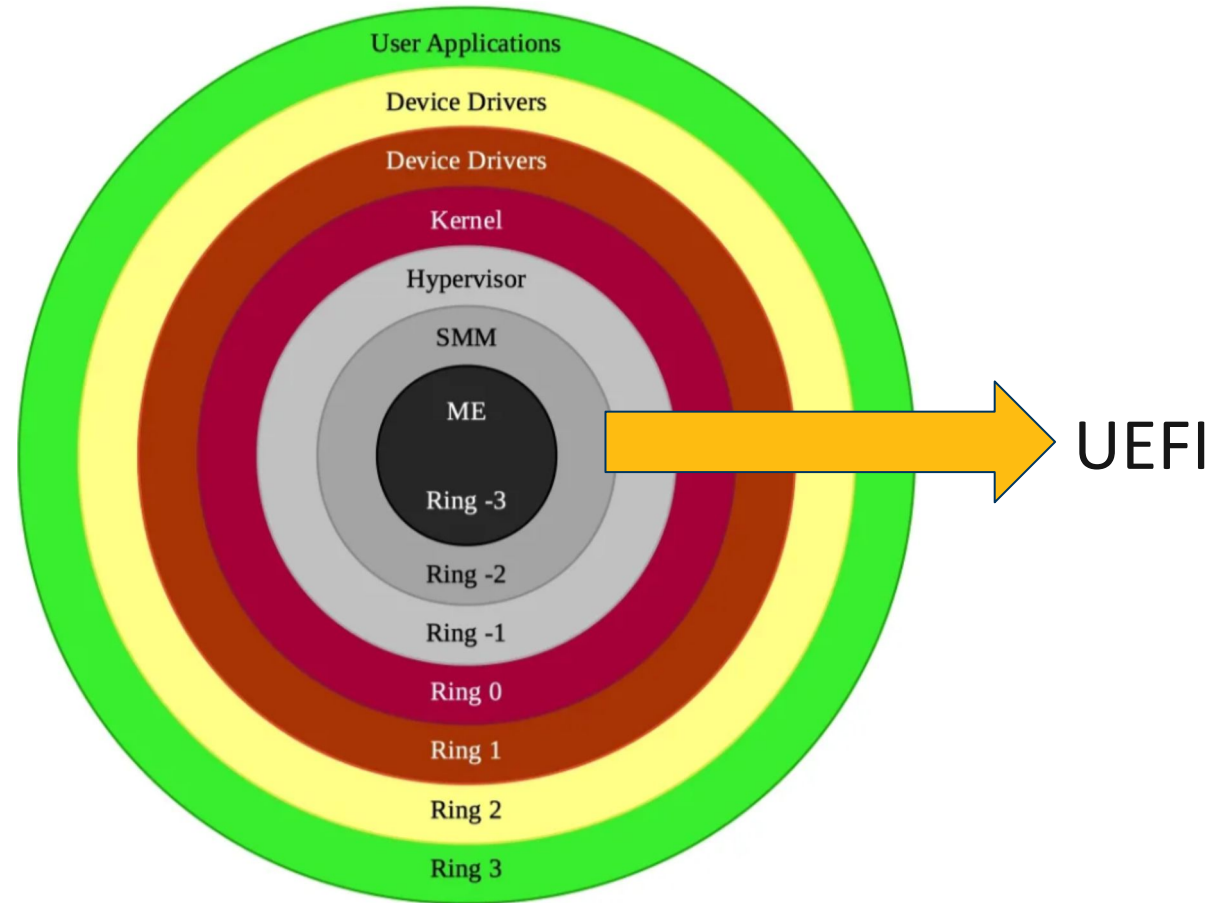


# STASE: Static Analysis Guided Symbolic Execution for UEFI Vulnerability Signature Generation

**Md Shafiuzzaman**, Achintya