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

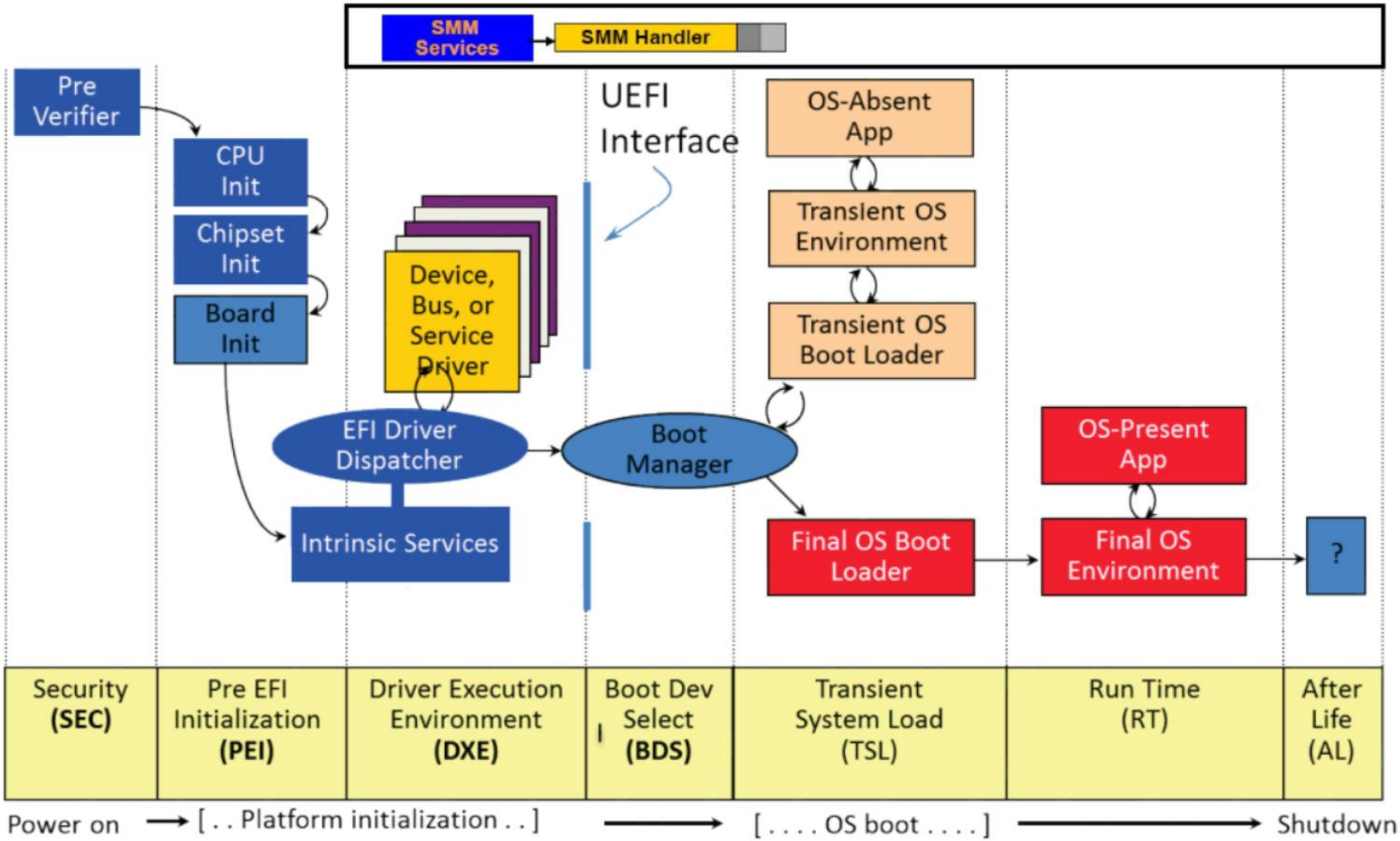
Notes about examples in the talk

- All fragments of source code as well as memory dumps relate to open-source projects:
    - Firmware for MinnowBoard:
    - 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

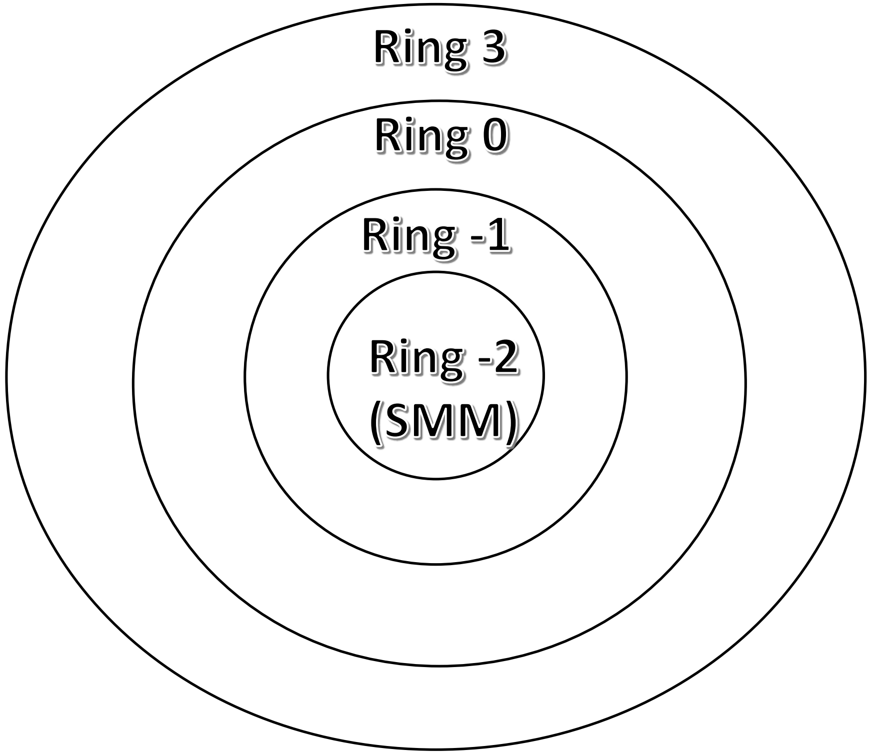
- UEFI Firmware code base is huge (millions of lines of code)
  - ✓ SMM code always in the most critical scope
  - ✓ Legacy code/support makes validation more fun ;)
- Boot procedure after power on, sleep and hibernate differentials. It requires additional effort for fuzzing
  - ✓ Code coverage can be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


SMM Services
SMM Handler
Pre Verifier
CPU Init
Chipset Init
Board Init
Device, Bus, or Service Driver
EFI Driver Dispatcher
Intrinsic Services
UEFI Interface
Boot Manager
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Final OS Boot Loader
OS-Present App
Final OS Environment
?
Security (SEC)
Pre EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Dev Select (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on
[. . Platform initialization . .]
[. . . . OS boot . . . .]
Shutdown

