CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems

Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, Taesoo Kim

USENIX Annual Technical Conference
July 14, 2017

The Affiliated Institute of ETRI    Georgia Institute of Technology    Purdue University
Why Microsoft can’t detect a driver with a bug (NDProxy)?

```cpp
bool flag_table[125] = {false};
void (*fn_table[36])();

int dispatch_device_io_control(ulong ctrl_code, ulong *buf) {
    switch (ctrl_code) {
    case 0x8fff23c4:
        ...
    case 0x8fff23cc:
            return -1;
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();

        for (int i=1; i<=buf[0]; ++i) {...}
    }
}

```
Why Microsoft can’t detect a driver with a bug (NDProxy)?

```c
bool flag_table[125] = {false};
void (*fn_table[36])();

int dispatch_device_io_control(ulong ctrl_code, ulong *buf) {
    switch (ctrl_code) {
    case 0x8fff23c4:
        ...
    case 0x8fff23cc:
            return -1;
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();
        for (int i=1; i<=buf[0]; ++i) {...}
    }

```
Why Microsoft can’t detect a driver with a bug (NDProxy)?

bool flag_table[125] = {false};  
void (*fn_table[36])(());

Microsoft’s large-scale fuzzing tools couldn’t this bug

```c
    case 0x8fff23cc:
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])(());
    for (int i=1; i<=buf[0]; ++i) {...}
```


Challenge 1: Path explosion because of array and loop

```c
bool flag_table[125] = {false};
void (*fn_table[36])();

int dispatch_device_io_control(ulong ctrl_code, ulong *buf) {
    switch (ctrl_code) {
    case 0x8fff23c4:
        ...
        break;
    case 0x8fff23cc:
            return -1;
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();

        for (int i=1; i<=buf[0]; ++i) {...}
    }
```
Challenge 1: Path explosion because of array and loop

```c
bool flag_table[125] = {false};
void (*fn_table[36])();

int dispatch_device_io_control(ulong ctrl_code, ulong *buf) {
    switch (ctrl_code) {
    case 0x8fff23c4:
        ...
    case 0x8fff23cc:
            return -1;
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();

        for (int i=1; i<=buf[0]; ++i) {...}
    }
```
Challenge 1: Path explosion because of array and loop

```c
bool flag_table[125] = {false};
void (*fn_table[36])();

int dispatch_device_io_control(ulong ctrl_code, ulong *buf) {
    switch (ctrl_code) {
    case 0x8fff23c4:
        ...
        break;
    case 0x8fff23cc:
            return -1;
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();
        for (int i=1; i<=buf[0]; ++i) {...}
    }
```
Challenge 1: Path explosion because of array and loop

```c
bool flag_table[125] = {false};
void (*fn_table[36])();

int dispatch_device_io_control(ulong ctrl_code, ulong *buf)
{
    switch (ctrl_code) {
    case 0x8fff23c4:
        ...
    case 0x8fff23cc:
            return -1;
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();
        for (int i=1; i<=buf[0]; ++i) {...}
    }
}
```

**Symbolic variables**

**Symbolic memories**

**Loop controlled by a symbolic variable**
Challenge 1: Path explosion because of array and loop

```c
bool flag_table[125] = {false};
void (*fn_table[36])();

int dispatch_device_io_control(ulong ctrl_code, ulong* buf) {
    switch (ctrl_code) {
    case 0x8fff23c4:
        ...
    case 0x8fff23cc:
            return -1;
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();
        for (int i=1; i<=buf[0]; ++i) {...}
    }
}
```

More than million paths (124 x 36 x 246) to explore because of two arrays and a single loop

- **Symbolic memories**
  - Loop controlled by a symbolic variable
Challenge 1: Path explosion because of array and loop

