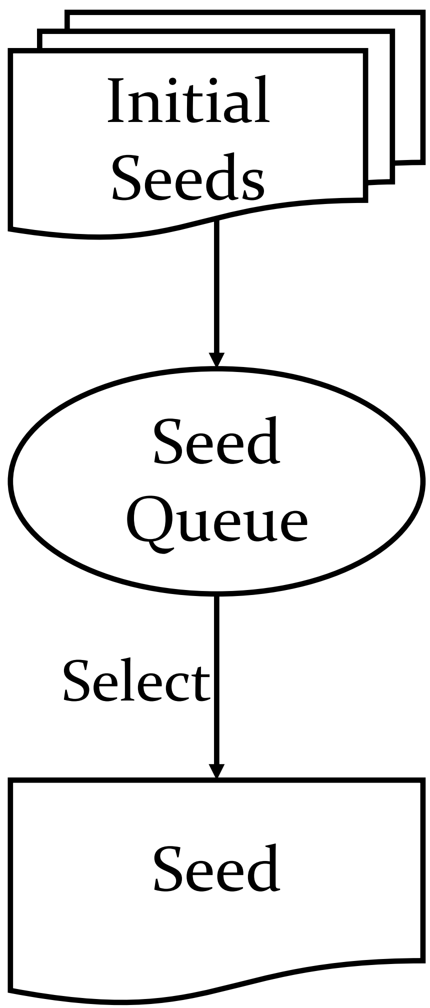


Fuzz Testing is efficient in Bug Detection

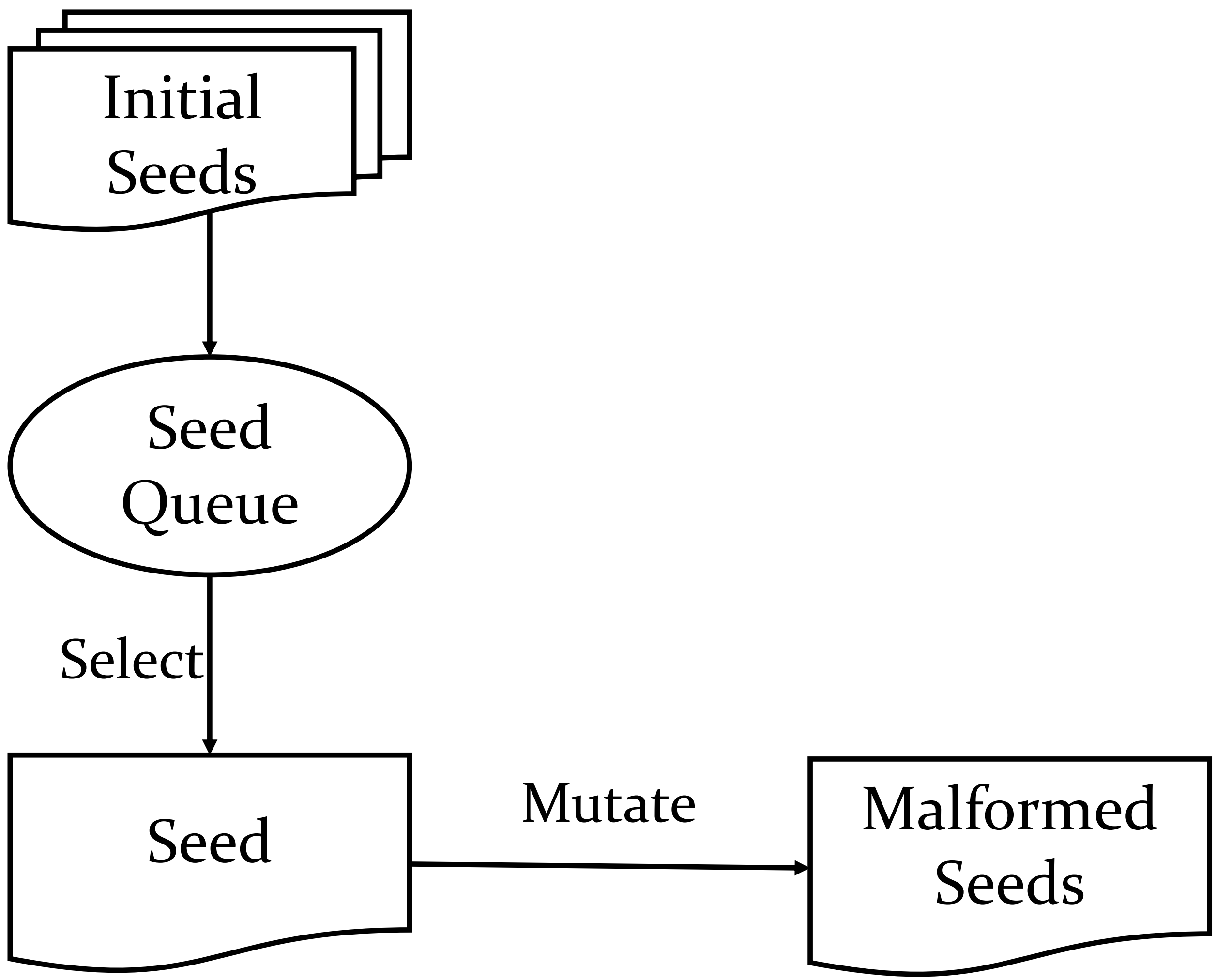
Fuzz Testing Workflow



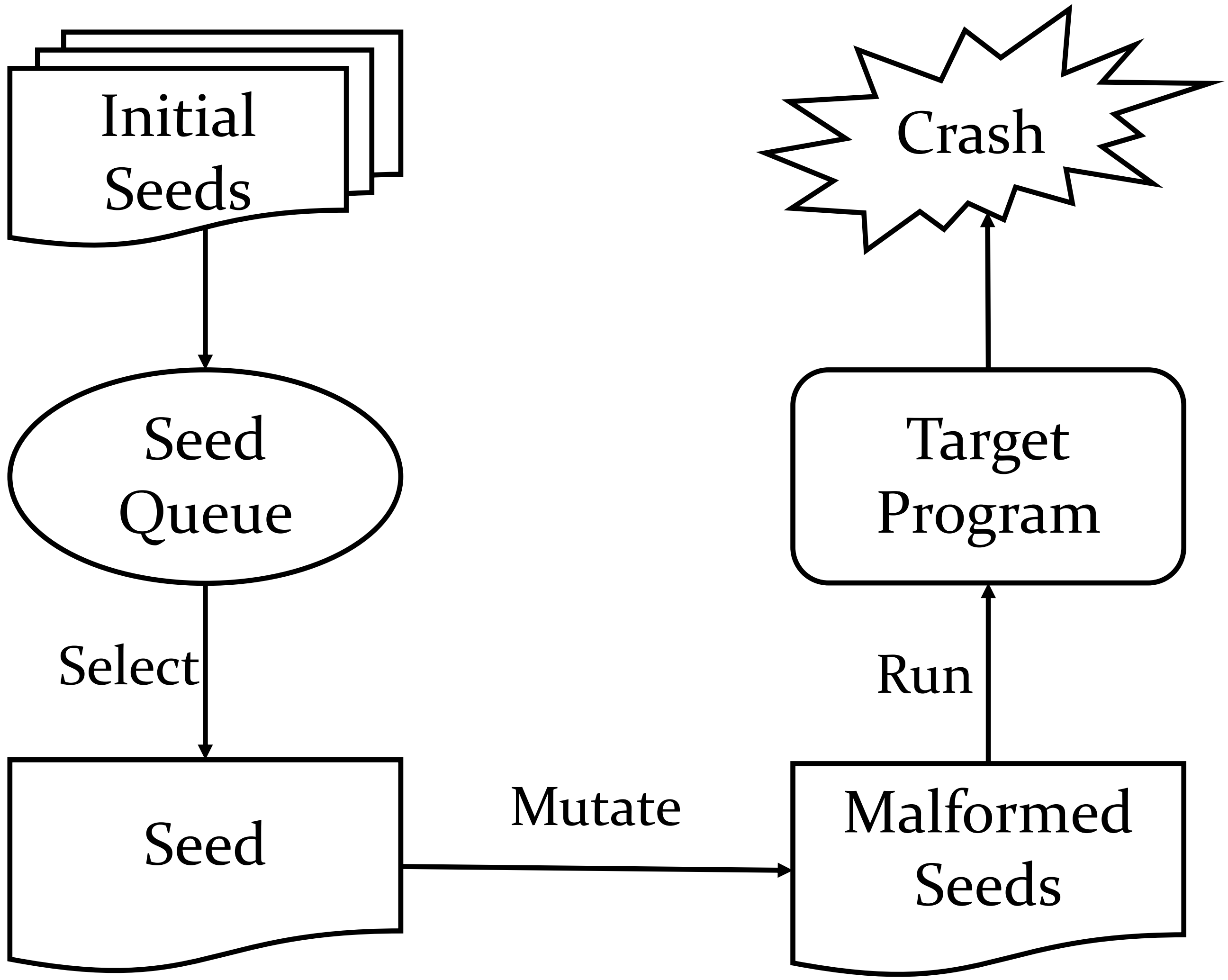
Fuzz Testing Workflow



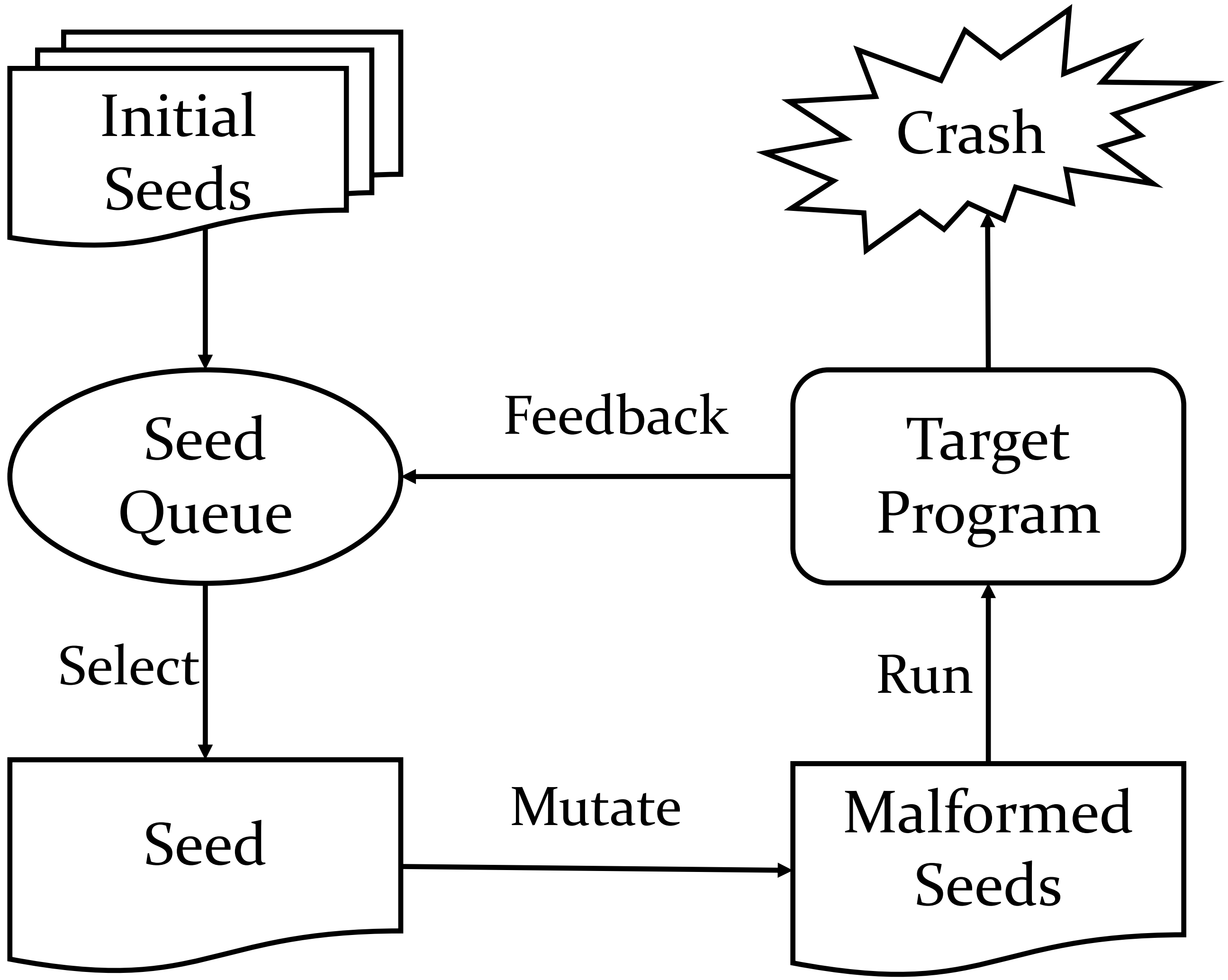
Fuzz Testing Workflow



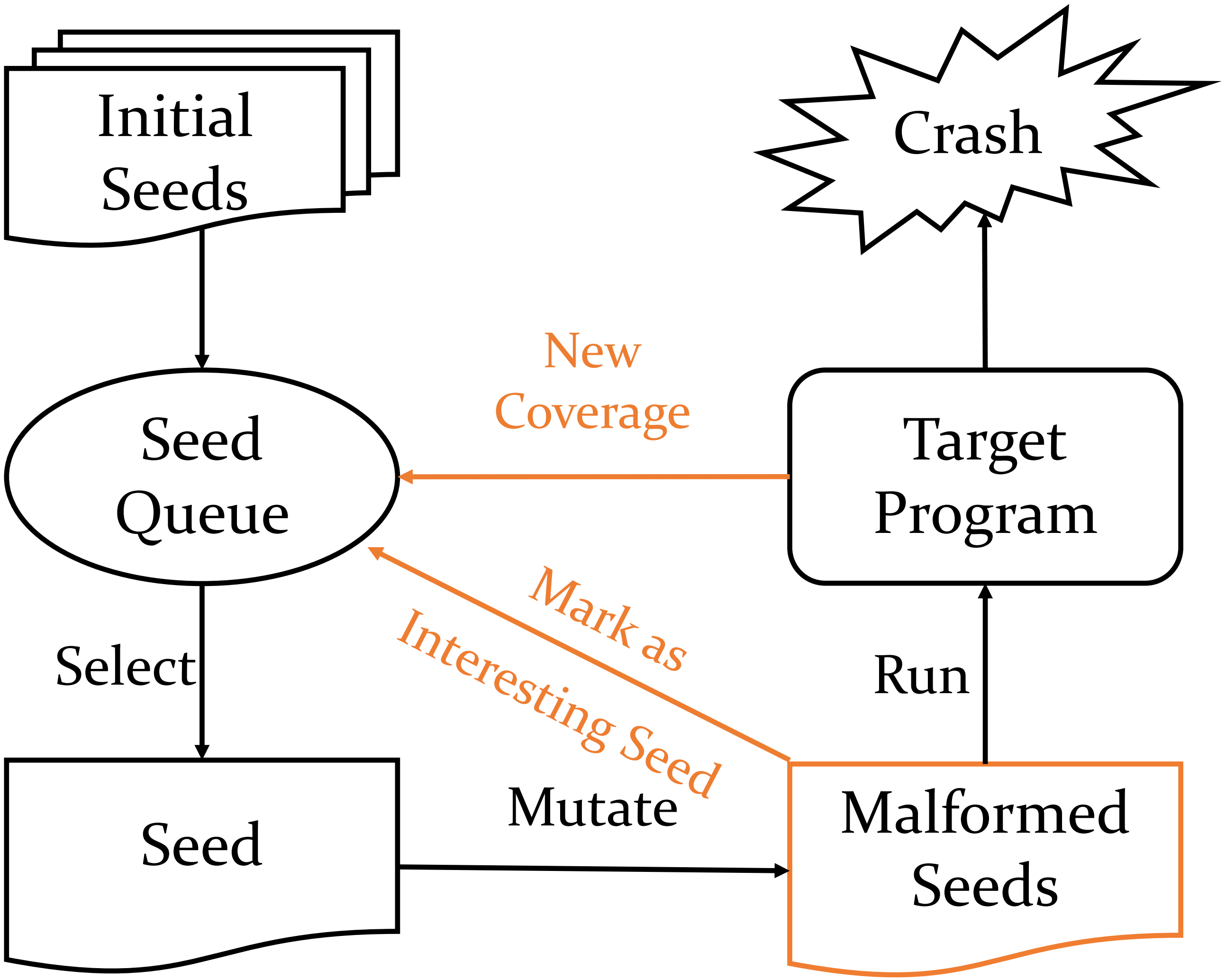
Fuzz Testing Workflow



Fuzz Testing Workflow

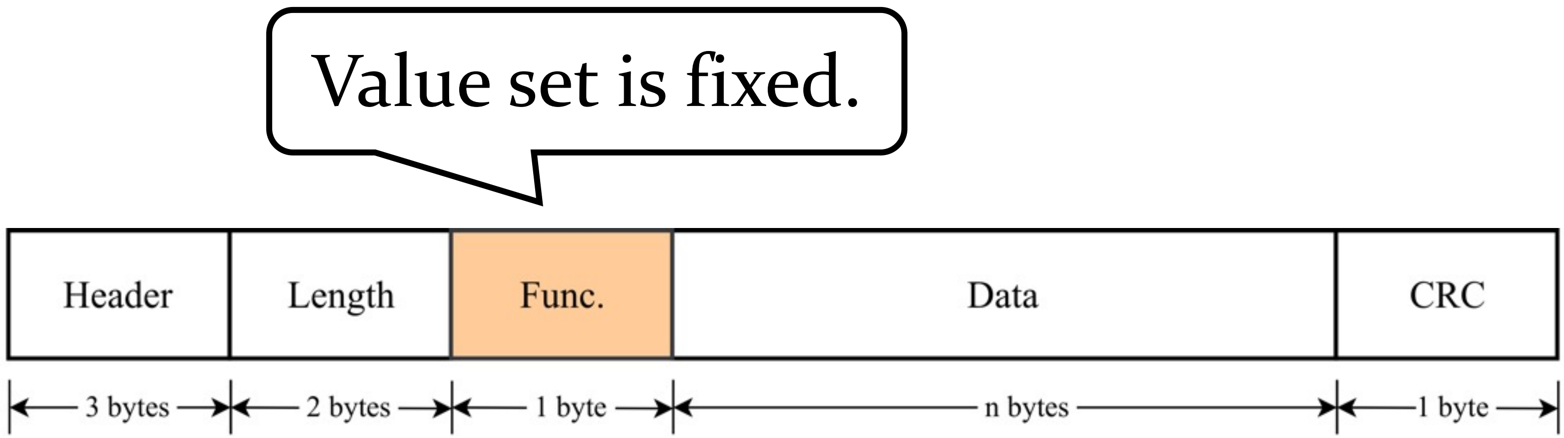


Coverage-Guided Fuzz Testing



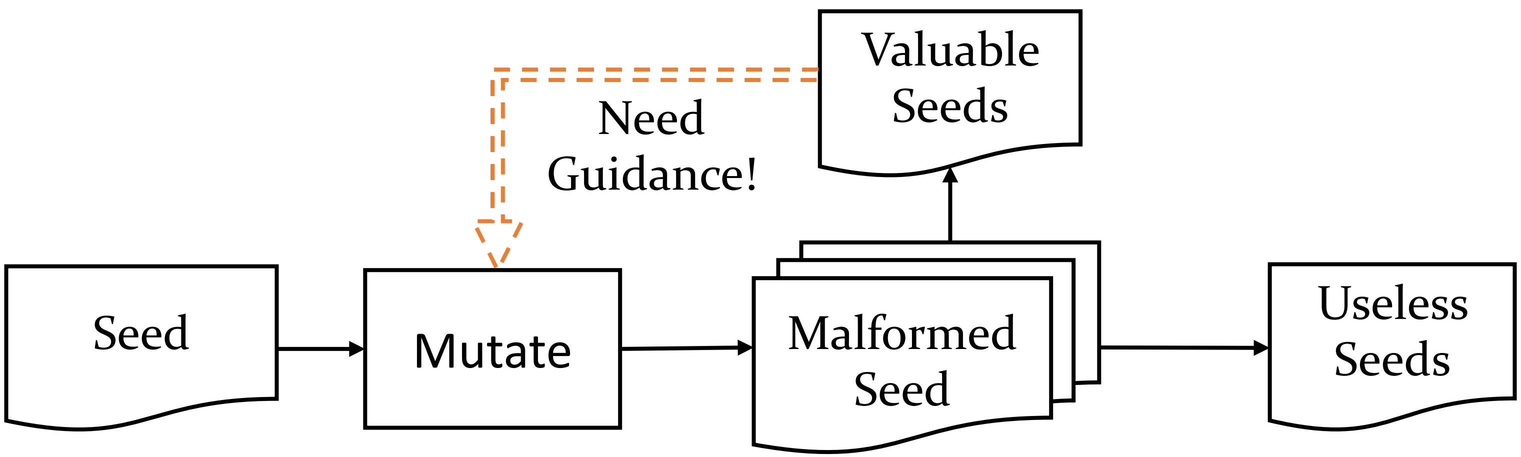
Challenges for Traditional Fuzzers

- Challenge 1: Traditional fuzzers are unaware of protocol information, treating each bit/byte equally is inefficient.



Challenges for Traditional Fuzzers

- Challenge 2: Critical guidede information such as valuable path information embedded in seed inputs is routinely underutilized.



Intuition