Desai, Laboni Sarker, Tevfik Bultan

# Securing UEFI: Guarding the Gatekeeper

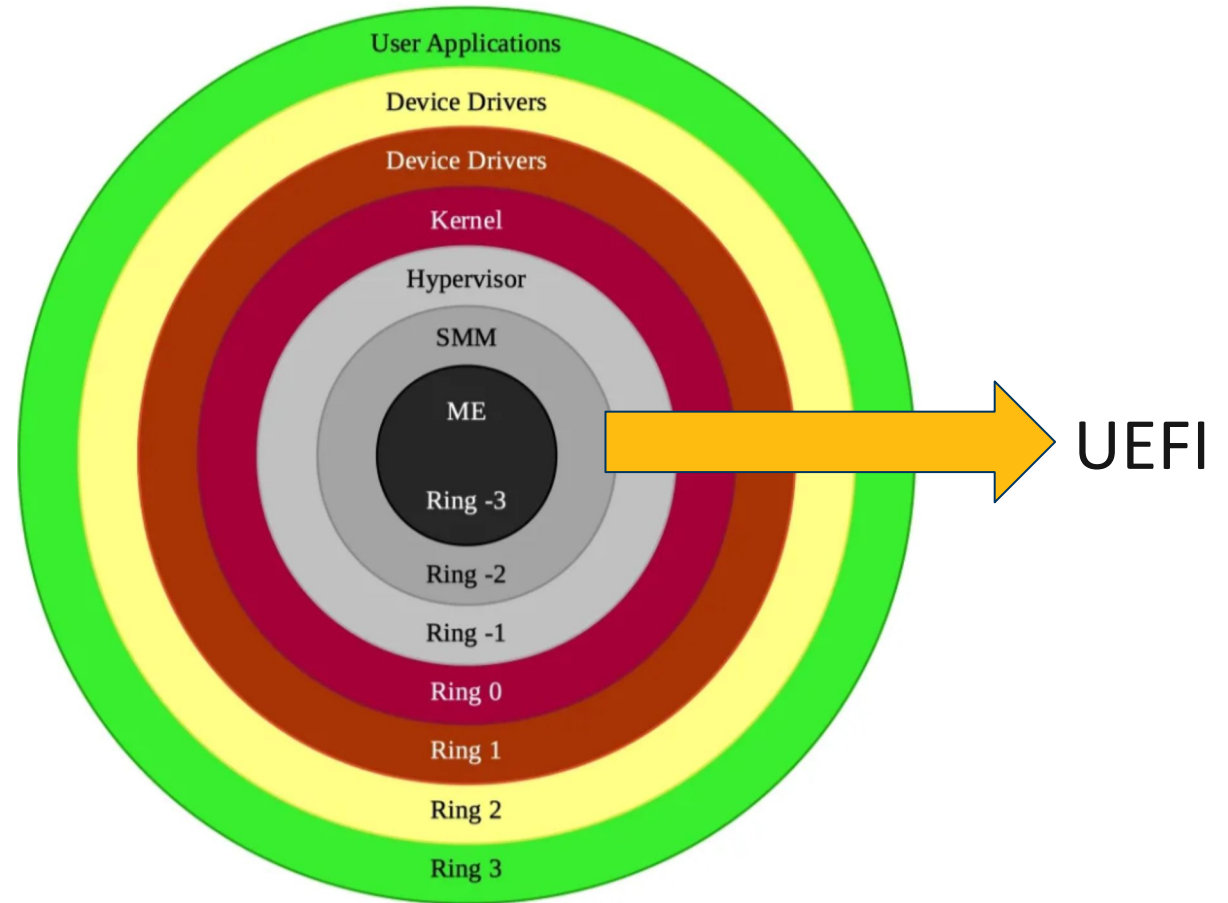
1. UEFI has higher privileged security access