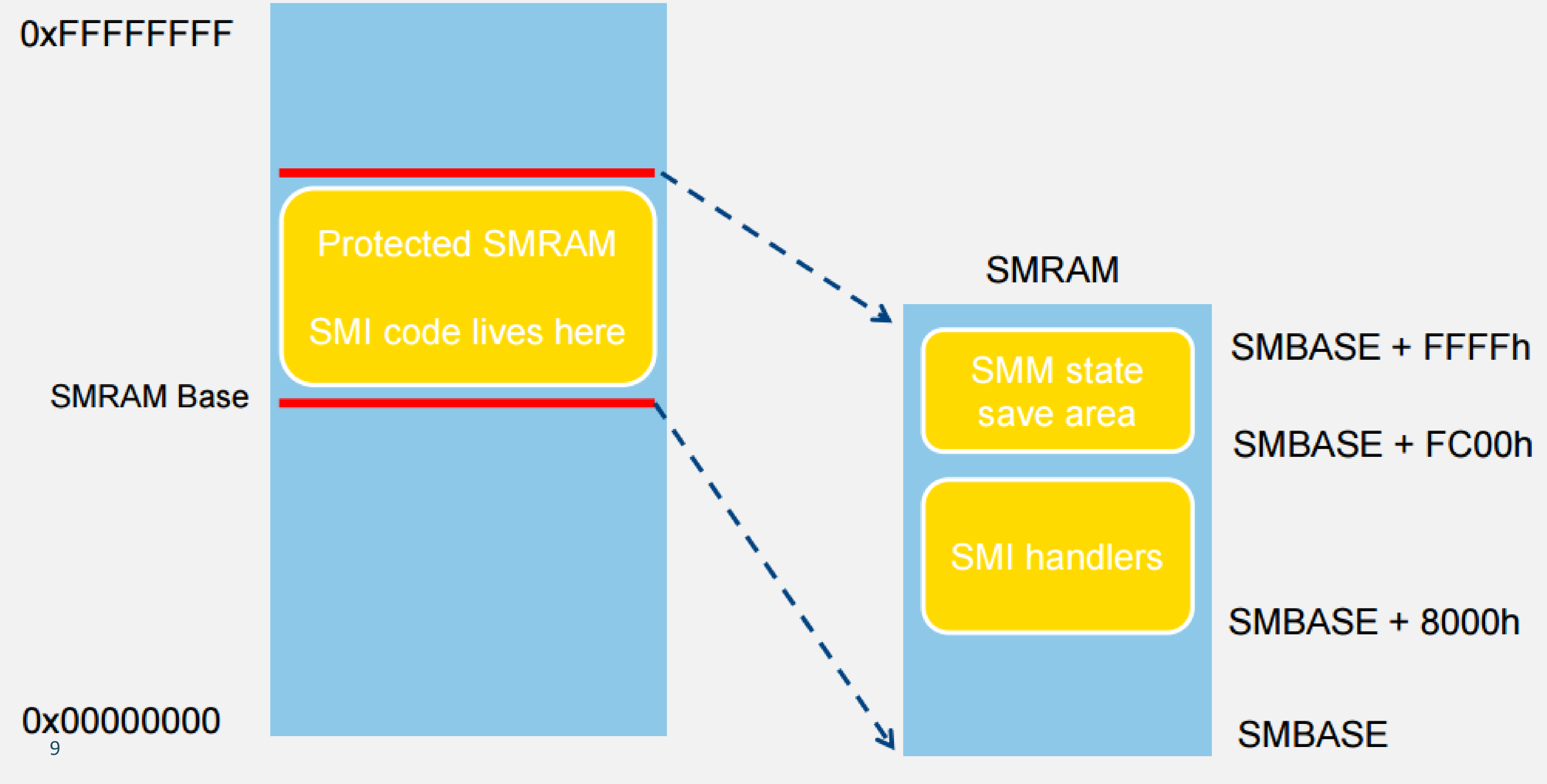
SMM Specifics

System Managment Mode (SMM)

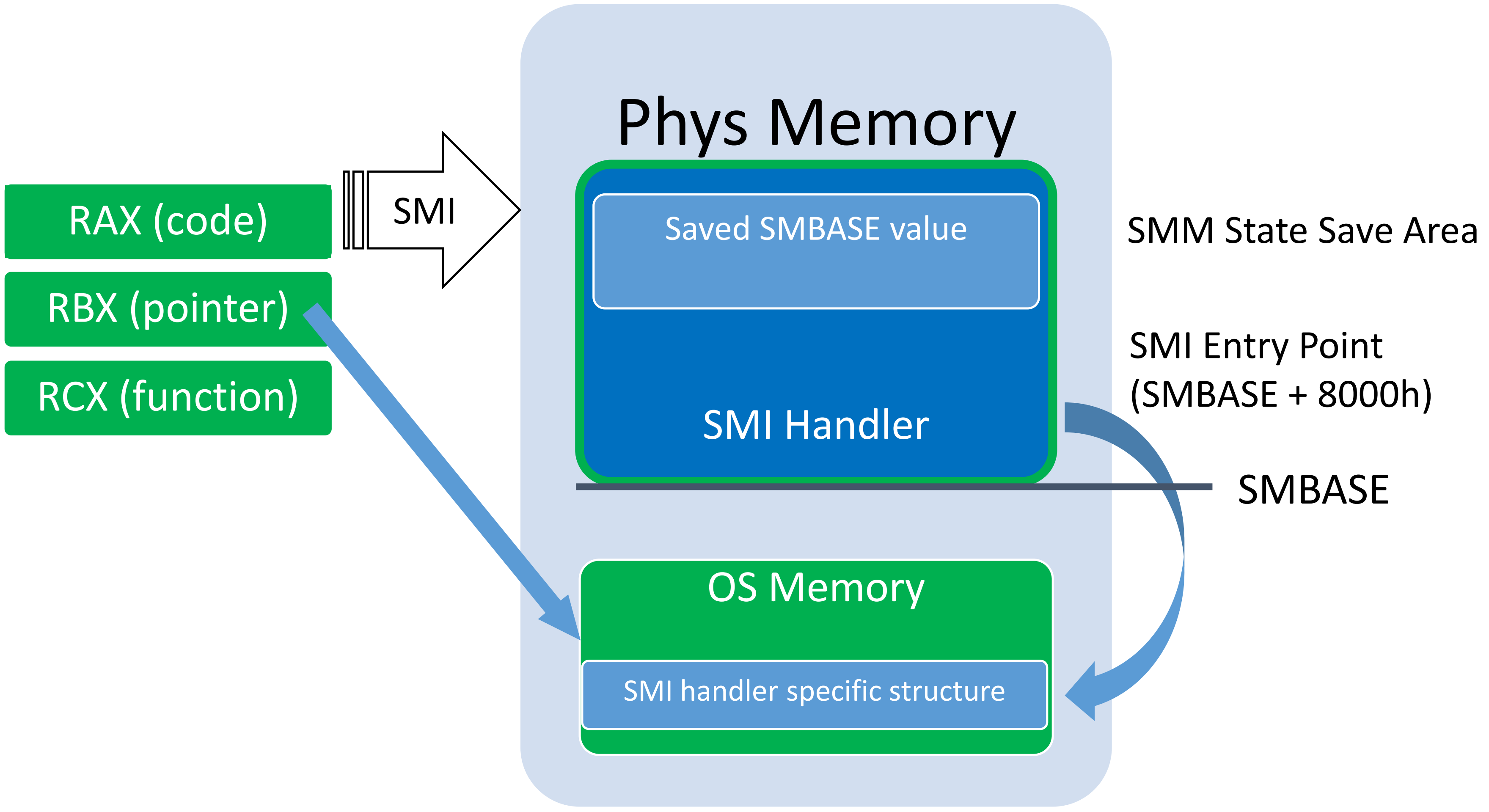
- System Management Mode (SMM) is a highly-privileged mode of CPU
- SMRAM is a range in DRAM reserved for SMI handlers (protected for access from the OS)



System Management Interrupt (SMI) Handlers



Pointer Arguments to SMI Handlers (Legacy notation)



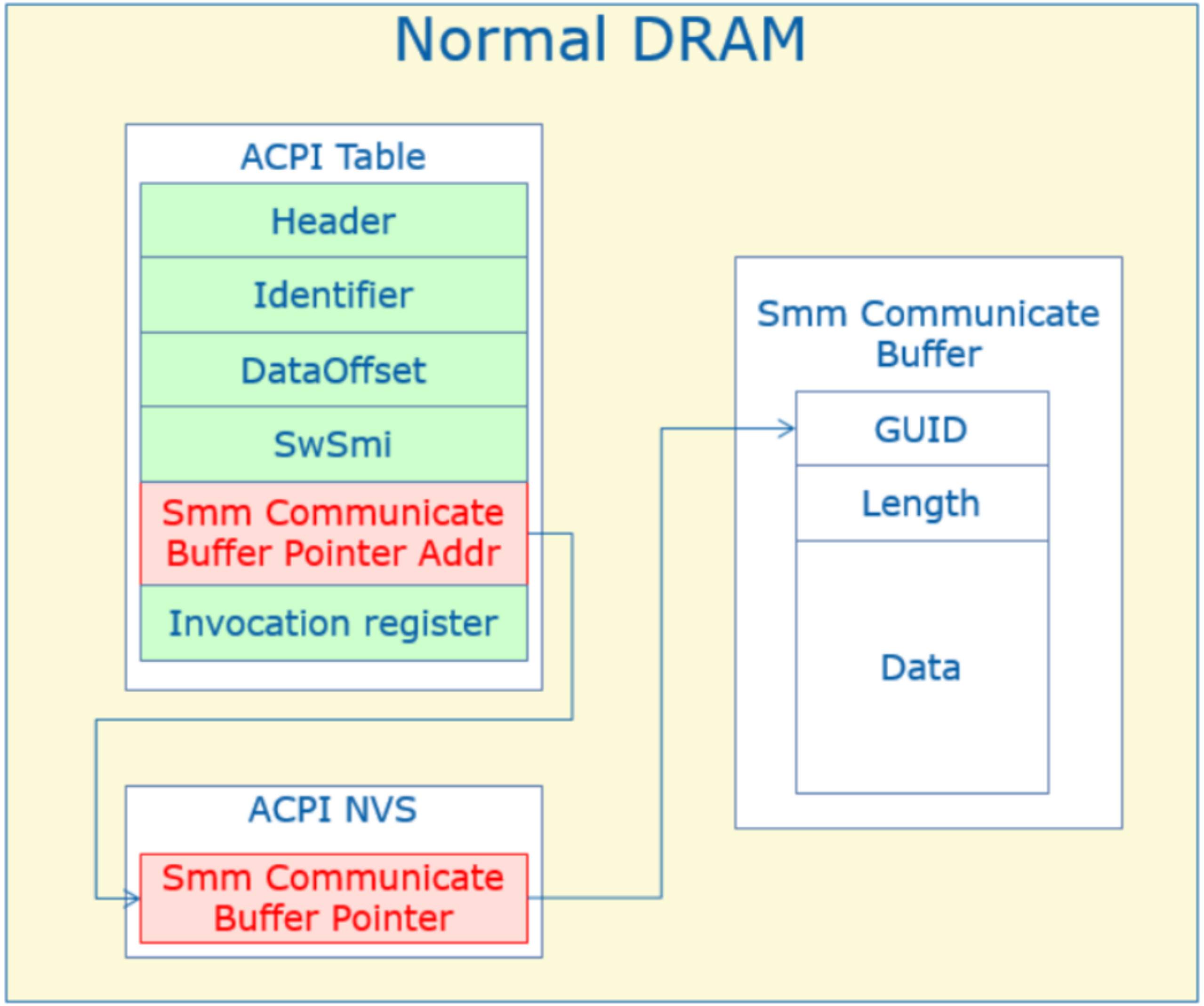
- CPU stores current value of SMBASE in SMM save state area on SMI and restores it on RSM