- Function code field plays an essential role in ICS protocol, making fuzzers aware of function code information can help them determine where and how to mutate.

- Some security-sensitive points in the protocol (e.g., dynamic memory allocation *malloc*, we define them as vulnerable operations) can be obtained to assist fuzzers in generating more inputs so as to exercise those vulnerable operations more often.

Outline

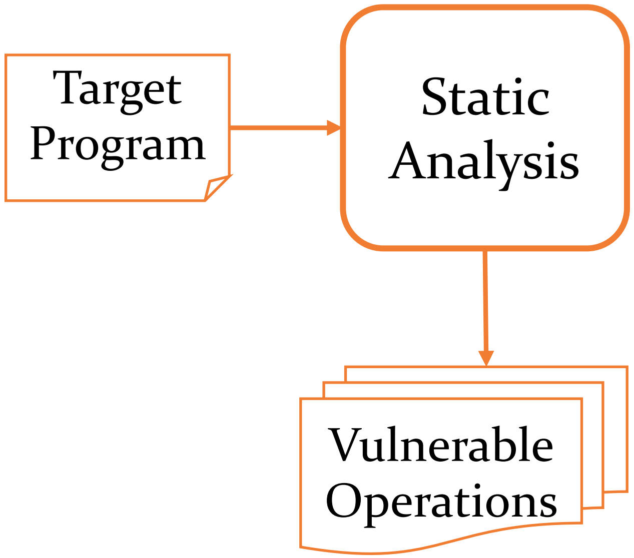
- **Introduction**
  - Background
  - Motivation
- **Polar**
  - **System Design**
  - **Evaluation**
- **Conclusion**

Key Questions

- Q1: How to locate vulnerable operations in target ICS protocol program?
- Q2: How to extract function code information for given ICS protocol program?
- Q3: How to effectively and efficiently fuzz for security vulnerability detection by leveraging information obtained above?

