Excite project:
All the truth about Symbolic Execution for BIOS security

Ilia Safonov
Alex Matrosov

with Lee Rosenbaum and Zhenkun Yang
Acknowledgement

- **Excite team:**
  - Mark Tuttle
  - Lee Rosenbaum
  - Ilia Safonov
  - Zhenkun Yang

- **SeCoE team:**
  - Alex Matrosov
  - Rodrigo Branco
  - Topher Timzen

- **Our colleagues in Intel:**
  - Alex Bazhaniuk
  - Vincent Zimmer
  - John Loucaides
  - Brian Delgado
  - Alexey Kovalev
  - Jin Yang
All fragments of source code as well as memory dumps relate to open-source projects:

- Firmware for MinnowBoard: minnowboard.org
- EDK2: http://www.tianocore.org/edk2/

Otherwise it is artificial examples, which have no relations with Intel products.
UEFI Firmware Security Validation Challenges
Validation Challenges

- UEFI Firmware code base is huge (millions of lines of code)
  - SMM code always in the most critical scope
  - Legacy code/support makes validation more fun ;)
- Boot procedure after power on, sleep and hibernate differentials. It requires additional effort for fuzzing
  - Code coverage can be different even for the same code due to a huge number of global variables and hardware configuration
SMM Specifics
System Management Mode (SMM)

- System Management Mode (SMM) is a highly-privileged mode of CPU.
- SMRAM is a range in DRAM reserved for SMI handlers (protected for access from the OS).
System Management Interrupt (SMI) Handlers

- Protected SMRAM
  - SMI code lives here

- SMRAM
  - SMM state save area
  - SMI handlers

- SMBASE + FFFFh
- SMBASE + FC00h
- SMBASE + 8000h
- SMBASE
• CPU stores current value of SMBASE in SMM save state area on SMI and restores it on RSM
CommBuffer is a memory buffer used as a communication protocol between OS runtime and DXE SMI handlers. Pointer to CommBuffer is stored in “UEFI” ACPI table in ACPI memory.

Contents of CommBuffer are specific to SMI handler. For example Variable SMI handler read variable GUID, Name and Data from CommBuffer.
Pointer Arguments to SMI Handlers
(CommBuffer notation)

Normal DRAM

- ACPI Table
  - Header
  - Identifier
  - DataOffset
  - SwSmi
  - Smm Communicate Buffer Pointer Addr
  - Invocation register

- ACPI NVS
  - Smm Communicate Buffer Pointer

Smm Communicate Buffer
- GUID
- Length
- Data
Pointer Arguments to SMI Handlers (CommBuffer notation)
Pointer Arguments to SMI Handlers (CommBuffer notation)

- **Example:** VariableAuthenticated SMI Handler reads/writes UEFI variables from/to CommBuffer.
Excite project
Excite project combines dynamic symbolic execution and guided fuzzing for automatic test case generation, and our flow uses Intel Virtual Platform to dump BIOS data, replay tests (measuring code coverage) and find vulnerabilities.
SMM in current scope of Excite

- SMI call-out vulnerabilities for SMI handlers with CommBuffer notations:
  - Excite check execution outside SMRAM
  - Excite check memory access outside of valid regions:
    - SMRAM
    - MMIO
    - ACPI_NVS
    - BIOS reserved
- Excite does not check security configuration bits for the platform
Source code repository → BIOS build → Dump SMRAM using Simics → Harness generation

Playing tests and fuzzing in Simics → Test cases

Code coverage report → Issues report → Updated test cases

Symbolic Execution using S2E
Symbolic Execution Technique for Automatic Test Case Generation
Symbolic execution is a technique that can be used for automatic test generation which provides high code coverage.