Privilege Rings for Intel Architecture

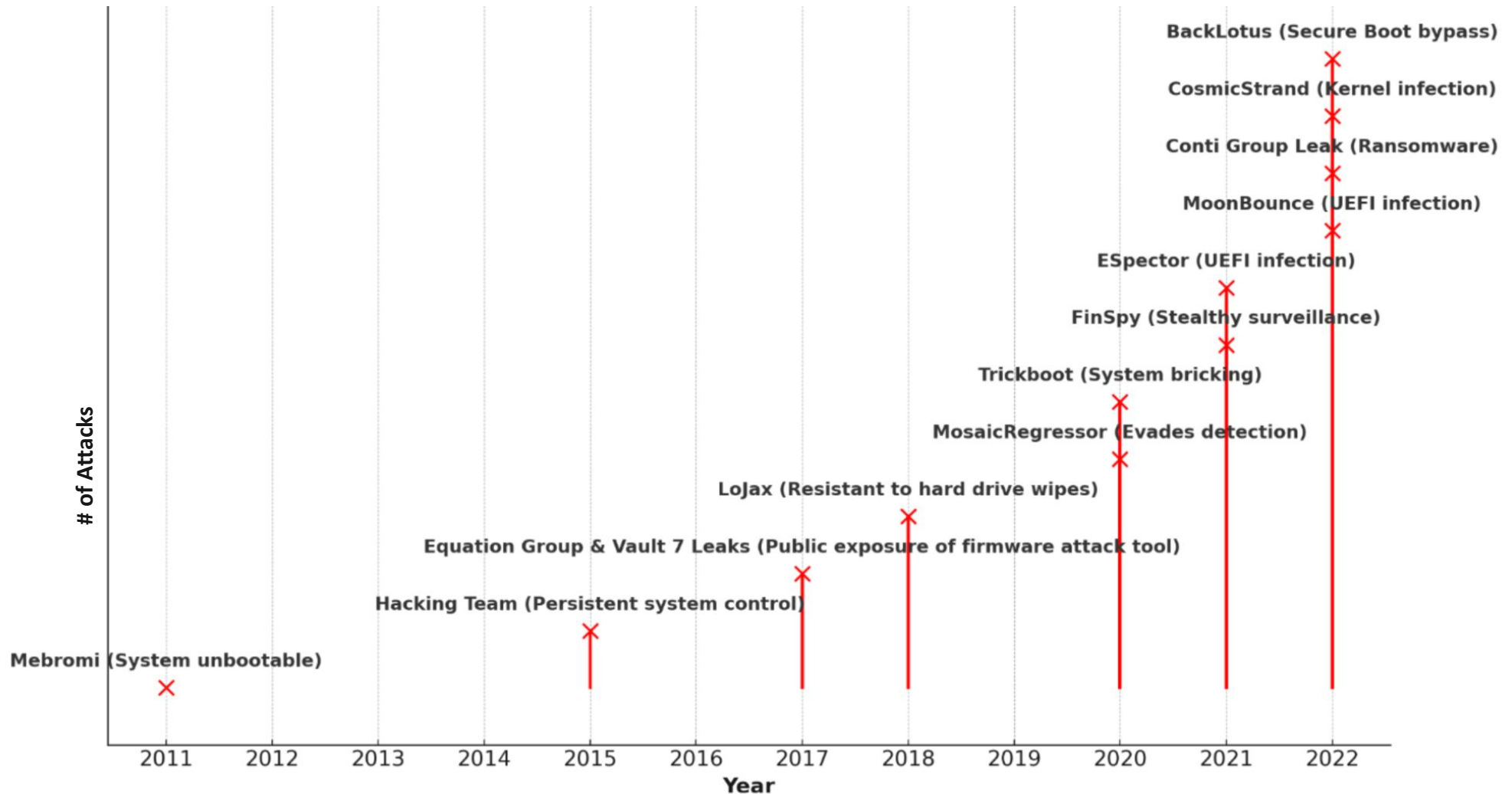
# Securing UEFI: Guarding the Gatekeeper

1. UEFI has higher privileged security access
2. UEFI exploits can persist through:
  - a. System reboot
  - b. OS reinstallation
  - c. Partial physical part replacement