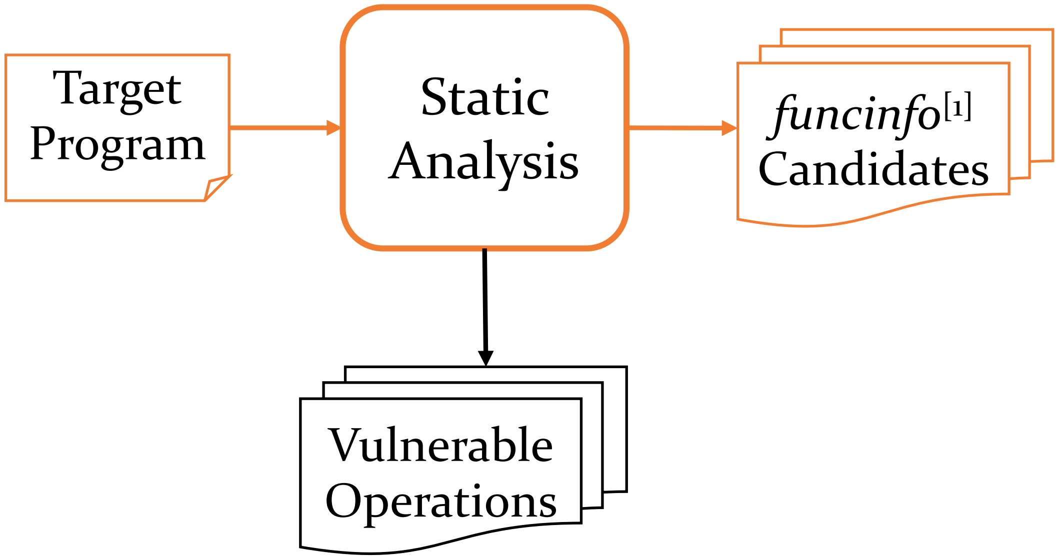
Polar Overview

Q1: locate vulnerable operations



Polar Overview

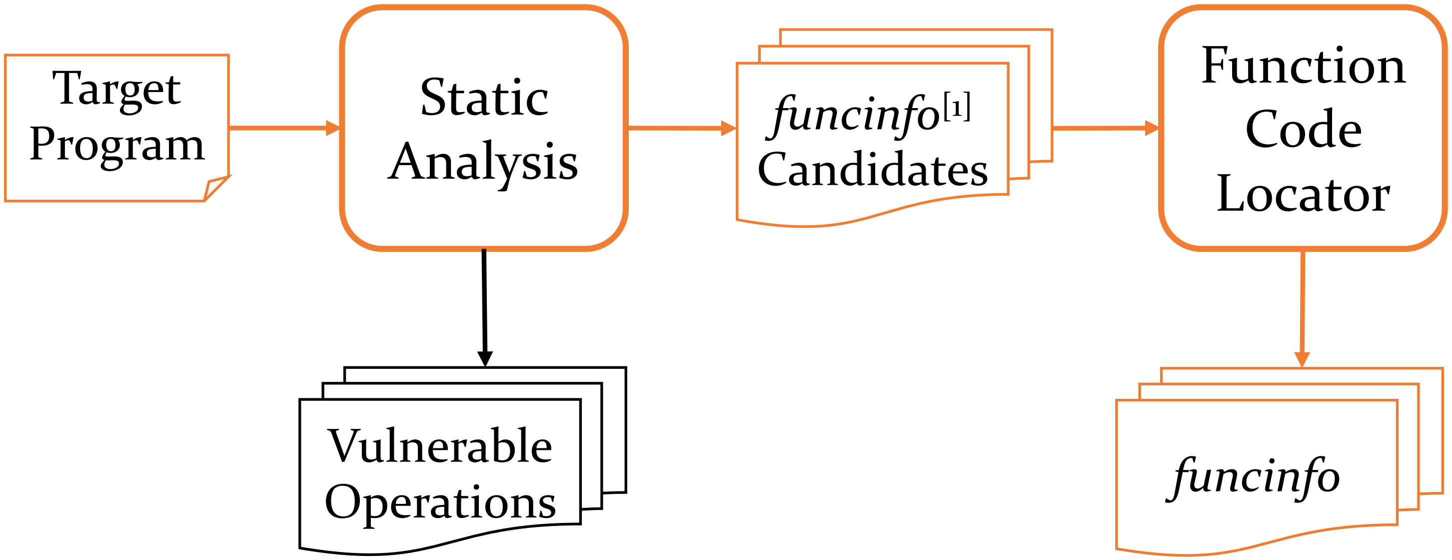
Q2: extract function code information



[1] *funcinfo* is is the abbreviation of function code information

Polar Overview

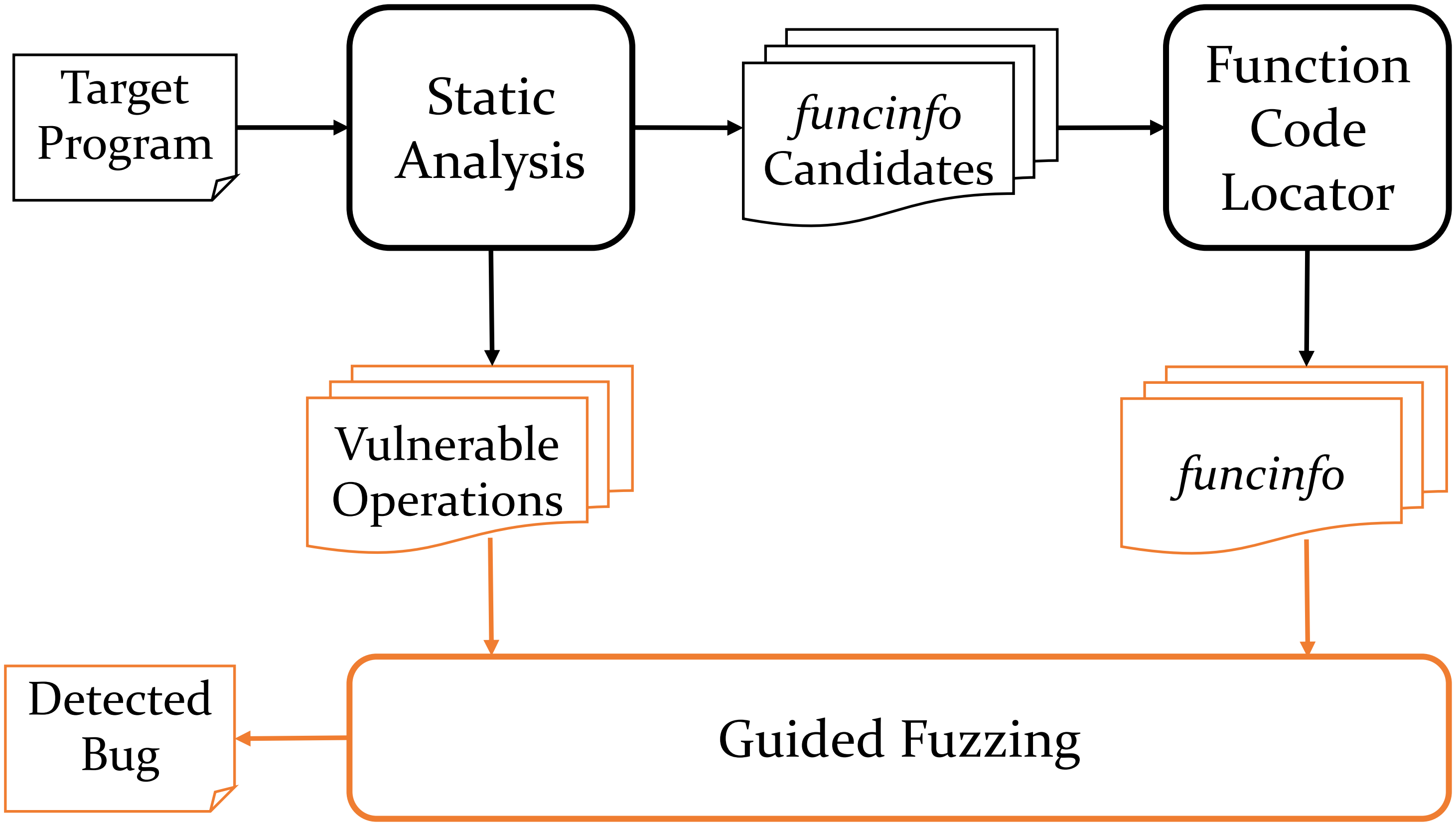
Q2: extract function code information



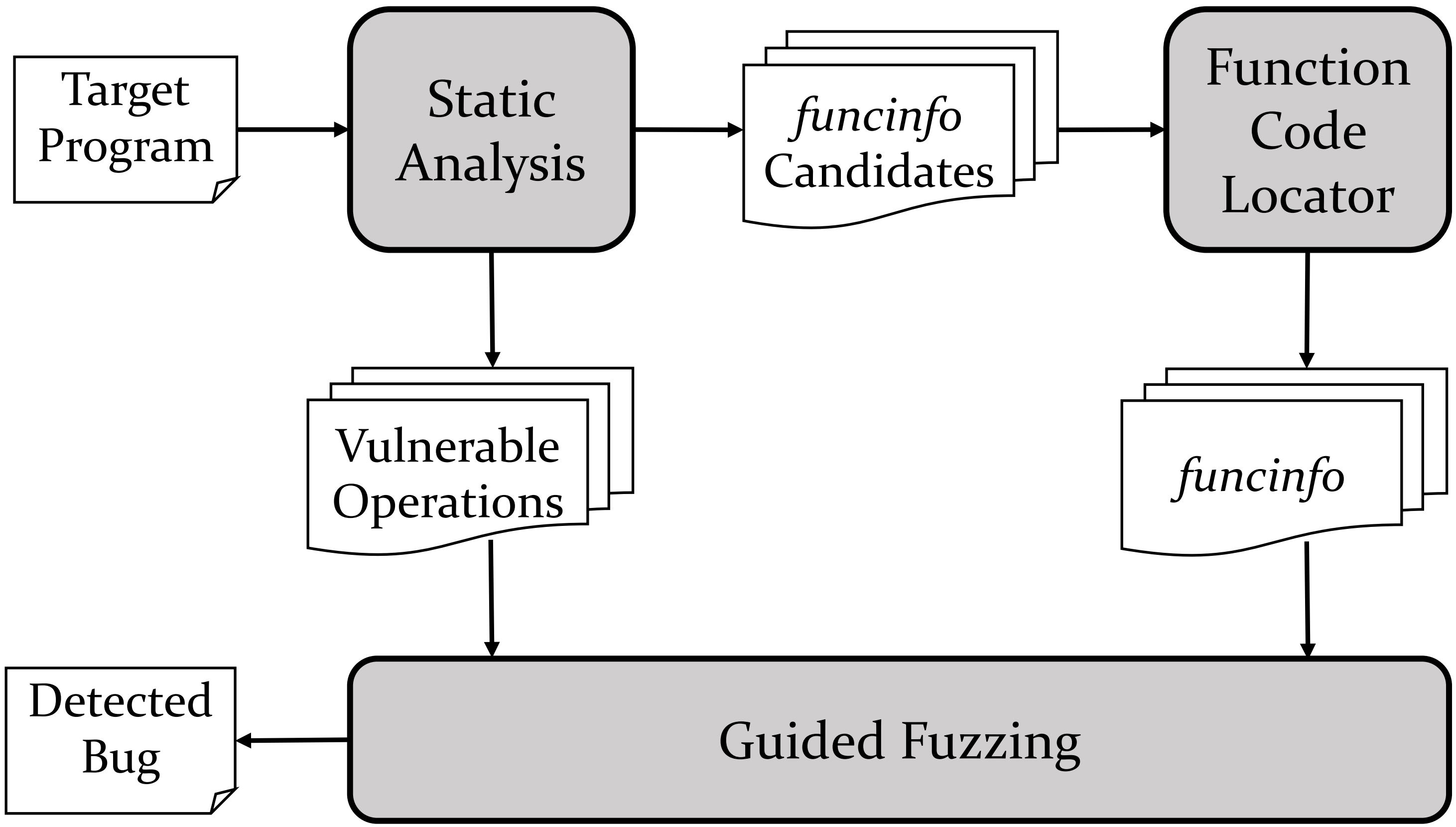
[1] *funcinfo* is is the abbreviation of function code information

Polar Overview

Q3: effectively and efficiently fuzz



Polar Overview



Static Analysis-Vulnerable Operation

- The operations related to memory are usually security-sensitive:
  a. dynamic memory allocation functions (e.g. malloc, realloc)
  b. functions implementing operations on strings (e.g. memcpy, strcpy).

```
void decode(FILE* fd) {
  ...
  int size = get_size(fd);
  int *p = malloc(size);
  ...
}
```