• The number of feasible program paths to test exponentially increases according to its size
• COTS OS is complex and huge
• Almost infinite number of paths to test
Challenge 2: Difficulty in constructing pre-contexts to test targets

```c
bool flag_table[125] = {false}; // default: false
void (*fn_table[36])();

int dispatch_device_io_control(ulong ctrl_code, ulong *buf)
{
    switch (ctrl_code) {
    case 0x8fff23c4:
        for (int i=0; i<125; ++i)
            flag_table[i] = true;
    case 0x8fff23cc:
        ...
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();
    }
```
Challenge 2: Difficulty in constructing pre-contexts to test targets

bool flag_table[125] = {false}; // default: false
void (*fn_table[36])(());

int dispatch_device_io_control(ulong ctrl_code, ulong *buf)
{
    switch (ctrl_code) {
    case 0x8fff23c4:
        for (int i=0; i<125; ++i)
            flag_table[i] = true;
    case 0x8fff23cc:
        ...
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();
    }
Challenge 2: Difficulty in constructing pre-contexts to test targets

```c
bool flag_table[125] = {false}; // default: false
void (*fn_table[36])();
```

Difficult to construct pre-contexts to trigger bugs

```
for (int i=0; i<125; ++i)
    flag_table[i] = true;

switch (ctrl_code) {
    case 0x8fff23c4:
        for (int i=0; i<125; ++i)
            flag_table[i] = true;
    case 0x8fff23cc:
        ...
        if (flag_table[buf[1]])
            (*fn_table[buf[2]])();
}
```

should be executed to trigger the bug
Challenge 2: Difficulty in constructing pre-contexts to test targets

• Many functions and code blocks have **pre-contexts** to execute them correctly
  • Execution order to set up states (open before read), input validation (checksum), ...

• **Difficult to construct or guess pre-contexts**
Challenge 2: Difficulty in constructing pre-contexts to test targets

• Many functions and code blocks have pre-contexts to execute them correctly
  • Execution order to set up states (open before read), input validation (checksum), ...

Research goal: Can we make a concolic testing tool that
1) avoids path explosion and
2) constructs pre-contexts automatically?
Idea 1: Test paths likely having bugs first

• Prioritize **array and loop boundary states**
• Detect bugs due to a lack of proper boundary checks
Idea 2: Construct pre-contexts using real programs

• Let real programs run until they call target OS APIs
  • Would have prepared necessary conditions before calling the APIs (they will call open syscall before read syscall)

• Hook the API calls and initiate concolic testing
Promising results

• Implemented by modifying S2E and evaluated with Windows 7 and Windows Server 2008
• Found 21 unique crashes in six device drivers
  • Two local privilege escalation vulnerabilities
  • Information disclosure in a crypto driver
Overview of CAB-Fuzz

1. Symbolization (synthetic or on-the-fly)
2. Concolic execution
3. Analyzing crashes

COTS OSes (e.g., Windows) -> Crash DB (e.g., memory dump) -> Vulnerability information (e.g., classification)

BSOD detector
Synthetic symbolization with S2E

```c
ulong ctrl_code = 0; ulong in_buf[IN_BUF_SIZE] = {0};
NtCreateFile(&device_handle,..., &object_attributes,...);
s2e_make_symbolic(&ctrl_code, sizeof(ctrl_code), "code");
s2e_make_symbolic(&in_buf, sizeof(in_buf), "buf");
NtDeviceIoControlFile(
    device_handle, NULL, NULL, NULL,
    &io_status_block,
    ctrl_code, &in_buf, IN_BUF_SIZE,
    &out_buf, OUT_BUF_SIZE);
```
Synthetic symbolization with S2E

```c
ulong ctrl_code = 0; ulong in_buf[IN_BUF_SIZE] = {0};

NtCreateFile(&device_handle,..., &object_attributes,...);

s2e_make_symbolic(&ctrl_code, sizeof(ctrl_code), "code");
s2e_make_symbolic(&in_buf, sizeof(in_buf), "buf");

NtDeviceIoControlFile(
    device_handle, NULL, NULL, NULL,
    &io_status_block,
    ctrl_code, &in_buf, IN_BUF_SIZE,
    &out_buf, OUT_BUF_SIZE);
```