Pointer Arguments to SMI Handlers (CommBuffer notation)

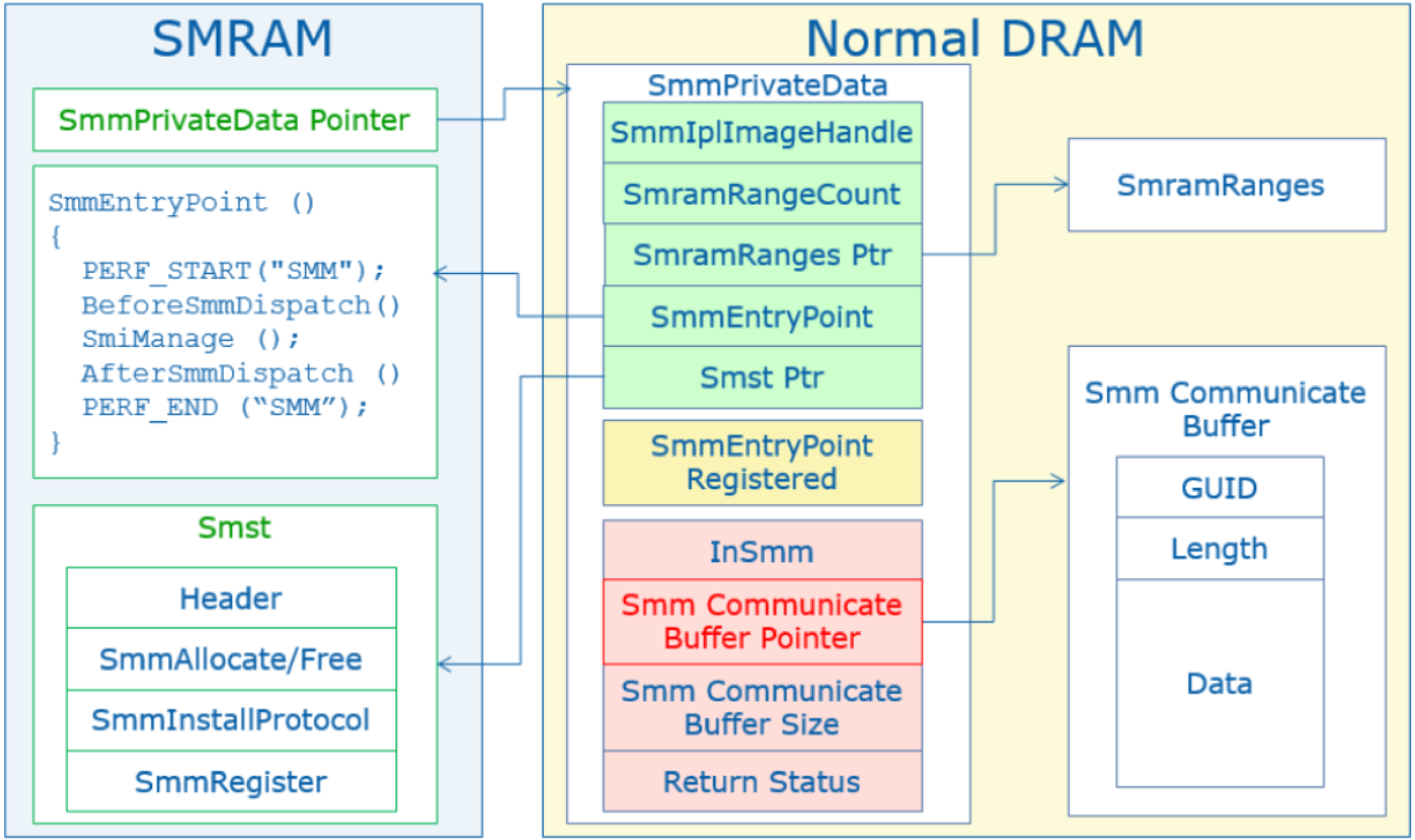
- CommBuffer is a memory buffer used as a communication protocol between OS runtime and DXE SMI handlers. Pointer to CommBuffer is stored in “UEFI” ACPI table in ACPI memory

- Contents of CommBuffer are specific to SMI handler. For example *Variable* SMI handler read variable GUID, Name and Data from CommBuffer

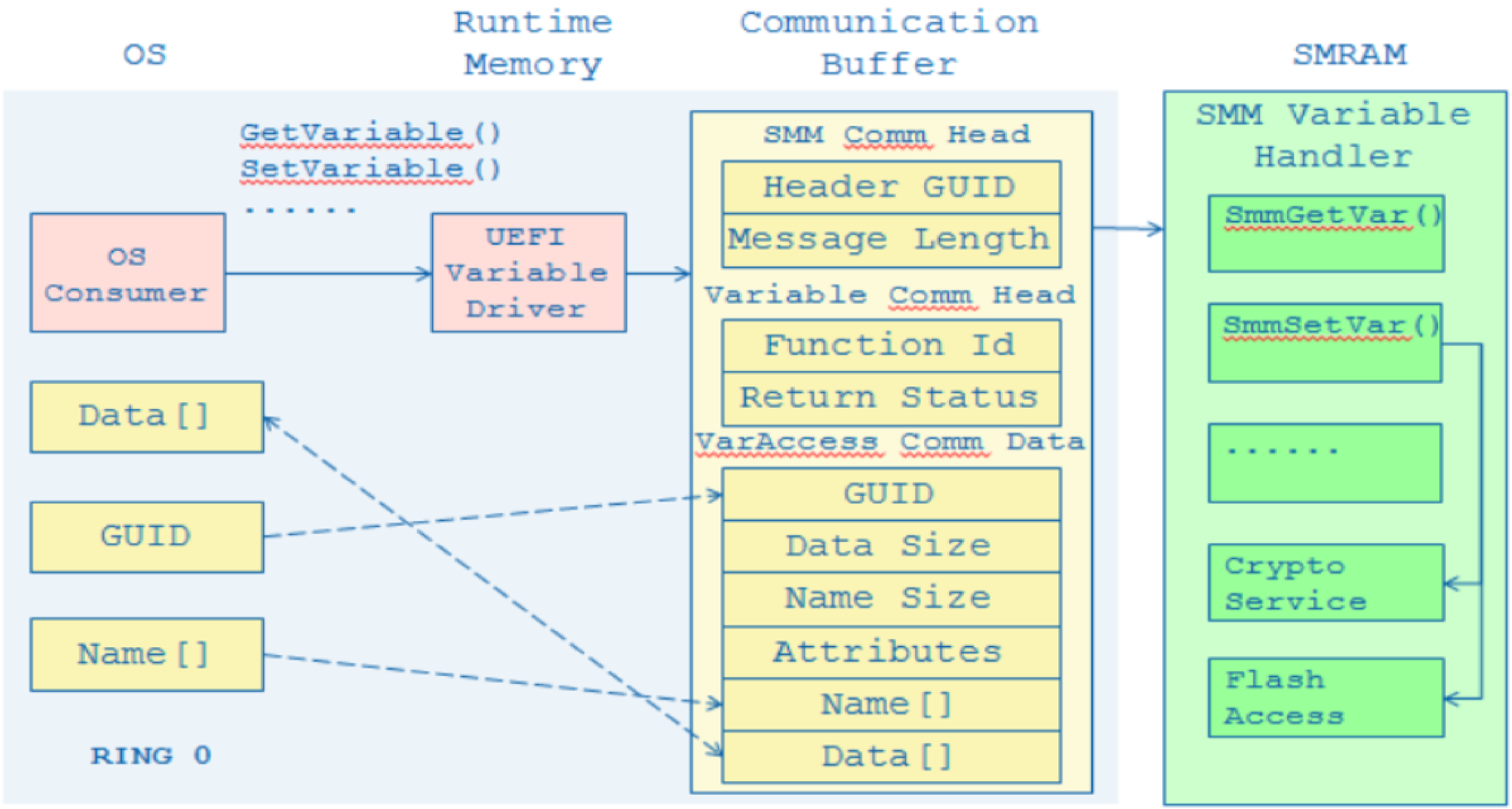
Pointer Arguments to SMI Handlers (CommBuffer notation)