Privilege Rings for Intel Architecture

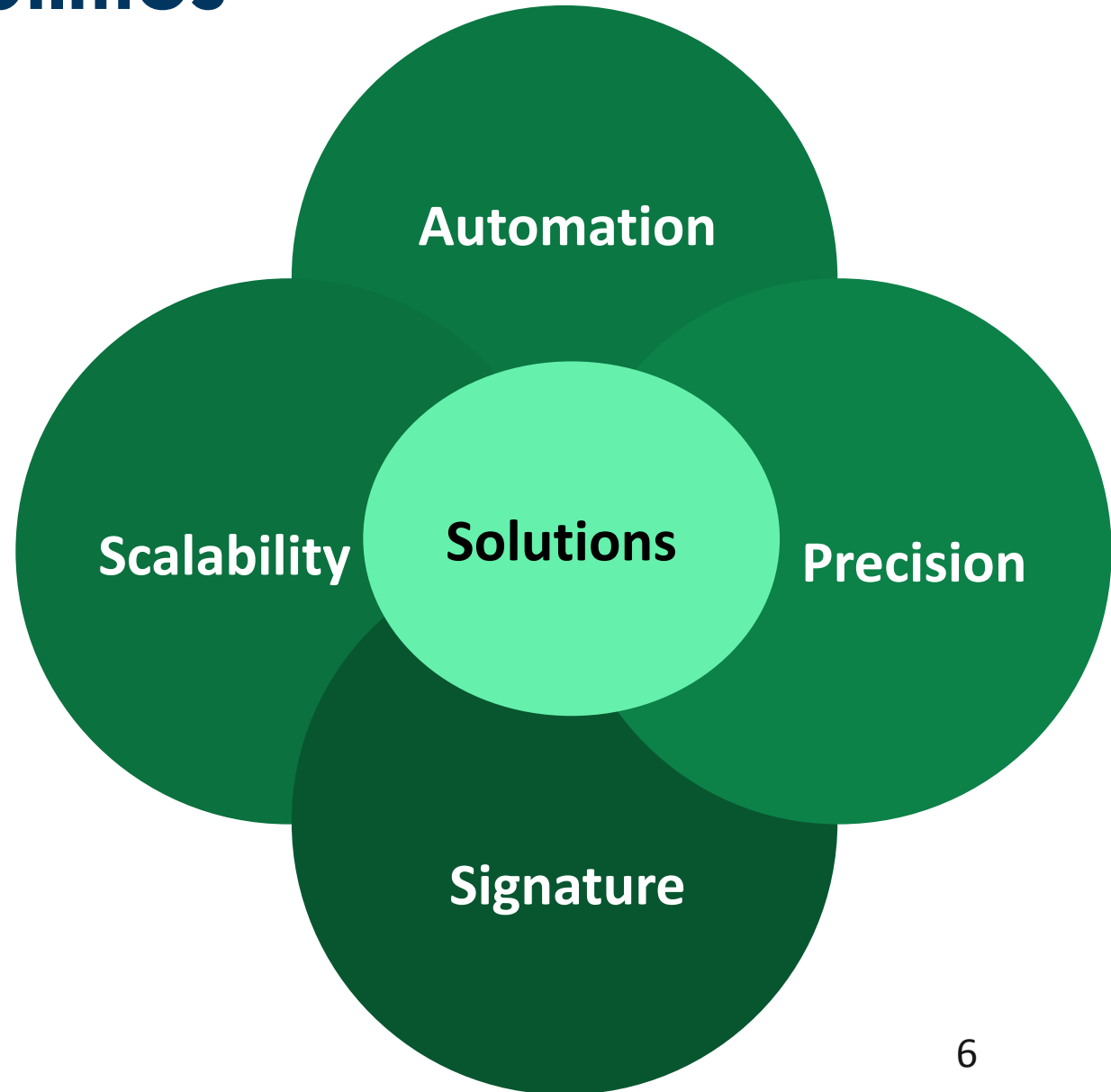
# Impact of Growing UEFI Threats



# Detecting UEFI Vulnerabilities



# Detecting UEFI Vulnerabilities



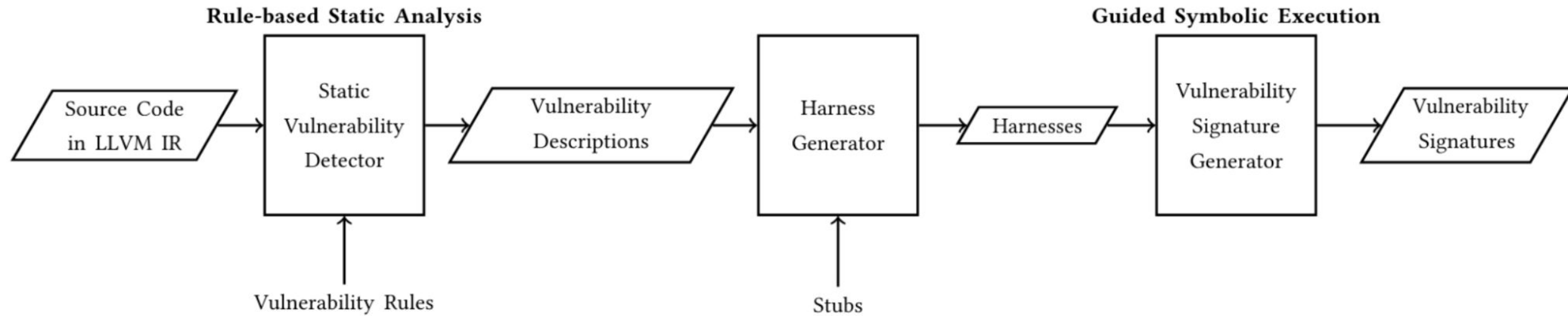
# Detecting UEFI Vulnerabilities

	Scalability	Precision	Automation	Vulnerability Signature
Static Analysis	Yes	Prone to FP	Yes	No
Fuzzing	Requires Guidance	Yes	Requires Harnesses	No (traditional fuzzing)
Symbolic Execution	Requires Guidance	Yes	Requires Harnesses	Yes

# Detecting UEFI Vulnerabilities

	Scalability	Precision	Automation	Vulnerability Signature
Static Analysis	Yes	Prone to FP	Yes	No
Fuzzing	Requires Guidance	Yes	Requires Harnesses	No (traditional fuzzing)
Symbolic Execution	Requires Guidance	Yes	Requires Harnesses	Yes
STASE	Yes	Yes	Yes	Yes

# STASE: Static Analysis Guided Symbolic Execution



# Rule-based Static Analysis

## Static Vulnerability Detector:



## Rule Example to capture out of bounds access :

```
index_tainted_out_of_bound_primitive (?func, ?op1, ?op2, ?op2,  
?instr, ?line) :-      argv_alloc(?valloc),  
                        ( subset.var_points_to(_,?valloc,_,?op2);  
                          ( subset.operand_points_to(_,?ialloc,_,?op2),  