Static Analysis-Vulnerable Operation

- The operations related to memory are usually security-sensitive:
  a. dynamic memory allocation functions (e.g. malloc, realloc)
  b. functions implementing operations on strings (e.g. memcpy, strcpy).

```
1 void decode(FILE* fd) {
2   ...
3   int size = get_size(fd);
4   int *p = malloc(size);
5   ...
6 }
```

Report Entry:

source file	line	function
decoder.c	4	malloc

- Static Analysis Module locates those operations by scanning the source code.

Static Analysis-Function code candidate

Observation:

The function code processing statement is usually a multi-branch statement

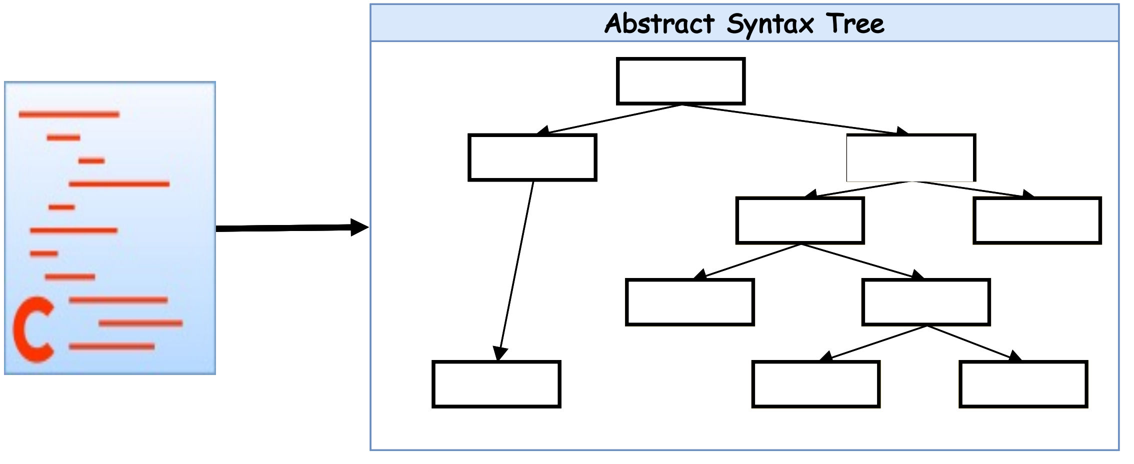
Static Analysis-Function code candidate

Observation:

The function code processing statement is usually a multi-branch statement

Solution:

- Step1: Translate the source code into abstract Syntax tree (AST).