Specify target API
Synthetic symbolization with S2E

```c
ulong ctrl_code = 0; ulong in_buf[IN_BUF_SIZE] = {0};

NtCreateFile(&device_handle, ..., &object_attributes, ...);

s2e_make_symbolic(&ctrl_code, sizeof(ctrl_code), "code");
s2e_make_symbolic(&in_buf, sizeof(in_buf), "buf");

NtDeviceIoControlFile(device_handle, NULL, NULL, NULL,
                        &io_status_block, 
                        ctrl_code, &in_buf, IN_BUF_SIZE,
                        &out_buf, OUT_BUF_SIZE);
```

Specify target drivers

Symbolize two arguments

Specify target API
Synthetic symbolization with S2E

```c
ulong ctrl_code = 0; ulong in_buf[IN_BUF_SIZE] = {0};
NtCreateFile(&device_handle,..., &object_attributes,...);
s2e_make_symbolic(&ctrl_code, sizeof(ctrl_code), "code");
s2e_make_symbolic(&in_buf, sizeof(in_buf), "buf");
NtDeviceIoControlFile(
    device_handle, NULL, NULL, NULL,
    &io_status_block,
    ctrl_code, &in_buf, IN_BUF_SIZE,
    &out_buf, OUT_BUF_SIZE);
```

Specify target drivers
Symbolize two arguments
Specify target API
Don’t symbolize the size to avoid path explosion
Array-boundary prioritization

- Concretize the **lowest** and **highest** addresses of symbolic memory first
- Compute the boundary addresses using KLEE solver’s `getRange` function
  - For symbolic memory triggering a state fork at least twice
Loop-boundary prioritization

• Concretize a loop as **no loop execution**, a **single** execution, and the **maximum** executions

• Use a fork-and-kill approach to deal with unclear loop conditions and structures
  • Let a loop execute until it forks no more states (maximum)
  • Kill or pause uninteresting loop states
Prioritization reduces # of state forks to detect a bug

... if (buf[0]>246 &&
    buf[1]>124 &&
    buf[2]>36)
    return -1;
if (flag_table[buf[1])
    (*fn_table[buf[2]])();
for (int i=1; i<=buf[0];
    ++i) {...}...
Prioritization reduces # of state forks to detect a bug

```c
... if (buf[0]>246 && buf[1]>124 && buf[2]>36) return -1;
if (flag_table[buf[1]]) (*fn_table[0])();
for (int i=1; i<=buf[0]; ++i) {...}
... if (flag_table[buf[1]]) (*fn_table[0])();
for (int i=1; i<=0; ++i) {...}
... if (flag_table[buf[1]]) (*fn_table[0])();
for (int i=1; i<=1; ++i) {...}
... if (flag_table[buf[1]]) (*fn_table[0])();
for (int i=1; i<=246; ++i) {...}
```
Prioritization reduces # of state forks to detect a bug

if (flag_table[buf[1]) (*fn_table[0])();
for (int i=1; i<=buf[0];
++i) {...}

if (flag_table[buf[1])
(*fn_table[0])();
for (int i=1; i<=0;
++i) {...}

if (flag_table[buf[1])
(*fn_table[0])();
for (int i=1; i<=1;
++i) {...}

if (flag_table[buf[1])
(*fn_table[0])();
for (int i=1; i<=246;
++i) {...}
On-the-fly symbolization

Run a real program
On-the-fly symbolization

Run a real program

Interact with kernel

Kernel-space

Kernel

User-space

Program

Kernel data structures

Global variables
On-the-fly symbolization

Run a real program

Call a target function with valid arguments

Interact with kernel

User-space

Kernel-space

Program

Kernel

Kernel data structures

Global variables

Hooking
On-the-fly symbolization

Run a real program

Call a target function with valid arguments

Interact with kernel

Initiate concolic testing
Evaluation

• How efficiently did CAB-Fuzz detect the known vulnerability (NDProxy)?
• How many new crashes did CAB-Fuzz discover?
• What particular characteristics did the newly discovered crashes exhibit?
CAB-Fuzz crashed NDProxy within two seconds

- No prioritization: 2 s
- Prioritization: ~2 hours!