                            subset.ptr_points_to(_,?valloc,_,?ialloc) ) ),  
                        instr_func(?instr, ?func),  
                        indexaccessinstructions(?op1, ?op2, ?instr),  
                        instr_pos(?instr, ?line, ?col).
```

# Code Example

## 1. DHCPv4 offer packet handler

```
EFI_STATUS PxeBcHandleDhcp4Offer  
(IN PXEBC_PRIVATE_DATA *Private)
```

## 2. Call sequence

```
PxeBcCopyDhcp4Ack() ->  
PxeBcParseDhcp4Packet() ->  
PxeBcParseVendorOptions()
```

## 4. Vulnerable code segment

```
#define  
SET_SINGLE_VENDOR_OPTION_BIT_MAP  
(x, y, z) ((* (x + (* (UINT32 *) (y)  
/ 32))) = (UINT32) ((UINT32)  
((x) [* (UINT32 *) (y) / 32]) | BIT  
((z) % 32)))
```

## 3. Vulnerable Function

```
VOID PxeBcParseVendorOptions ( IN  
EFI_DHCP4_PACKET_OPTION  
*Dhcp4Option, IN  
PXEBC_VENDOR_OPTION *VendorOption  
) { BOOLEAN Iwc = FALSE; UINT32  
*BitMap;  
EFI_DHCP4_PACKET_OPTION  
*PxeOption;  
  
    BitMap = VendorOption->BitMap;  
  
    PxeOption =  
(EFI_DHCP4_PACKET_OPTION  
*) &Dhcp4Option->Data[0];  
  
    if (!Iwc)  
        SET_SINGLE_VENDOR_OPTION_BIT_M  
AP (BitMap, Dhcp4Option->Data,  
PxeOption->OpCode); ...}
```