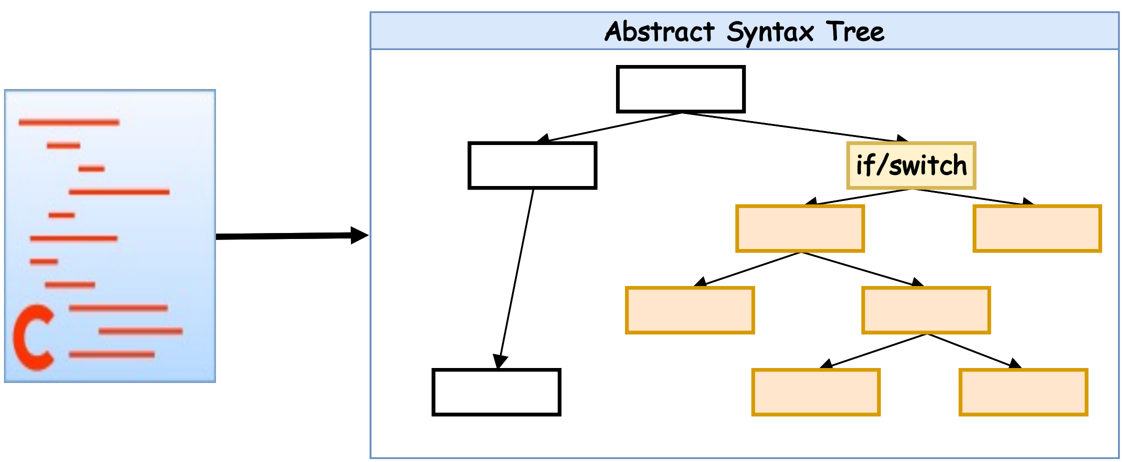
Static Analysis-Function code candidate

Observation:

The function code processing statement is usually a multi-branch statement

Solution:

- Step1: Translate the source code into abstract Syntax tree (AST).
- Step2: Use DFS to traverse AST and locate multi-branch subtree.



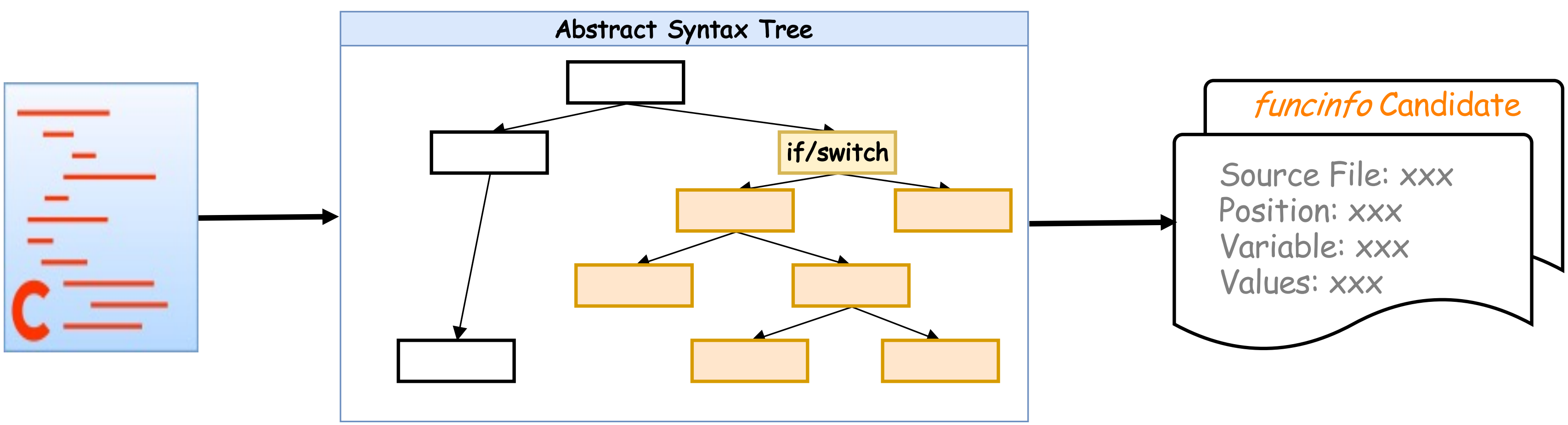
Static Analysis-Function code candidate

Observation:

The function code processing statement is usually a multi-branch statement

Solution:

- Step1: Translate the source code into abstract Syntax tree (AST).
- Step2: Use DFS to traverse AST and locate multi-branch subtree.
- Step3: Extract related information from subtree.



Function Code Locator

Observation:

The byte offsets of function code in the protocol packets are fixed.

Function Code Entry:

Source File: decoder.c
Position: line 12
Start Byte: 6
End Byte: 7
Variable: func_code
Values: [0x01, 0x0F, 0x16, 0x17]

Solution:

- Step1: Run target program with packets sampled on network.
- Step2: Use taint analysis to infer which bytes in input packet determine the value of the variable in *funcinfo* candidate.
- Step3: Check whether the byte offset is always the same, if not, discard the candidate.

Guided Fuzzing

- After the above two modules, we know the positions of vulnerable operations and function code statements.

```
void decode(FILE* fd) {
    ...
    int size = get_size(fd);
    int *p = malloc(size);
    ...
    switch(func_code) {
      case 1:
      ...
      case n:
      ...
    }
}
```

Vulnerable Operation

Function Code Statement

Guided Fuzzing

- After the above two modules, we know the positions of vulnerable operations and function code statements.
- Lightweight instrumentation is applied to trace them for each program execution.

```
void decode(FILE* fd) {
    ...
    int size = get_size(fd);
    LOG(...)
    int *p = malloc(size);
    ...
    LOG(...)
    switch(func_code) {
      case 1:
      ...
      case n:
      ...
    }
}
```

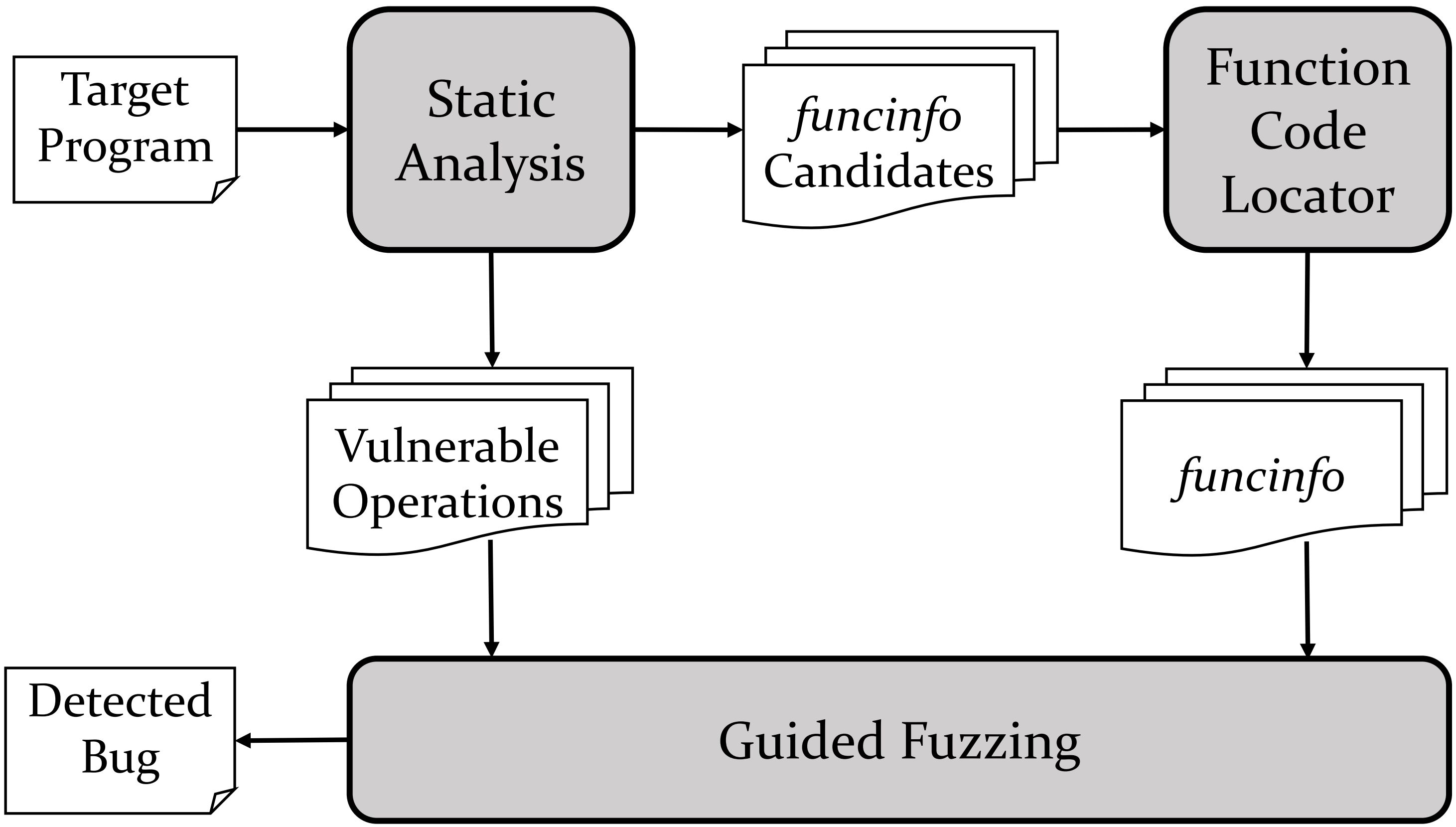
Instrumentation

Vulnerable Operation

Instrumentation

Function Code Statement

Guided Fuzzing



Three Optimized Fuzzing Strategies

Guided Fuzzing

- We add three fuzzing strategies, one strategy for vulnerable operations and two strategies for function code aware.

Power Schedule

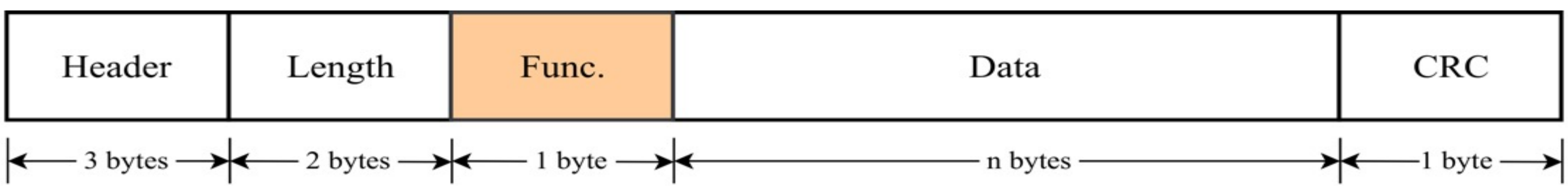
- For seed I, Count_I donates hit times of vulnerable operations during execution.
- The more Count_I is, the more energy would be assigned to I for further mutation.

$$\mathcal{E}(I) = min\left(\frac{\mathcal{E}_{ini}(I)}{\beta} \cdot h(\text{Count}_I), M\right)$$

Function Code Field Protection

Observation:

The legal values of function code are taken from a fixed small set, where enumerating exhaustively would be unnecessary.