The main idea is to substitute parameters of functions with symbolic values and then execute the function parametrically such that [1]:

- the values of all variables are computed as symbolic expressions over the symbolic input values
- the execution can proceed along any feasible path
Symbolic Execution Tree

- Symbolic Execution Tree (SET) is created during symbolic execution
  - nodes of a SET represent the symbolic program states and edges represent transitions between these states
  - symbolic state consists of symbolic variables, a program location and a path constraint (PC) which is the conjunction of all the logical constraints collected over the variables to reach that program location
  - the paths of a SET characterize all execution paths

- In Static Symbolic Execution, SET is constructed for the whole program under analysis and without usage of concrete values of variables
Path constraint (PC)

- e.g.: $X > Y \land Y + X \leq 8$
- solution of the constraint is a set of assignments, one for each variable that makes the constraint satisfiable
- \{X = 3, Y = 2\} is a solution but \{X = 6, Y = 5\} is not

A constraint solver is a tool that finds satisfying assignments for a constraint, if it is satisfiable

SMT (Satisfiability Modulo Theory) solver is used to check the satisfiability of each PC
int some_func(uint x, uint y)
{
    if (x > y)
    {
        x = x - y;
    }
    else
    {
        x = y - x;
        if (x >= 0)
        {
            x *= y;
        }
        else
        {
            x /= y;
        }
    }
    return x;
}

```
x: X, y: Y
PC: true
1: if(x>y) - then
2: x = x - y;
```

```
x: X-Y, y: Y
PC: X>Y
5: if(x>=0) - then
6: x*=y;
```

```
x: (X-Y)*Y, y: Y
PC: X>Y^X-Y>=0
```

```
x: (Y-X)*Y, y: Y
PC: X<X^Y-Y<=0
```

```
x: X-Y, y: Y
PC: X>Y
5: if(x>=0) - else
6: x*=y;
```

```
x: (X-Y)*Y, y: Y
PC: X>Y^X-Y>=0
```

```
x: Y-X, y: Y
PC: X<=Y
5: if(x>=0) - then
6: x*=y;
```

```
x: (Y-X)*Y, y: Y
PC: X<=Y^Y-X>=0
```

```
x: X-Y, y: Y
PC: X<=Y
4: x = y - x;
```

```
x: Y-X, y: Y
PC: X<=Y
5: if(x>=0) - else
---
Unsatisfiable PC
>> Infeasible path
```

```
x: (Y-X)*Y, y: Y
PC: X<=Y^Y-X>=0
```

```
x: X-Y, y: Y
PC: X<=Y
5: if(x>=0) - else
---
Unsatisfiable PC
>> Infeasible path
```

---
"Dead Code"
Limitations of Static SE

- Inability to solve very complex and non-linear constraints
  - $X \% 9 > 3 \land Y > 15$
  - $(X >> 4) \land 2 < Y$

- Inability to handle external calls
  - $f(X) > 0$, where function $f$ is inaccessible for static analysis

- Inability to deal with parallel execution

Mitigation of the limitations: Dynamic Symbolic Execution or Concolic Testing
Dynamic Symbolic Execution

Concolic technique performs symbolic execution dynamically along an execution path of a concrete input and generates tests one by one for each path.

\[ PC = pc_1 \land pc_2 \land pc_3 \land pc_4 \land \ldots \]

\[ PC' = pc_1 \land pc_2 \land \neg pc_3 \Rightarrow I_2 \]
Path explosion challenge

- Number of feasible paths grows exponentially with the size of the code
- Loops lead to a huge number of test cases
- The number can be even infinite for programs with unbounded loops and recursion
- Symbolic execution engine can get stuck due to polling loops in firmware
Search strategies for SET

Which path should be selected?

- Mainly random search generates a test set with a better code coverage, but such test set is not deterministic
- De-randomization is required for reproducibility, but it is palliative
Combining Symbolic Execution and Fuzzing
Combining Symbolic Execution and fuzzing

```c
typedef struct {
    int signature;
    int num;
} SOME_BUF;

int some_function(SOME_BUF *pbuf)
{
    if (pbuf->signature == 0x12345)
    {
        return (int)sqrt((double)pbuf->num);
    }
    return 0;
}

Fuzzing of tests generated by symbolic execution is a better way!
```

Unlikely a fuzzer would generate the constant 0x12345. In contrast, symbolic execution creates a test for covering code inside.