# Vulnerability Descriptions

$$V_{st_P} = \langle P, E, I, A, K, L, U \rangle$$

- Program ( $P$ ) and Line number ( $L$ ) in source code:
  - `PxeBcParseVendorOptions(), NetworkPkg/UefiPxeBcDxe/PxeBcDhcp4.c#L215`
  - `PxeBcParseVendorOptions(), NetworkPkg/UefiPxeBcDxe/PxeBcDhcp4.c#L232`
- Attacker EntryPoint ( $E$ ) and Variables to be made symbolic ( $I$ ):
  - `PxeBcHandleDhcp4Offer and %3`
  - `PxeBcHandleDhcp4Offer and %3`
- Assertions ( $A$ ):
  - `Assert(%276 >= 0 && %276 < Sizeof(%270))`
  - `Assert(%334 >= 0 && %334 < Sizeof(%327))`
- Target LLVM-level Instructions ( $K$ )
- Instructions to be sliced ( $U$ )

# Harness Generation: Path Exploration

$$V_{st_P} = \langle P, E, I, A, K, L, U \rangle$$

- $P \rightarrow$  Program to run symbolic execution ([PxeBcDhcp4.c](#))
- $E \rightarrow$  Symbolic execution entrypoint ([PxeBcHandleDhcp4Offer](#))
- $I \rightarrow$  Symbolic variables

Arguments of the entry point function: [\\*Private](#)

Global variables: [Status](#), [IsPxeOffer](#)

- $A \rightarrow$  Assertion template
- $L \rightarrow$  Assertion locations

[PxeBcParseVendorOptions\(\)](#), [NetworkPkg/UefiPxeBcDxe/PxeBcDhcp4.c#L215](#)

[PxeBcParseVendorOptions\(\)](#), [NetworkPkg/UefiPxeBcDxe/PxeBcDhcp4.c#L232](#)

- $P-U \rightarrow$  Sliced code

# Harness Generation: Environment Configuration

01	Stubs	<ul style="list-style-type: none"><li>● Global tables</li><li>● Hardware interactions</li><li>● Memory handling</li></ul>
02	Symbolic Values	<ul style="list-style-type: none"><li>● PCD Variables</li><li>● Firmware Parameters</li></ul>
03	Predefined Values	<ul style="list-style-type: none"><li>● Protocol GUIDs</li></ul>

# Guided Symbolic Execution

Instrument Code using Harnesses:



Symbolic Execution on Instrumented Code Segments:

```
KLEE: ERROR: ./dynamic_harness/./edk2/NetworkPkg/UefiPxeBcDxe/PxeBcDhcp4.c:232: ASSERTION FAIL: arrayIndex >= 0 && arrayIndex < bitmapSize
KLEE: NOTE: now ignoring this error at this location
^CKLEE: ctrl-c detected, requesting interpreter to halt.
KLEE: ERROR: ./dynamic_harness/./edk2/NetworkPkg/UefiPxeBcDxe/PxeBcDhcp4.c:232: Query timed out (fork).
KLEE: NOTE: now ignoring this error at this location
KLEE: halting execution, dumping remaining states

KLEE: done: total instructions = 19487
KLEE: done: completed paths = 14
KLEE: done: partially completed paths = 509
KLEE: done: generated tests = 12
```

# Vulnerability Signature

## Precondition:

```
array IsPxeOffer[1] : w32 → w8 = symbolic
array Private→DhcpAck.Dhcp4→OptList[8] : w32 → w8 =
symbolic
array Private→SelectIndex[4] : w32 → w8 = symbolic
array Status[8] : w32 → w8 = symbolic
(Eq 255 (ReadLSB w32 0 Private→SelectIndex))
  (Eq false
    (Slt (ReadLSB w64 0 Status)
      0))
  (Eq false
    (Eq 0 (Read w8 0 IsPxeOffer)))
  (Eq false
    (Eq 0
      N0:(ReadLSB w64 0
Private→DhcpAck.Dhcp4→OptList)))
```

## Simplified form:

```
Private→SelectIndex = 255 ^ Status ≥ 0 ^
IsPxeOffer ≠ 0 ^ Private→DhcpAck.Dhcp4→OptList ≠ 0
```

## Postcondition:

```
Error: ASSERTION FAIL: arrayIndex
≥ 0 && arrayIndex < bitmapSize
File:
./dynamic_harness/ ../demo2_edk2/Ne
tworkPkg/UefiPxeBcDxe/PxeBcDhcp4.c
Line: 232
assembly.ll line: 14927
State: 526
```

## Simplified form:

```
Out-of-bounds_access ASSERTION
FAIL
at the program location
PxeBcParseVendorOptions@PxeBcDhcp4
.c:232
```