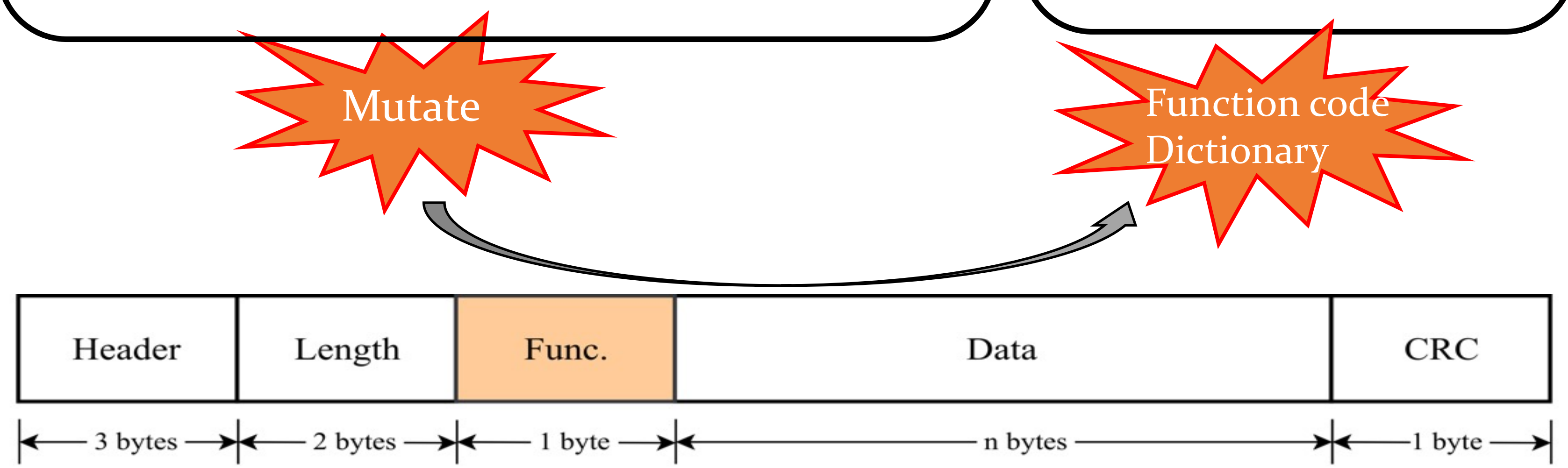
Function Code Field Protection

Observation:

The legal values of function code are taken from a fixed small set, where enumerating exhaustively would be unnecessary.

Solution:

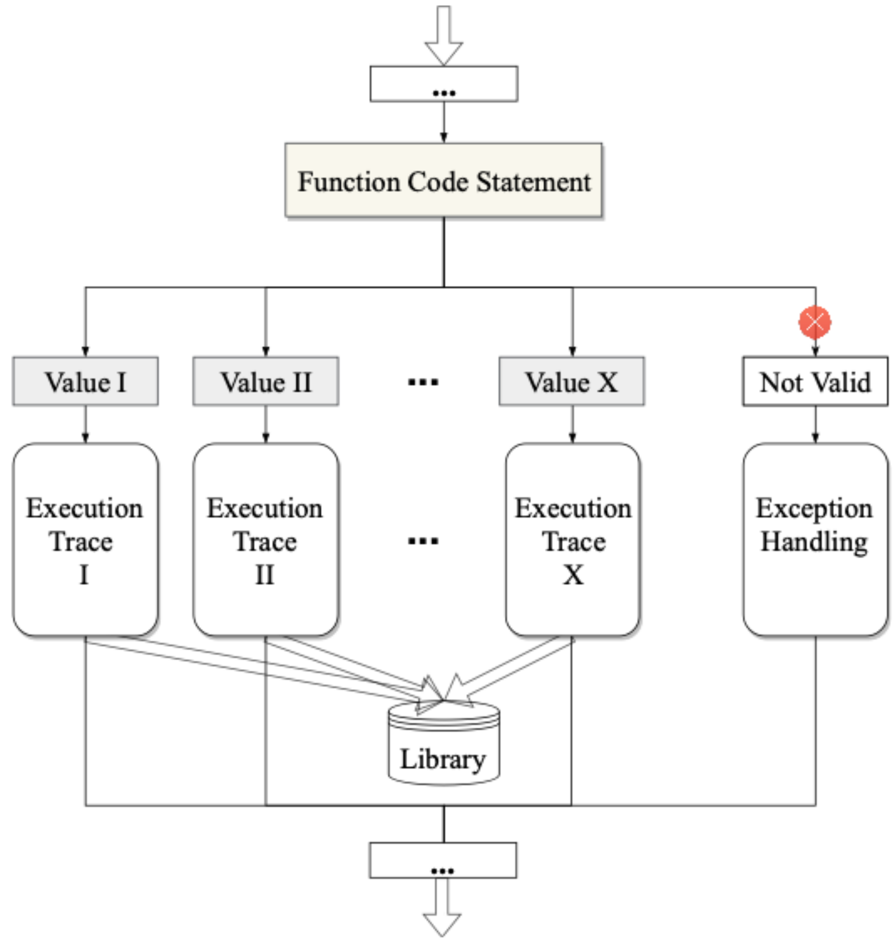
Protect it against random mutation.



Synchronization Mechanism

Observation:

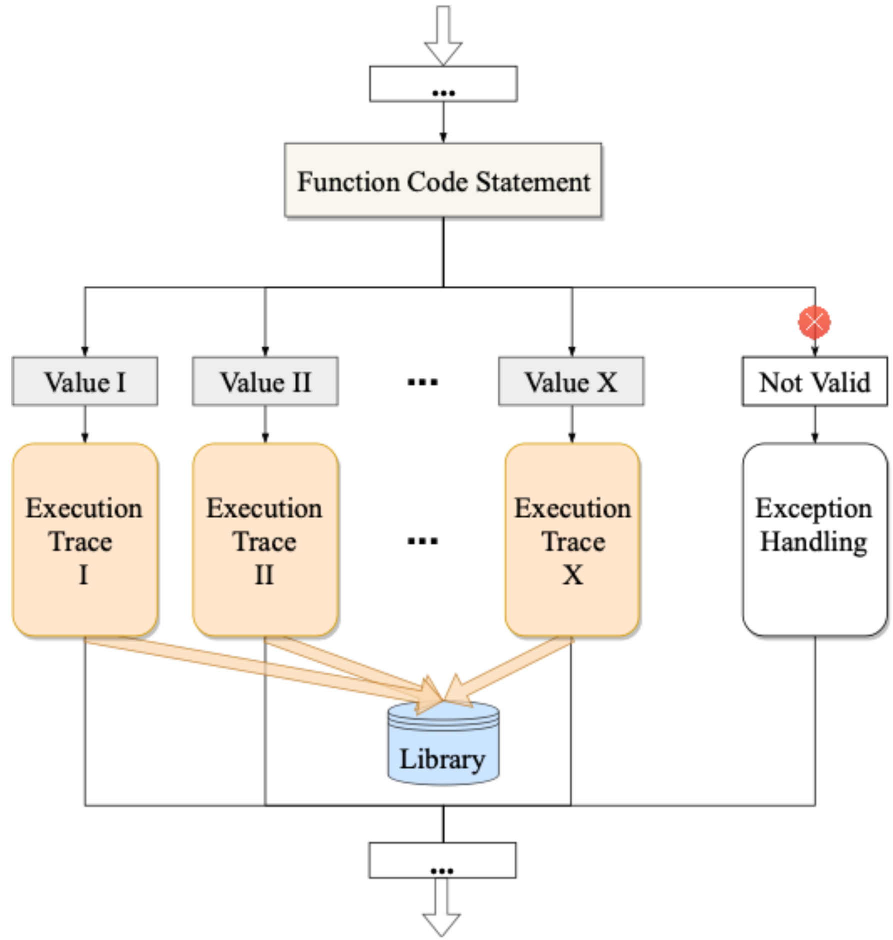
- Different values of function code cause different execution traces.



Synchronization Mechanism

Observation:

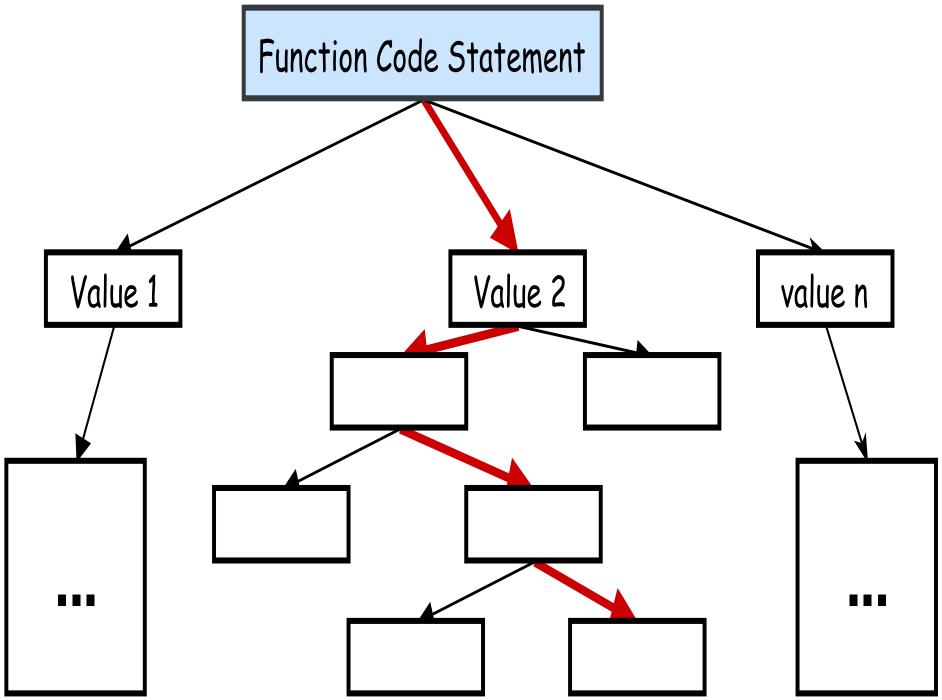
- Different values of function code cause different execution traces.
- But there are also some similarities between different traces: they tend to include some same code snippet or call the same functions in the library.