Negative pbuf->num leads to error!

Unlikely the symbolic execution creates a test with negative pbuf->num. Probability of generating negative pbuf->num by fuzzing is high.
Fuzzing guided by code coverage

Similar scheme to American Fuzzy Lop (AFL) fuzzer [2]
Gray-box fuzzing

1. Variation of CommBufSize from 1 to 100 and FunctionId in CommBuf from 0 to 20

2. Application of the following fuzzing strategies for tests collected in a pool as long as we have improvement in code coverage; the strategies were inspired by AFL but with taking into account SMM specific:

   • Walking 1 bit flip, step = 1 bit
   • Walking byte flips: 1, 2, 4 and 8 bytes, step = 1 byte
   • Walking insertion of addresses inside SMRAM, outside SMRAM in ACPI_NVS and outside SMRAM in “available” memory region, step = 1 byte
   • Changing of FunctionId and cyclic rotation of CommBuf fragment
   • Random splicing of test cases
   • Walking addition and subtraction of small constant for byte, word, dword and qword, step = 1 byte
typedef struct {
    unsigned short   Bus;
    unsigned int     Device;
    unsigned short   Port;
    unsigned int     Function;
    char             Password[48];
} COMM_BUF_4_SOME_HANDLER;

We know the format of CommBuffer for each handler:
• meaning of fields
• data types
• sometimes ranges of data
• interesting constants, for example GUIDs, addresses inside and outside SMRAM

It is possible to do a better fuzzing based on a priori knowledge!
Symbolic Execution Engines
# Open-source Symbolic Execution tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Architecture / Language</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>jCUTE</td>
<td>Java</td>
<td><a href="https://github.com/osl/jcute">https://github.com/osl/jcute</a></td>
</tr>
<tr>
<td>Otter</td>
<td>C</td>
<td><a href="https://bitbucket.org/khooyp/otter/overview">https://bitbucket.org/khooyp/otter/overview</a></td>
</tr>
<tr>
<td>KLEE</td>
<td>llvm</td>
<td><a href="http://klee.github.io/">http://klee.github.io/</a></td>
</tr>
<tr>
<td>S2E</td>
<td>binary x86, x86-64, ARM</td>
<td><a href="http://s2e.epfl.ch">http://s2e.epfl.ch</a></td>
</tr>
<tr>
<td>Triton</td>
<td>binary x86, x86-64</td>
<td><a href="http://triton.quarkslab.com">http://triton.quarkslab.com</a></td>
</tr>
<tr>
<td>angr</td>
<td>libVEX based</td>
<td><a href="http://angr.io/">http://angr.io/</a></td>
</tr>
</tbody>
</table>
S²E – Selective Symbolic Execution

• S²E is based on KLEE symbolic execution engine and QEMU virtual machine [4]
• Flexible architecture with plug-ins
1. Mapping dump of SMRAM to harness memory space by mmap
2. Making symbolic of input parameters of a SMM handler: CommBuffer and size of CommBuffer
3. Set RSP value of stack pointer in SMM handler captured in boot procedure
4. Invocation by pointer of SMM handler from mapped SMRAM
5. Return back RSP of the harness program
Excite details
Intel Virtual Platform

- Perfect simulation of hardware
- Boot after power on, sleep and hibernate
- Dump SMRAM, memory map and other parameters
- Disassembling
- Replaying test cases generated by $s^2e$ and fuzzing
- Dynamic check of accesses out of allowable memory regions and SMRAM call-outs
- Measurement of code coverage without instrumenting of BIOS
Dump SMRAM

• Simics has access to all memory, even to SMRAM when SMRAM is locked
• Base address and size of SMRAM are captured from serial boot log
• SMRAM is dumped just after SMRAM is locked, trigger of it is message in serial boot log