# Experimental Setup and Benchmarks

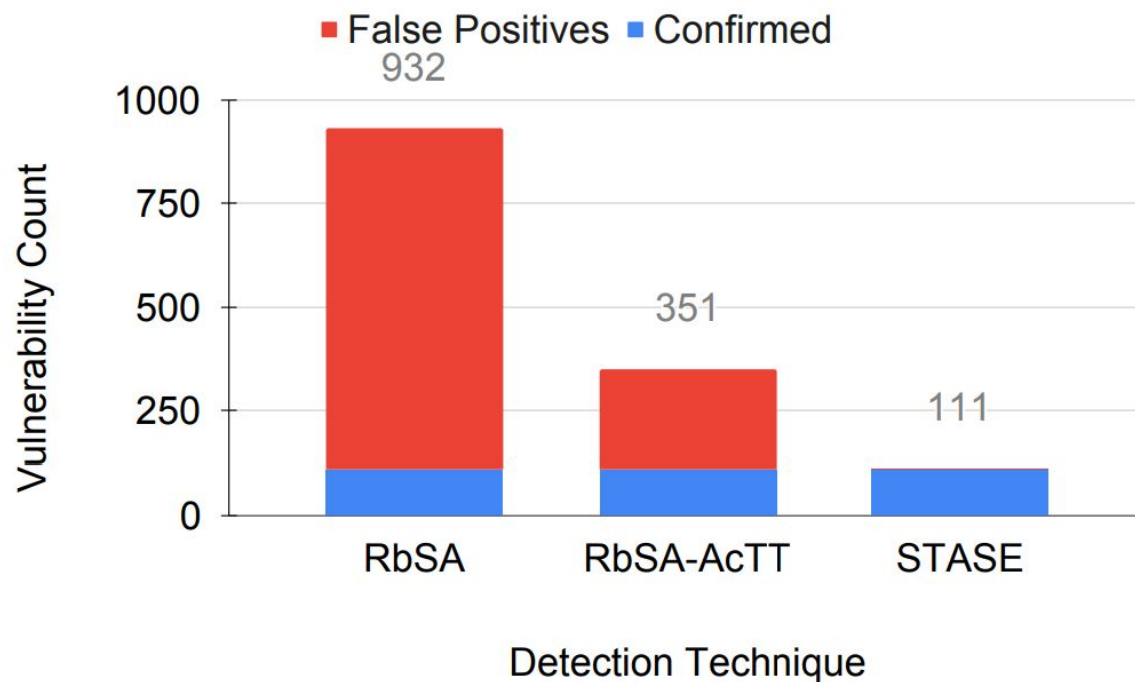
## Benchmark

- EDKII SMM Drivers
- EDKII Network Module
- DARPA HARDEN Challenge Sets
- Injected EDKII

## Techniques Compared

- **RbSA**: Rule-based Static Analysis
- **RbSA-AcTT**: RbSA + Attacker-controlled Taint Tracking
- **SE**: Symbolic Execution using KLEE
- **SE-ECH**: SE + Environment Configuration Harness
- **HBFA**: Intel's Host-Based Firmware Analyzer