Pointer Arguments to SMI Handlers (CommBuffer notation)



Pointer Arguments to SMI Handlers (CommBuffer notation)

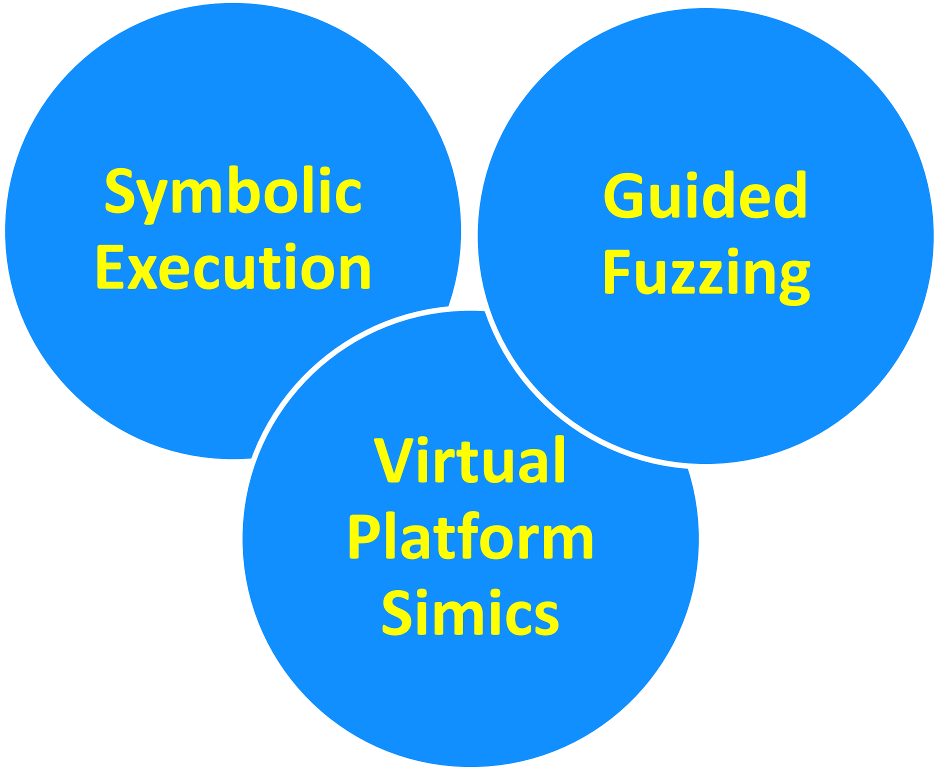


- **Example:** VariableAuthenticated SMI Handler reads/writes UEFI variables from/to CommBuffer

Excite project

Excite project

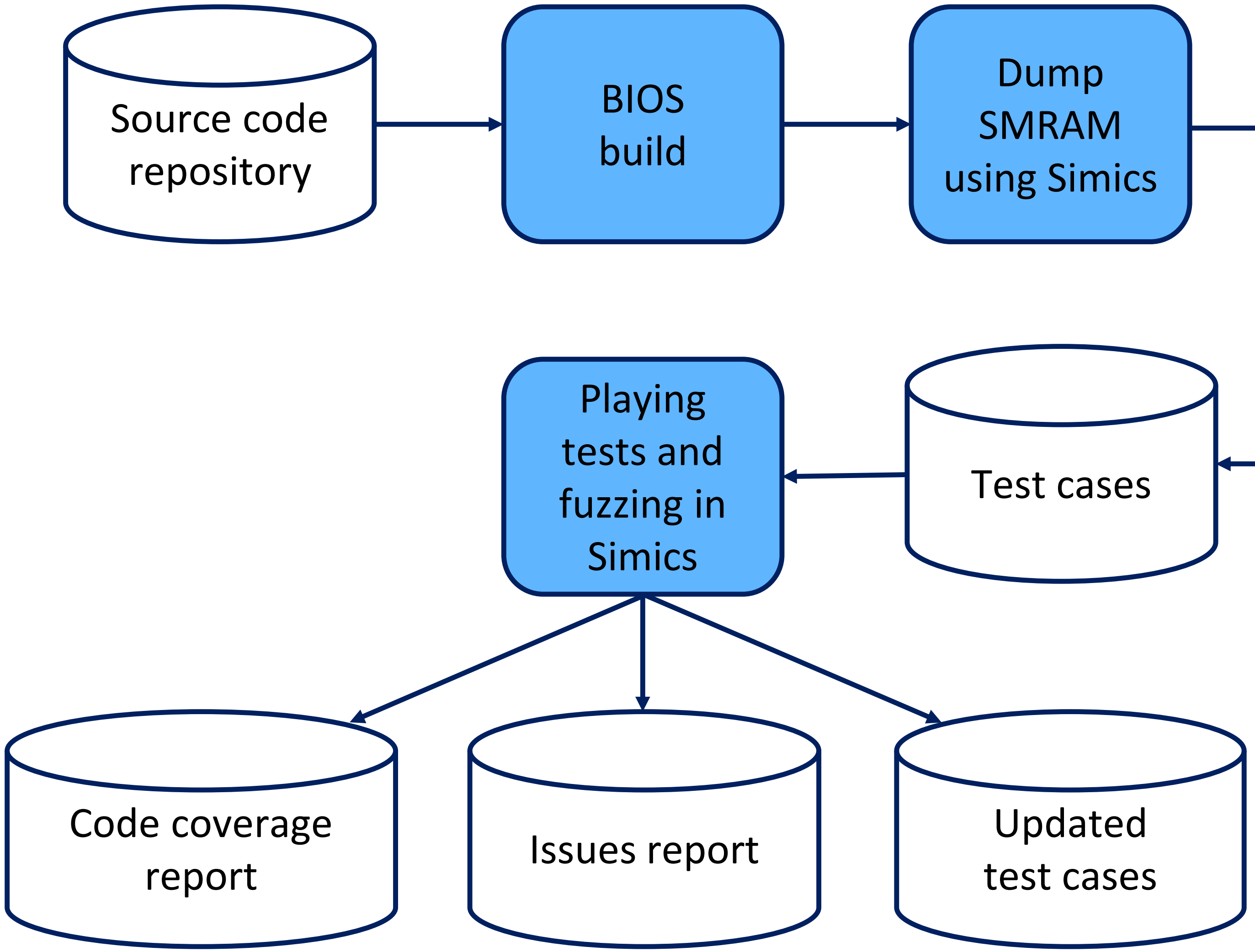
Excite project combines dynamic symbolic execution and guided fuzzing for automatic test case generation, and our flow uses Intel Virtual Platform to dump BIOS data, replay tests (measuring code coverage) and find vulnerabilities



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

Excite flow



Source code repository
BIOS build
Dump SMRAM using Simics
Harness generation
Symbolic Execution using S2E
Test cases
Playing tests and fuzzing in Simics
Code coverage report
Issues report
Updated test cases