Fragment of boot log for open-source MinnowMax BIOS

SMM IPL opened SMRAM window
SMM IPL found SMRAM window 3B001000 - 3B3FFFFFF
SMM IPL loading SMM Core at SMRAM address 3B3F6000
SMM IPL calling SMM Core at SMRAM address 3B3F62C0
mMaximumSupportAddress = 0xFFFFFFFF
InstallProtocolInterface: F4CCBF7-F6E0-47FD-9DD4-10A8F150C191 39CB9440 ...
SmmLockBox SmmLockBoxHandler Exit
SmmLockBoxDxeLib SetLockBoxAttributes - Exit (Success)
SMM IPL locked SMRAM window
Scanning SMRAM

- Parsing PECOFF and extraction of `.text` & `.rdata` sections
- Several SMI handlers entry points can be found in SMI_HANDLER structures which has `smih` signature
- Other SMI handler entry points can be found in DATABASE_RECORD structures which has `DBRC` signature [7, 8]
SMI_HANDLER and SMI_ENTRY structures

```c
typedef struct {
    UINTN Signature;
    LIST_ENTRY Link;
    SMM_HANDLER_ENTRY_POINT2 Handler;
    SMI_ENTRY *SmiEntry;
} SMI_HANDLER;

typedef struct {
    UINTN Signature;
    LIST_ENTRY AllEntries;
    EFI_GUID HandlerType;
    LIST_ENTRY SmiHandlers;
} SMI_ENTRY;
```

EFI scripts for IDA Pro [5] contains a broad collection of known GUIDs.
Playing and tracing test cases

EFI_STATUS EFIAPI SmmHandler (
    IN EFI_HANDLE DispatchHandle,
    IN CONST VOID *RegisterContext,
    IN OUT VOID *CommBuf,
    IN OUT UINTN *CommBufSize );

- Simics can trace all executed instructions and memory accesses
- Captured Issues:
  1. Call-out SMM
  2. Memory access out of allowable regions (SMRAM, MMIO, ACPI_NVS, BIOS reserved)
  3. ASSERT
Code Coverage measuring

\[
\text{Code coverage} = \frac{\sum \text{instructions, which were executed}}{\sum \text{reachable instructions in computing tree}} \times 100\%
\]

- Dynamic tracing for calculating the sum of executed instruction: we just mark addresses of executed instructions in Simics
- Traversal of a computing tree on a disassembled code for calculating the sum of reachable instructions
- Distribution of statement code coverage from assembler level to C-source level by Microsoft dbghelp.dll
- Estimation of branch/decision coverage [3]
- Measuring of function coverage
Traversing a computing tree

Challenge in processing of indirect calls and jumps:

```c
    call qword ptr 12[rcx]
    jmp rax
```

We collect addresses of indirect calls and jumps during playing of test cases, addresses are stored in a map that is used in a recursive procedure for traversal of computing:

```c
std::map<int*, std::set<int*>* ic_map;
```

Pseudo-code of traversal of computing tree based on disassembled code

```c
Set_label( cur_addr ) {  
    if (asm_label[ cur_addr ] != 0) return
    while (true) {        
        asm_label[ cur_addr ] == 1  
        if (instruction[ cur_addr ] == "ret")    
            return      
        if (instruction[ cur_addr ] == "call"  
            or instruction[cur_addr ] == cond. jump) {  
            extraction of destination address:  
            from instruction for direct call  
            from map for indirect call  
            Set_label( destination_addr )
        }
        if (instruction[cur_addr ] == "jmp") {  
            extraction of destination address:  
            from instruction for direct call  
            from map for indirect call  
            Set_label( destination_addr )    
            return
    }
    cur_addr = get_next_address
}
```
How long it works

• Now we deal with 10-20 SMI handlers
• $s^2e$ generates about 20000 test cases per handler in 2 hour
• 3 hours are necessary for playing 20000 test cases in Simics and at least 5 additional hours for fuzzing
• **Total time in one thread**: $10 \times (2 + 3 + 5) = 100$ hours $\approx 4$ days
• Fortunately, each handler can be processed independently in parallel
• Test cases playing and fuzzing can be parallelized as well
• We use 2 servers, each one has 54 CPU and 64 GB RAM
• **Total time for 2 servers**: $< 4$ hours
Parallel execution