Synchronization Mechanism

Solution:

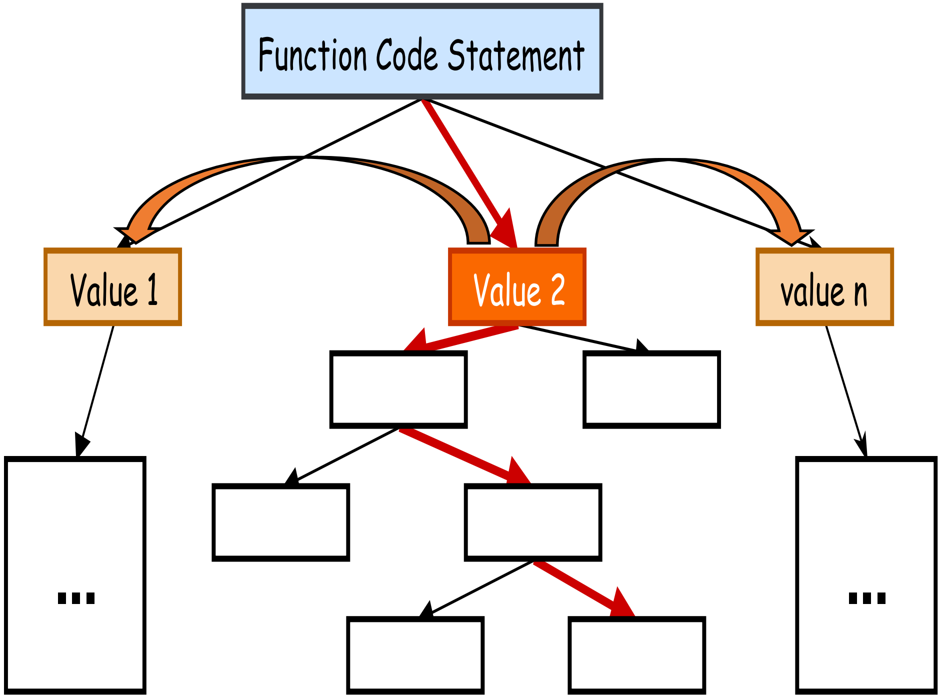
- During fuzzing, some seeds may achieve new coverage.



Synchronization Mechanism

Solution:

- During fuzzing, some seeds may achieve new coverage.
- New path information can be synchronized to help explore new paths for other seeds with different values of the function code.



Evaluation

- Component evaluation
  - E1: Whether Polar can locate function code statements?
  - E2: Are proposed fuzzing strategies valuable?
- Overall evaluation
  - E3: Whether Polar can detect previously unknown vulnerabilities in real-world ICS protocol programs?

Experiment Setup

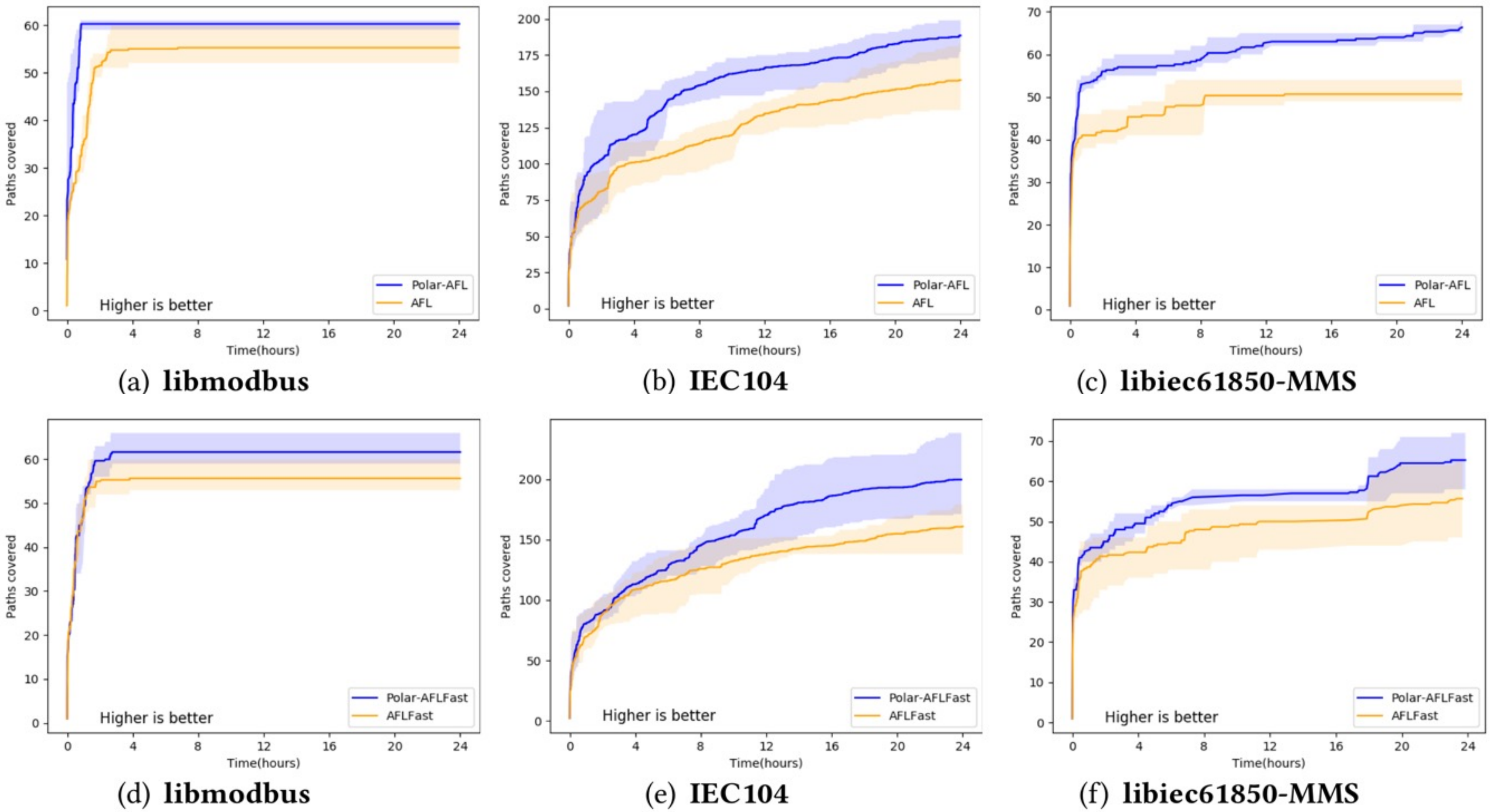
- We selected three widely used open-source implementations of ICS protocols.
- Including Modbus, IEC104 and IEC 61850.
- Those ICS protocols are international standard widely used in critical infrastructures.

E1: Locate Function Code

Polar precisely located function code of those protocols.

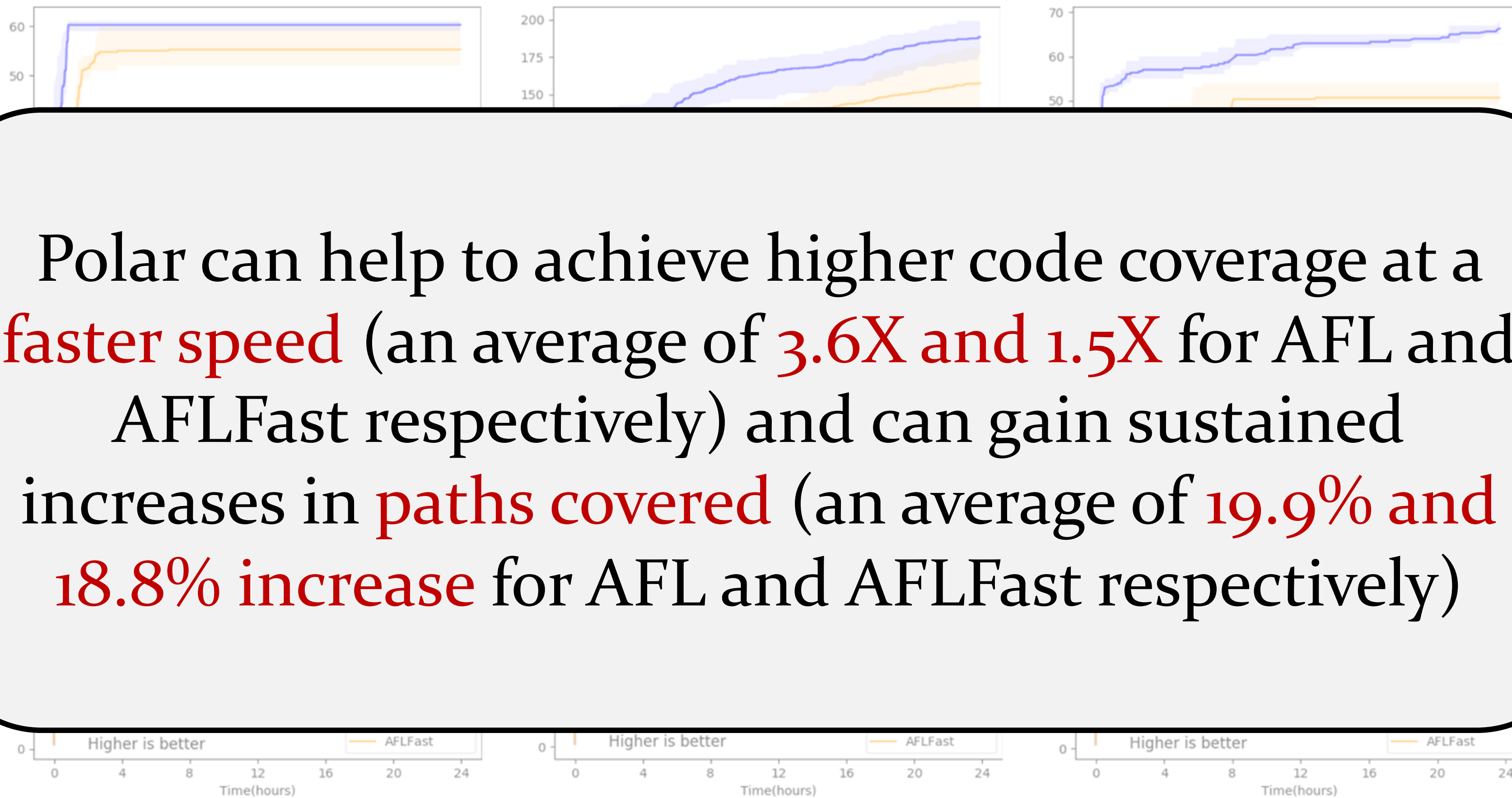
Project	$\lvert\mathcal{M}\rvert$	$\lvert funcinfo \rvert$	Set of Legal Values (hexadecimal) for Each *funcinfo* Piece	True?
libmodbus	11	1	[01,02,03,04,05,06,07,0F,10,11,16,17]	✓
IEC104	12	2	[07,13,43,0B,23,83,64]	✓
			[83,64,67,30,32,80,81]	✓
libiec61850	174	1	[02,80,A1,82,A4,A5,A6,AB,AC,AD]	✓