Symbolic Execution Technique for Automatic Test Case Generation

Symbolic Execution

- 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

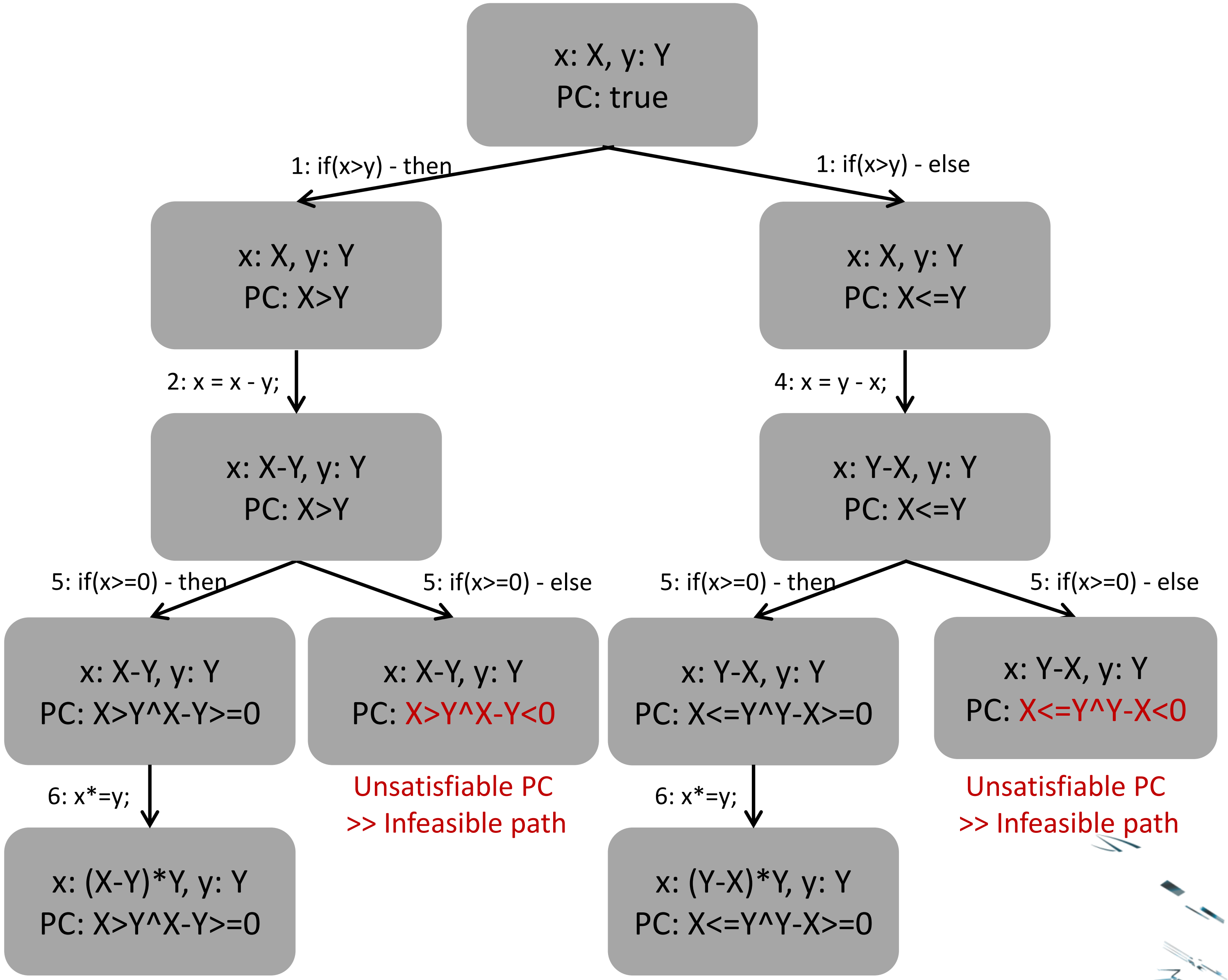
- Symbolic Execution Tree (SET) is created during symbolic execution
  - nodes of a SET represent the symbolic program states and edges represent transitions between these states
  - symbolic state consists of symbolic variables, a program location and a path constraint (PC) which is the conjunction 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

Constraints and SMT Solvers

- Path constraint (PC)
  - e.g.: $X > Y \wedge Y+X \leq 8$
  - 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)
   {
1:   if (x > y)
2:     x = x - y;
3:   else
4:     x = y - x;
5:   if (x >=0)
6:     x *= y;
7:   else
8:     x /= y;
9:   return x;
   }
```

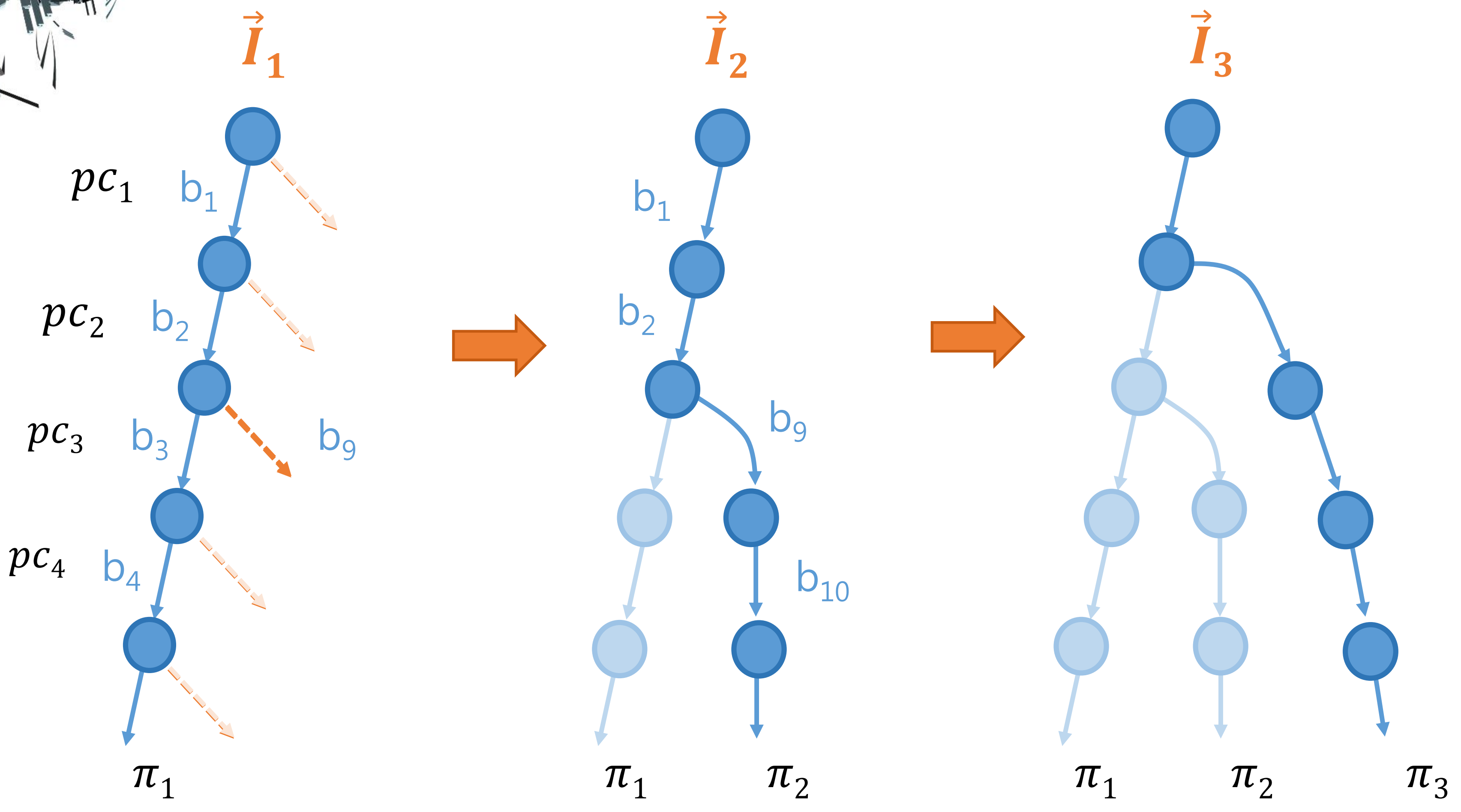
"Dead Code" (lines 7–8)



Limitations of Static SE

- Inability to solve very complex and non-linear constraints
  - X % 9 > 3 Λ Y > 15
  - (X >> 4) & 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



$PC = pc_1 \wedge pc_2 \wedge pc_3 \wedge pc_4 \wedge \ldots$ $PC' = pc_1 \wedge pc_2 \wedge \neg pc_3 \Longrightarrow \vec{I}_2$

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

CONCOLIC

=

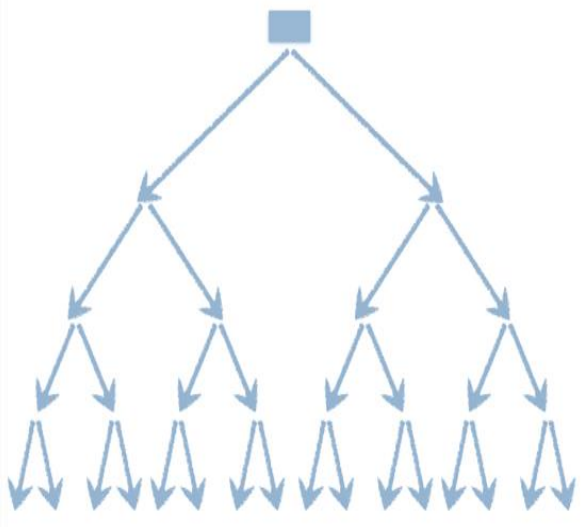
CONCRETE

+

SYMBOLIC

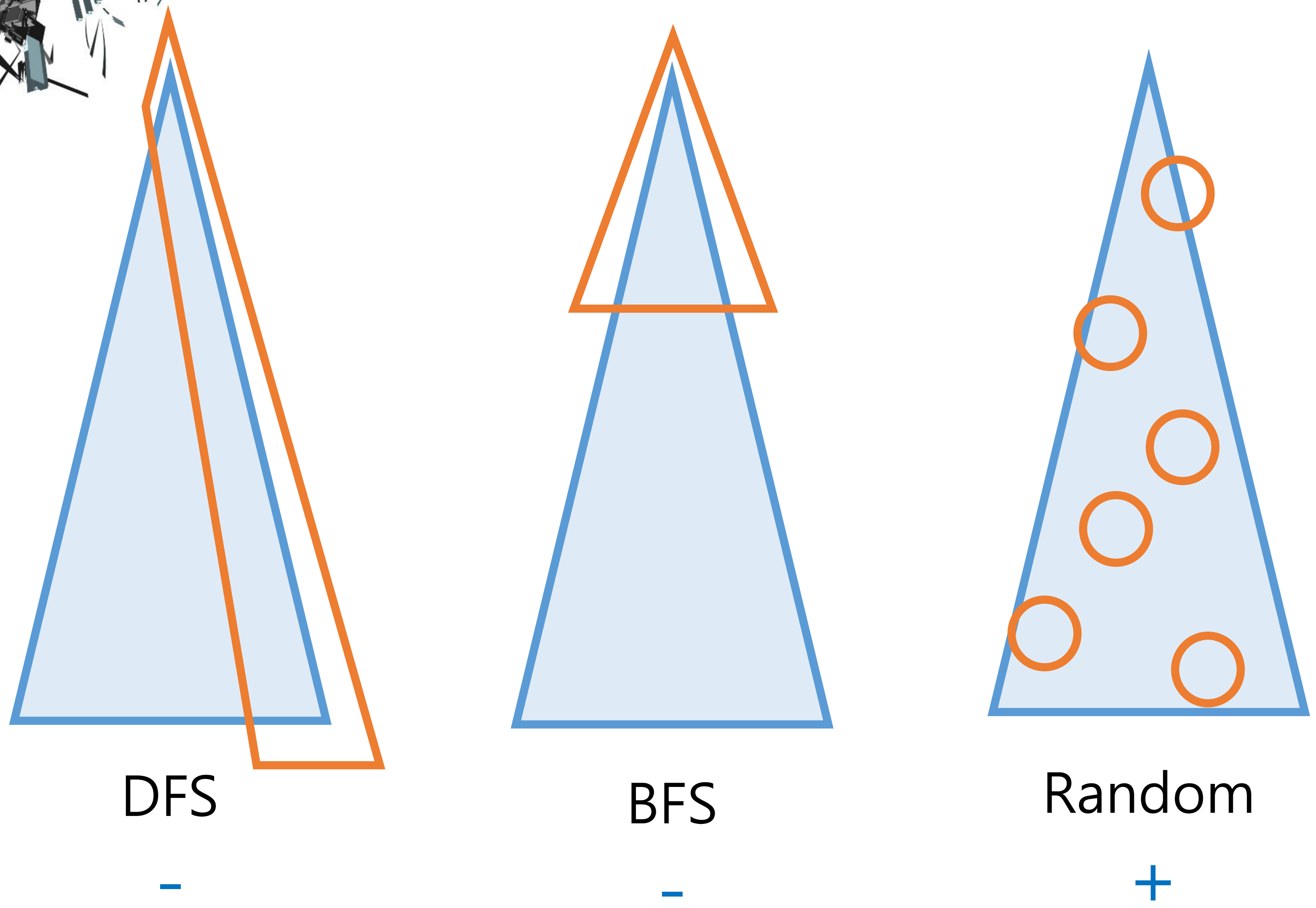
Path explosion challenge

- 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