- Dump SMRAM
  - Tests cases generation for handler 1
  - Tests cases generation for handler N
    - Tests cases playing for handler 1, thread 1
    - Tests cases playing for handler N, thread P
    - Fuzzing for handler 1, thread 1
    - Fuzzing for handler N, thread P
    - Fuzzing for handler 1, thread M
    - Fuzzing for handler N, thread T
Results
### Code Coverage Outcomes

<table>
<thead>
<tr>
<th>SMM Handler</th>
<th>Baseline(^1)</th>
<th>Simple BlackBox Fuzzing(^2)</th>
<th>Symbolic Execution</th>
<th>Symbolic Execution and Fuzzing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handler 1</td>
<td>0 %</td>
<td>7 %</td>
<td>88 %</td>
<td>90 %</td>
</tr>
<tr>
<td>Handler 2</td>
<td>0 %</td>
<td>5 %</td>
<td>58 %</td>
<td>65 %</td>
</tr>
<tr>
<td>Handler 3</td>
<td>49 %</td>
<td>24 %</td>
<td>57 %</td>
<td>60 %</td>
</tr>
<tr>
<td>Handler 4</td>
<td>46 %</td>
<td>3 %</td>
<td>51 %</td>
<td>55 %</td>
</tr>
<tr>
<td>Handler 5</td>
<td>0 %</td>
<td>38 %</td>
<td>47 %</td>
<td>47 %</td>
</tr>
</tbody>
</table>

1. Code coverage is measured in normal boot process after power on
2. 50000 random tests
Code coverage report

Asm statement coverage = 80.2%
Source statement coverage = 89.0%
Branch coverage = 44.3%
Function coverage = 90.0%

```
len = GetPathSize(path);
LIST_FOR_EACH (entry) {
    list = GET_LIST (entry, link);
    if (_memcmp(list->node.path, path, len) == 0) {
        _memcpy(list->node.pwd, password, PWD_LEN);
        return SUCCESS;
    }
}

dev = AllocatePool (D_LIST_SIZE);
if (dev == NULL) {
    return ERROR;
}
_memcpy(dev->node.pwd, password, PWD_LEN);
```

The line was covered

Some lines in the macro were covered, some were not

The line wasn’t covered

There was only one decision TRUE or FALSE for the condition

Artificial example
We worked with well-tested production-level version of BIOS. So, the number of real issues is not high, but the issues were detected automatically.

<table>
<thead>
<tr>
<th>SMM Handler</th>
<th>Simple BlackBox Fuzzing</th>
<th>Symbolic Execution</th>
<th>Symbolic Execution and Fuzzing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handler 1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Handler 2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Handler 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Handler 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Handler 5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example of Issue report

1. Access outside of valid memory regions in `100.txt` on address 0x18f79859 (1 byte(s) 0)!

   0x8cd9cf78  mov al,byte ptr [rax]  line: 110  file: `\lib\module.c`

   BYTE SomeType ( void* Node )
   {
     ASSERT (Node != NULL);
     return ((SOME_STRUCT *)(Node))->Type;
   }

   Call stack:
   0x8cd9d038  line: 211  file: `\lib\module.c`

     ASSERT (Node != NULL);
     return (BOOLEAN) (SomeType (Node) == SOME_TYPE);

Note: Stack trace truncated for display, report contains the full stack trace.
Future plans

• Validation of integer/buffers overflows for checking memory corruptions in SMI Handlers inside SMM
• Support of more SMI handlers, selection of appropriate variables to be symbolic
• Increase of code coverage by means of more symbolic variables
• Experiments with other Symbolic Execution engines
• Investigation of approaches for testing BIOS beyond SMM
References

2. AFL: http://lcamtuf.coredump.cx/afl/
5. EFI scripts for IDA Pro: https://github.com/snare/ida-efiutils
7. Xeno Kovah and Corey Kallenberg, How Many Million BIOSes Would you Like to Infect?”, 2015
Thank you!