E2: Optimize Fuzzing



Number of paths covered by different fuzzing techniques averaged over 25 runs with different seeds

E2: Optimize Fuzzing



Polar can help to achieve higher code coverage at a faster speed (an average of 3.6X and 1.5X for AFL and AFLFast respectively) and can gain sustained increases in paths covered (an average of 19.9% and 18.8% increase for AFL and AFLFast respectively)

(d) **libmodbus** (e) **IEC104** (f) **libiec61850-MMS**

Number of paths covered by different fuzzing techniques averaged over 25 runs with different seeds

E3: Previously Unknown Vulnerabilities

- Polar has exposed **10** previously unknown vulnerabilities, **6** of which have been assigned unique CVE identifiers in the U.S National Vulnerability Database.

Project	Type	Advisory	Total
libiec61850	heap buffer overflow NULL pointer dereference SEGV	CVE-2018-18834 , CVE-2018-19185 CVE-2018-18937, CVE-2018-19122 CVE-2018-19093, CVE-2018-19121	6
IEC104	stack buffer overflow SEGV denial of service	Bug-2019-0312 Bug-2019-0207, Bug-2019-0307 Bug-2019-0402	4

E3: Previously Unknown Vulnerabilities

mzillgith commented on 1 Nov 2018 · Contributor

Hi. Thank you for the hint. There has been a bug in the calculation of the GOOSE message size that estimated the size two byte too small. So depending on the data types of the GOOSE payload this problem is triggered. Should be fixed now.

mzillgith commented on 13 Nov 2018 · Contributor

Thanks for the hint. There was another problem in GOOSE payload length calculation. Should be fixed with this commit 8d728b3

CVE-ID

CVE-2018-19185 Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

An issue has been found in libIEC61850 v1.3. It is a heap-based buffer overflow in BerEncoder_encodeOctetString in mms/asn1/ber_encoder.c. This is exploitable even after CVE-2018-18834 has been patched, with a different dataSetValue sequence than the CVE-2018-18834 attack vector.

<table>
<tr><th>#</th><th>CVE ID</th><th>CWE ID</th><th># of Exploits</th><th>Vulnerability Type(s)</th><th>Publish Date</th><th>Update Date</th><th>Score</th><th>Gained Access Level</th><th>Access</th><th>Complexity</th><th>Authentication</th><th>Conf.</th><th>Integ.</th><th>Avail.</th></tr>
<tr><td>1</td><td>CVE-2018-19185</td><td>119</td><td></td><td>Overflow</td><td>2018-11-12</td><td>2018-12-14</td><td>7.5</td><td>None</td><td>Remote</td><td>Low</td><td>Not required</td><td>Partial</td><td>Partial</td><td>Partial</td></tr>
<tr><td colspan="15">An issue has been found in libIEC61850 v1.3. It is a heap-based buffer overflow in BerEncoder_encodeOctetString in mms/asn1/ber_encoder.c. This is exploitable even after CVE-2018-18834 has been patched, with a different dataSetValue sequence than the CVE-2018-18834 attack vector.</td></tr>
<tr><td>2</td><td>CVE-2018-19122</td><td>476</td><td></td><td></td><td>2018-11-09</td><td>2018-12-07</td><td>4.3</td><td>None</td><td>Remote</td><td>Medium</td><td>Not required</td><td>None</td><td>None</td><td>Partial</td></tr>
<tr><td colspan="15">An issue has been found in libIEC61850 v1.3. It is a NULL pointer dereference in Ethernet_sendPacket in ethernet_bsd.c.</td></tr>
<tr><td>3</td><td>CVE-2018-19121</td><td>476</td><td></td><td></td><td>2018-11-09</td><td>2018-12-07</td><td>4.3</td><td>None</td><td>Remote</td><td>Medium</td><td>Not required</td><td>None</td><td>None</td><td>Partial</td></tr>
<tr><td colspan="15">An issue has been found in libIEC61850 v1.3. It is a SEGV in Ethernet_receivePacket in ethernet_bsd.c.</td></tr>
<tr><td>4</td><td>CVE-2018-19093</td><td>284</td><td></td><td></td><td>2018-11-07</td><td>2018-12-13</td><td>5.0</td><td>None</td><td>Remote</td><td>Low</td><td>Not required</td><td>None</td><td>None</td><td>Partial</td></tr>
<tr><td colspan="15">** DISPUTED ** An issue has been found in libIEC61850 v1.3. It is a SEGV in ControlObjectClient_setCommandTerminationHandler in client/client_control.c. NOTE: the software maintainer disputes this because it requires incorrect usage of the client_example_control program.</td></tr>
<tr><td>5</td><td>CVE-2018-18957</td><td>119</td><td></td><td>Overflow</td><td>2018-11-05</td><td>2018-12-07</td><td>7.5</td><td>None</td><td>Remote</td><td>Low</td><td>Not required</td><td>Partial</td><td>Partial</td><td>Partial</td></tr>
<tr><td colspan="15">An issue has been found in libIEC61850 v1.3. It is a stack-based buffer overflow in prepareGooseBuffer in goose/goose_publisher.c.</td></tr>
<tr><td>6</td><td>CVE-2018-18937</td><td>476</td><td></td><td></td><td>2018-11-05</td><td>2018-12-07</td><td>5.0</td><td>None</td><td>Remote</td><td>Low</td><td>Not required</td><td>None</td><td>None</td><td>Partial</td></tr>
<tr><td colspan="15">An issue has been found in libIEC61850 v1.3. It is a NULL pointer dereference in ClientDataSet_getValues in client/ied_connection.c.</td></tr>
</table>

E3: Previously Unknown Vulnerabilities

- Taking the bug in IEC104 for example.

```
if(CsumTemp == csum){
    LOG("-%s-,data need ack:%d,Len:%d,seek:%d \n",__FUNCTION__,FlagNum,DataLen,Iec10x_Update_SeekAddr);

    for(i=0; i<3; i++){
        ret = IEC10X->SaveFirmware(DataLen,DataPtr,FirmwareType,Iec10x_Update_SeekAddr);
        if(ret == RET_SUCESS)
            break;
    }

    if(ret == RET_ERROR){
        LOG("save firmware error \n");
        break;
    }
    Iec104_BuildDataAck(TI, IEC10X_COT_DATA_ACK, FirmwareType, FlagNum,1);

    FirmFlagCount = FlagNum;
    Iec10x_Update_SeekAddr+=DataLen;
}else{
    LOG("%s,need ack check sum error:%d,need:%d,num...:%d\n",__FUNCTION__,CsumTemp,csum,FlagNum);
    //Iec104_BuildDataAck(TI, IEC10X_COT_ACT_TERMINAL, FirmwareType, FlagNum,1);
}
```

It is caused by tending to call an unimplemented function (SaveFirmware), which then leads to application crash.

E3: Previously Unknown Vulnerabilities

- Taking the bug in IEC104 for example.
- If this bug is made use of for destructive purposes, the server device can immediately shut down, causing the whole system to crash.

```
if(CsumTemp == csum){
    LOG("—%s—,data need ack:%d,Len:%d,seek:%d \n",__FUNCTION__,FlagNum,DataLen,Iec10x_Update_SeekAddr);

    for(i=0; i<3; i++){
        ret = IEC10X->SaveFirmware(DataLen,DataPtr,FirmwareType,Iec10x_Update_SeekAddr);
        if(ret == RET_SUCESS)
            break;
    }

    if(ret == RET_ERROR){
        LOG("save firmware error \n");
        break;
    }
    Iec104_BuildDataAck(TI, IEC10X_COT_DATA_ACK, FirmwareType, FlagNum,1);

    FirmFlagCount = FlagNum;
    Iec10x_Update_SeekAddr+=DataLen;
}else{
    LOG("%s,need ack check sum error:%d,need:%d,num...:%d\n",__FUNCTION__,CsumTemp,csum,FlagNum);
    //Iec104_BuildDataAck(TI, IEC10X_COT_ACT_TERMINAL, FirmwareType, FlagNum,1);
}
```

It is caused by tending to call an unimplemented function (SaveFirmware), which then leads to application crash.