```
3b2ff328  mov edx,0x3fd
3b2ff32d  in al,dx
3b2ff32e  test al,0x20
3b2ff330  je 0x3b2ff328
```

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

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

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

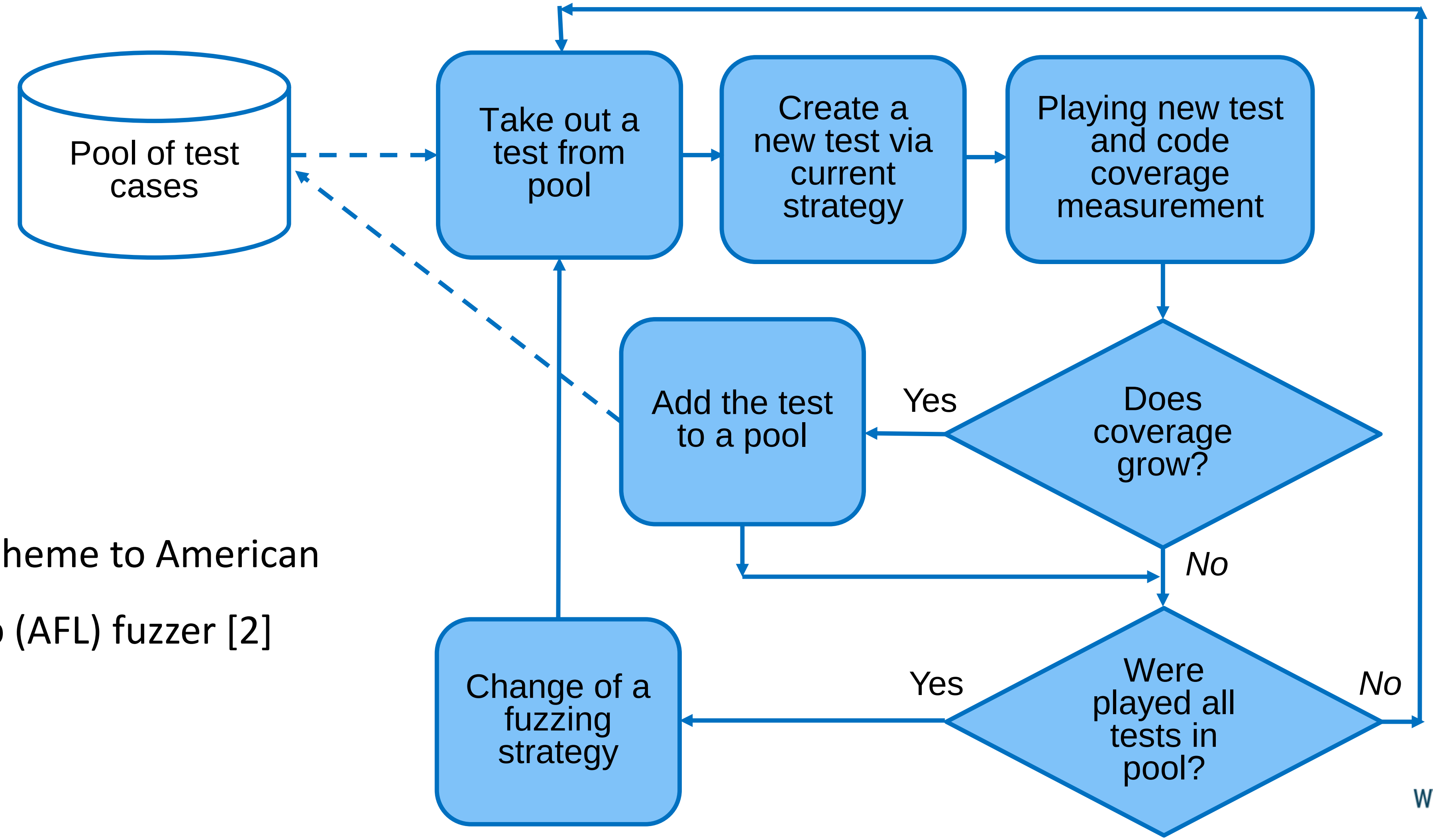
Unlikely a fuzzer would generate the constant 0x12345. In contrast, symbolic execution creates a test for covering code inside.

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 of tests generated by symbolic execution is a better way!

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

White-box fuzzing