- No prioritization: 17
- Prioritization: 78
CAB-Fuzz found 21 new crashes

• Synthetic symbolization
  • 274 device drivers in Windows 7 and Windows Server 2008

• On-the-fly symbolization
  • 16 real programs and 15 drivers the programs used

➢ Found 21 crashes in six among the drivers
CAB-Fuzz found 21 new crashes

![Graph showing the number of crashes for different modules and prioritization methods.]

- NDIS: 10 crashes
- SrvAdmin: - crashes
- NSI: 0 crashes
- ASYNCMAC: - crashes
- FileInfo: 0 crashes
- ehdrv: 0 crashes

Legend:
- Blue: No prioritization
- Orange: Prioritization
- Gray: On-the-fly

Synthetic symbolization
CAB-Fuzz found 21 new crashes

A lack of memory (path explosion)

Synthetic symbolization

- No prioritization
- Prioritization
- On-the-fly
CAB-Fuzz found 21 new crashes

<table>
<thead>
<tr>
<th></th>
<th>No prioritization</th>
<th>Prioritization</th>
<th>On-the-fly</th>
</tr>
</thead>
<tbody>
<tr>
<td>NDIS</td>
<td>-</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>SrvAdmin</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NSI</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ASYNCMAC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FileInfo</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ehdrv</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A lack of memory (path explosion)

Pre-contexts needed

Synthetic symbolization
CAB-Fuzz found 21 new crashes

A lack of memory (path explosion)
Invalid pre-context needed
Pre-contexts needed

Synthetic symbolization

- No prioritization
- Prioritization
- On-the-fly

#Crashes

<table>
<thead>
<tr>
<th>Component</th>
<th>Crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NDIS</td>
<td>10</td>
</tr>
<tr>
<td>SrvAdmin</td>
<td>4</td>
</tr>
<tr>
<td>NSI</td>
<td>0</td>
</tr>
<tr>
<td>ASYNCMAC</td>
<td>0</td>
</tr>
<tr>
<td>FileInfo</td>
<td>0</td>
</tr>
<tr>
<td>ehdrv</td>
<td>0</td>
</tr>
</tbody>
</table>
CAB-Fuzz found 21 new crashes

A lack of memory (path explosion)
Invalid pre-context needed
Pre-contexts needed

No program for on-the-fly

Synthetic symbolization

<table>
<thead>
<tr>
<th>Module</th>
<th>No prioritization</th>
<th>Prioritization</th>
<th>On-the-fly</th>
</tr>
</thead>
<tbody>
<tr>
<td>NDIS</td>
<td>0</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>SrvAdmin</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>NSI</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ASYNCMAC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FileInfo</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ehdrv</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
CAB-Fuzz found 21 new crashes

A lack of memory (path explosion)

No program for on-the-fly

Invalid pre-context needed

Pre-contexts needed

Synthetic and on-the-fly symbolizations are complementary to each other
What pre-contexts did drivers need?

• Selectively loaded (FileInfo)
  - Filesystem filter driver by Microsoft
  - Loaded only when a certain program started

• Access controlled (ehdrv)
  - Driver installed by antivirus software ESET Smart Security
  - Only accessible by the antivirus software itself
Prioritization reduced CPU time and memory usage

Time (s)

- NDIS
- SrvAdmin
- NSI

No prioritization  Prioritization

Memory (MB)

- NDIS
- SrvAdmin
- NSI

No prioritization  Prioritization
Limitations

• Reduce code coverage when prioritizing symbolic memory with instruction addresses (e.g., jump table)

• Cannot get boundary states from flexible data structures (e.g., linked list)
Limitations

• Have difficulties in regenerating on-the-fly-driven crashes
  • Lack of explicit control of pre-contexts construction

• Need to specify target APIs and programs
Conclusion

- CAB-Fuzz: A practical concolic testing tool for COTS OS
  - Check potentially vulnerable paths first
  - Analyze COTS OS without debug information and pre-contexts
- Found 21 crashes including three vulnerabilities with CVEs