```
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!*

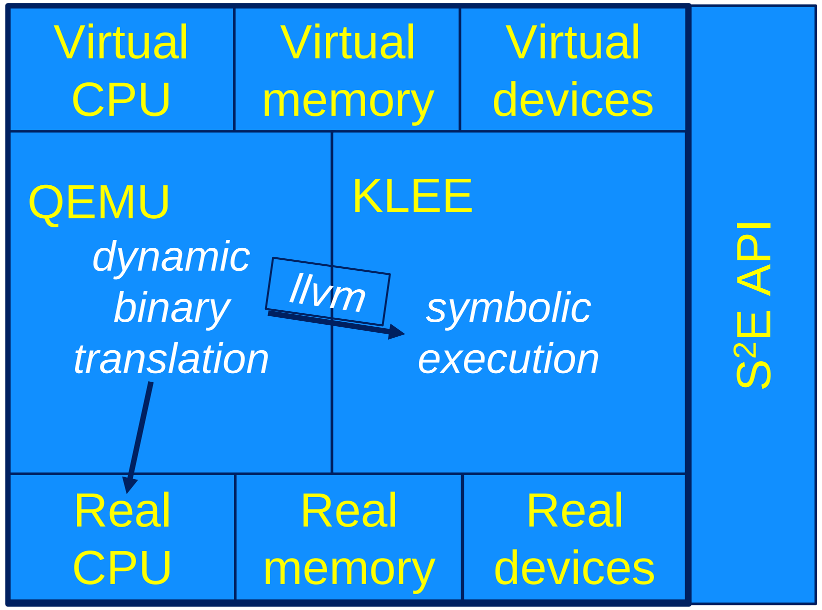
Symbolic Execution Engines

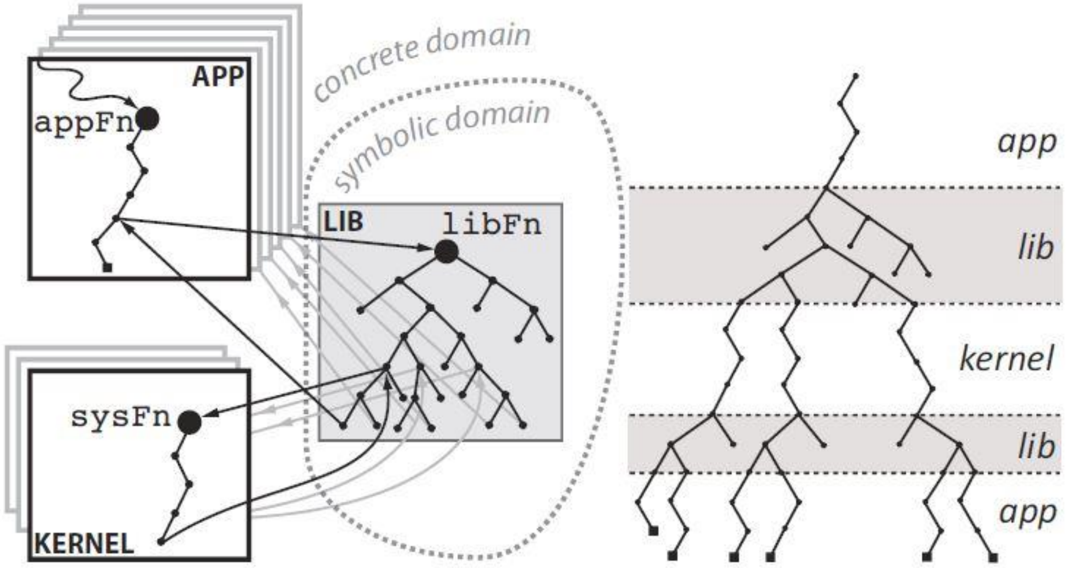
Open-source Symbolic Execution tools

Tool	Architecture / Language	URL
jCUTE	Java	https://github.com/osl/jcute
Otter	C	https://bitbucket.org/khooyp/otter/overview
KLEE	llvm	http://klee.github.io/
S2E	binary x86, x86-64, ARM	http://s2e.epfl.ch
Triton	binary x86, x86-64	http://triton.quarkslab.com
angr	libVEX based	http://angr.io/

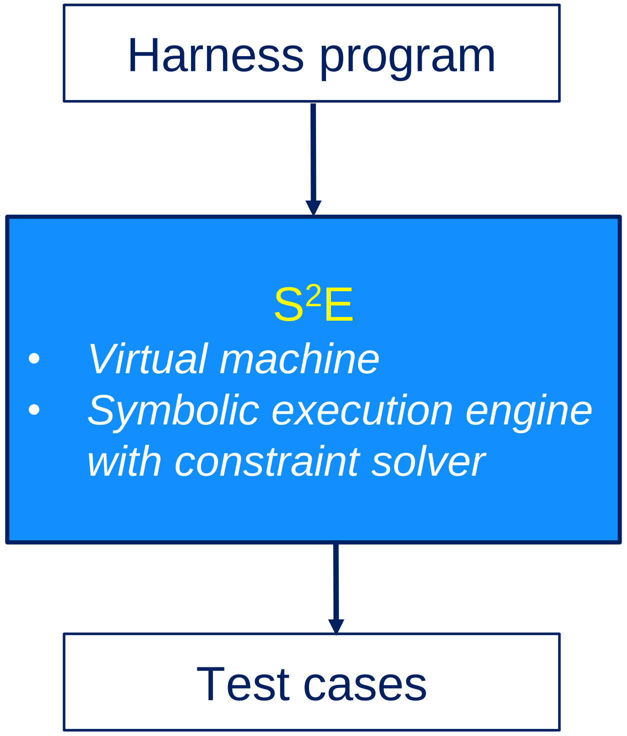
S^2E – Selective Symbolic Execution

- S^2E is based on KLEE symbolic execution engine and QEMU virtual machine [4]
- Flexible architecture with plug-ins

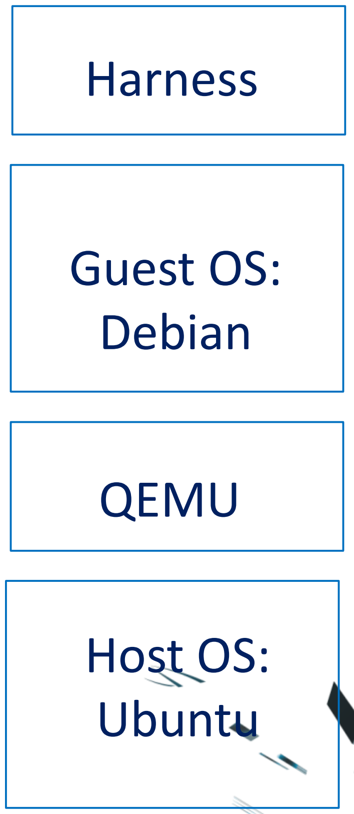




Test harness for S^2E

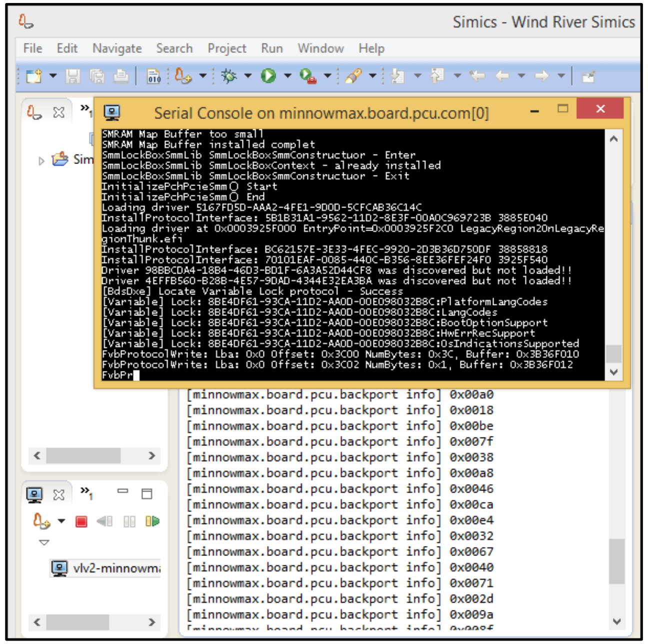


1. Mapping dump of SMRAM to harness memory space by mmap
2. Making symbolic of input parameters of a SMM handler: CommBuffer and size of CommBuffer
3. Set RSP value of stack pointer in SMM handler captured in boot procedure
4. Invocation by pointer of SMM handler from mapped SMRAM
5. Return back RSP 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

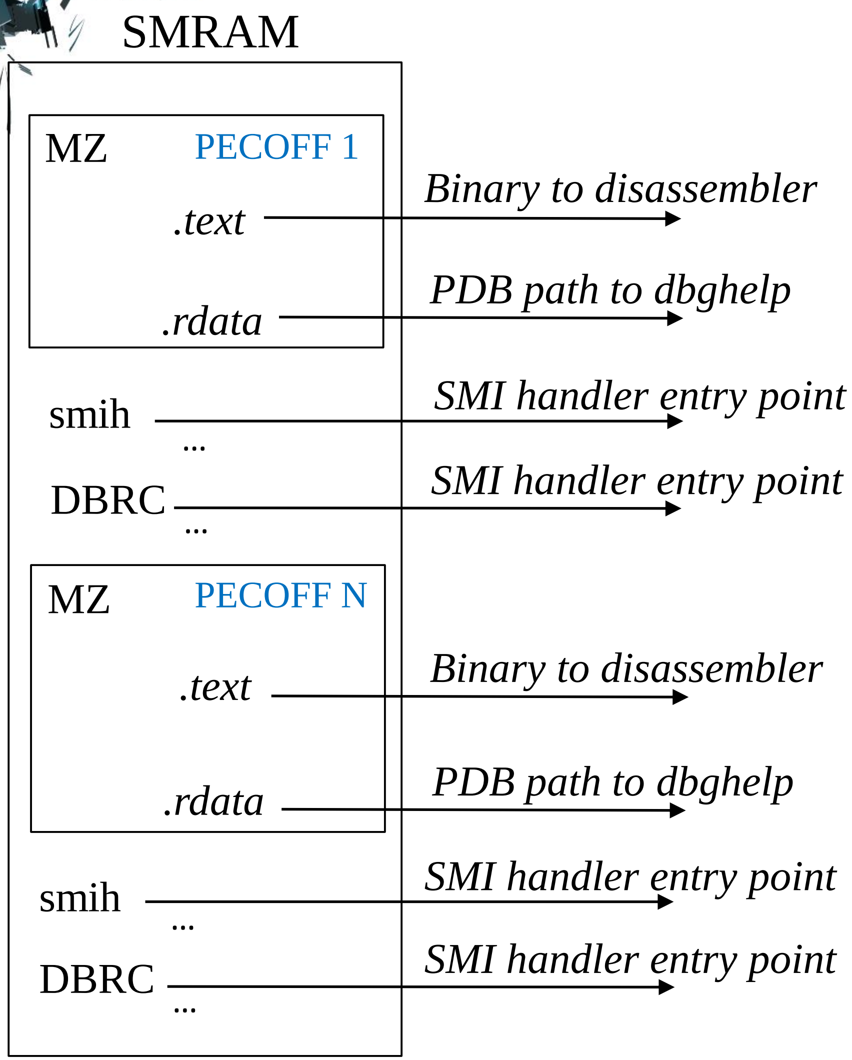
Dump SMRAM

- 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

```
SMM IPL opened SMRAM window
SMM IPL found SMRAM window 3B001000 - 3B3FFFFF
SMM IPL loading SMM Core at SMRAM address 3B3F6000
SMM IPL calling SMM Core at SMRAM address 3B3F62C0
mMaximumSupportAddress = 0xFFFFFFFFF
InstallProtocolInterface: F4CCBFB7-F6E0-47FD-9DD4-10A8F150C191 39CB9440
...
SmmLockBox SmmLockBoxHandler Exit
SmmLockBoxDxeLib SetLockBoxAttributes - Exit (Success)
SMM IPL locked SMRAM window
```

Fragment of boot log for open-source MinnowMax BIOS

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

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

```
73 6D 69 68 00 00 00 00  38 BB 31 3B 00 00 00 00  smih    8»1;
38 BB 31 3B 00 00 00 00  54 D5 31 3B 00 00 00 00  8»1;    TÕ1;
10 BB 31 3B 00 00 00 00  00 00 00 00 00 00 00 00  ►»1;
```

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

```
73 6D 69 65 00 00 00 00  90 E5 3F 3B 00 00 00 00  smie    ?å?;
98 2C 36 3B 00 00 00 00  D6 A3 68 65 5F 01 4A 4B  ,6;     Ö£he_☺JK
9C 89 1D 14 63 14 13 0A  D8 BB 31 3B 00 00 00 00  œ‰↔¶c¶‼◙Ø»1;
D8 BB 31 3B 00 00 00 00  00 00 00 00 00 00 00 00  Ø»1;
```

EFI scripts for IDA Pro [5] contains a broad collection of known GUIDs.

Playing and tracing test cases

```
set breakpoint on SMRAM_base_addr

script branch {
    while (TRUE) {
      wait for breakpoint
      stop execution
    }
}

forall test cases {
    %rsp = rsp_captured_in_boot
    &%rsp = SMRAM_base_addr
    %rcx = DispatchHandle
    %rdx = &RegisterContext
    %r8 = &CommBuf
    %r9 = &CommBufSize
    %rip = handler_entry_point
    enable tracing
    run execution
    disable tracing
}
```

```
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

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

- 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

Traversal of a computing tree

Challenge in processing of indirect calls and jumps:

```
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:

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

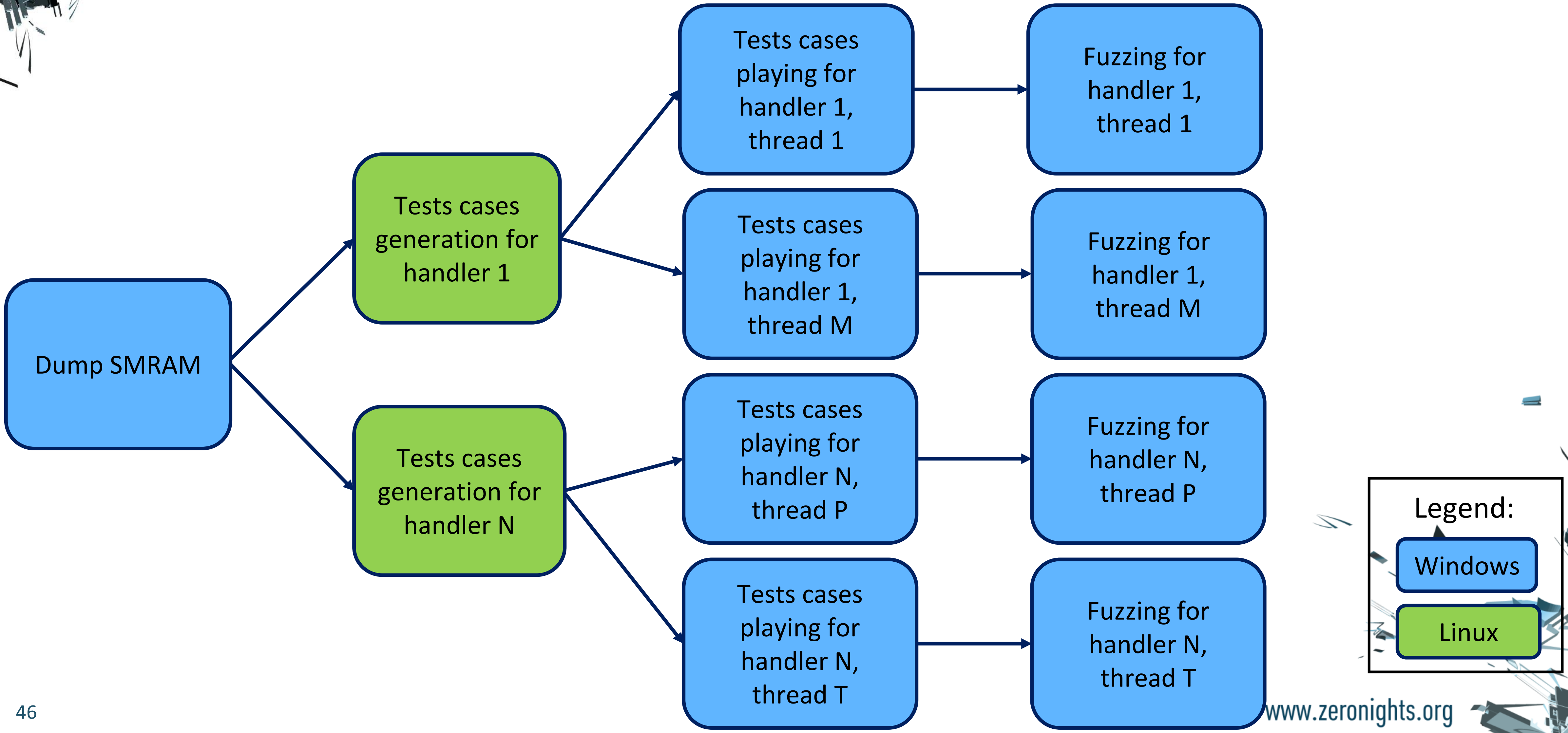
Pseudo-code of traversal of computing tree based on disassembled code

```
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*(2 + 3 + 5) = 100 hours =~ **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



Results

Code Coverage Outcomes

SMM Handler	Baseline[1]	Simple BlackBox Fuzzing[2]	Symbolic Execution	Symbolic Execution and Fuzzing
Handler 1	0 %	7 %	88 %	90 %
Handler 2	0 %	5 %	58 %	65 %
Handler 3	49 %	24 %	57 %	60 %
Handler 4	46 %	3 %	51 %	55 %
Handler 5	0 %	38 %	47 %	47 %

- [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);
- d-d-          if (_memcmp(list->node.path, path, len) == 0){
-                _memcpy(list->node.pwd, password, PWD_LEN);
-                return SUCCESS;
                 }
               }
+              dev = AllocatePool (D_LIST_SIZE);
+ d-d+         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

Automatically Detected Issues

SMM Handler	Simple BlackBox Fuzzing	Symbolic Execution	Symbolic Execution and Fuzzing
Handler 1	1	2	2
Handler 2	1	1	2
Handler 3	0	0	0
Handler 4	0	0	0
Handler 5	0	0	0

- 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.

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;
```

Instruction of the violation

Test file

Fragment of C-source that performed the access

C-file and line #

Unallowed address

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

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

Call stack trace with fragments of source code

Artificial example

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

1. Amal Khalil & Juergen Dingel, CISC836: Models in Software Development: Methods, Techniques, and Tools, 2015.
2. AFL: http://lcamtuf.coredump.cx/afl/
3. Code Coverage: http://www.bullseye.com/coverage.html
4. Vitaly Chipounov, Volodymyr Kuznetsov, George Candea, "S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems", 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011
5. EFI scripts for IDA Pro: https://github.com/snare/ida-efiutils
6. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A. and Bos, H., "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries", USENIX-2016
7. Xeno Kovah and Corey Kallenberg, How Many Million BIOSes Would you Like to Infect?", 2015
8. Extract SMI handlers: http://blog.cr4.sh/2016/10/exploiting-ami-aptio-firmware.html

Thank you!