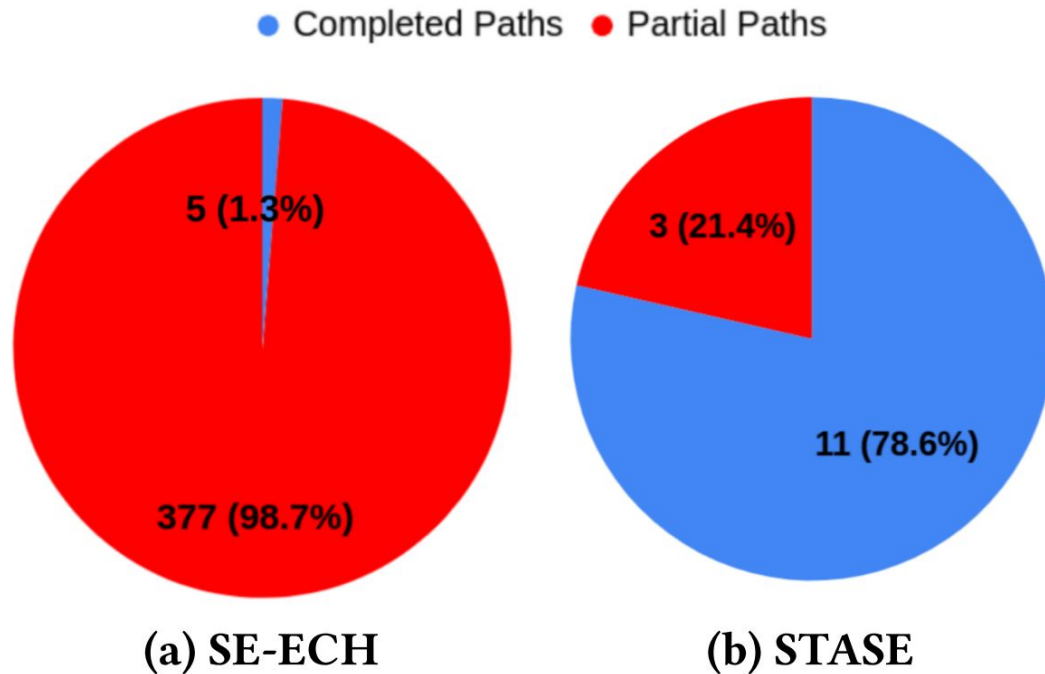
# False Positive Reduction



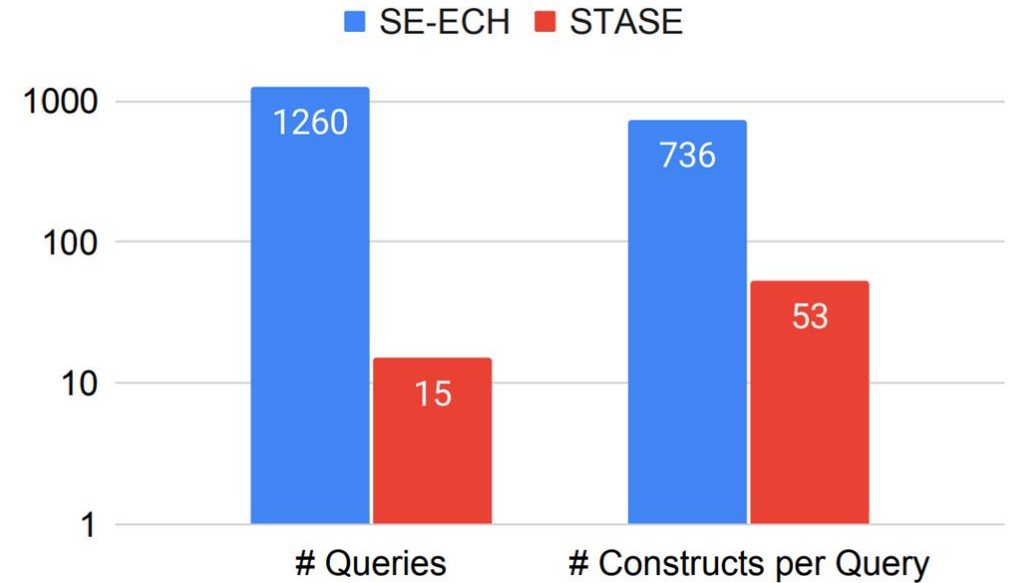
Vulnerability	RbSA		RbSA-AcTT		STASE	
	# Vul.	FP %	# Vul.	FP %	# Vul.	FP %
SMRAM READ	261	92.34%	115	82.61%	20	0.00%
SMRAM WRITE	501	88.02%	145	58.62%	60	0.00%
SMM Callout	2	0.00%	2	0.00%	2	0.00%
Buffer Overflow	55	70.91%	33	51.52%	16	0.00%
Integer Overflow	14	85.71%	9	77.78%	2	0.00%
Integer Underflow	44	84.09%	34	79.41%	7	0.00%
Out-of-bound Access	33	93.94%	10	80.00%	2	0.00%
Use After Free	2	50.00%	1	0.00%	1	0.00%
Division by Zero	20	95.00%	2	50.00%	1	0.00%

**STASE reduces false positive rate from 88% for RbSA and for 68% for RbSA-AcTT to 0%**

# Alleviating Scalability

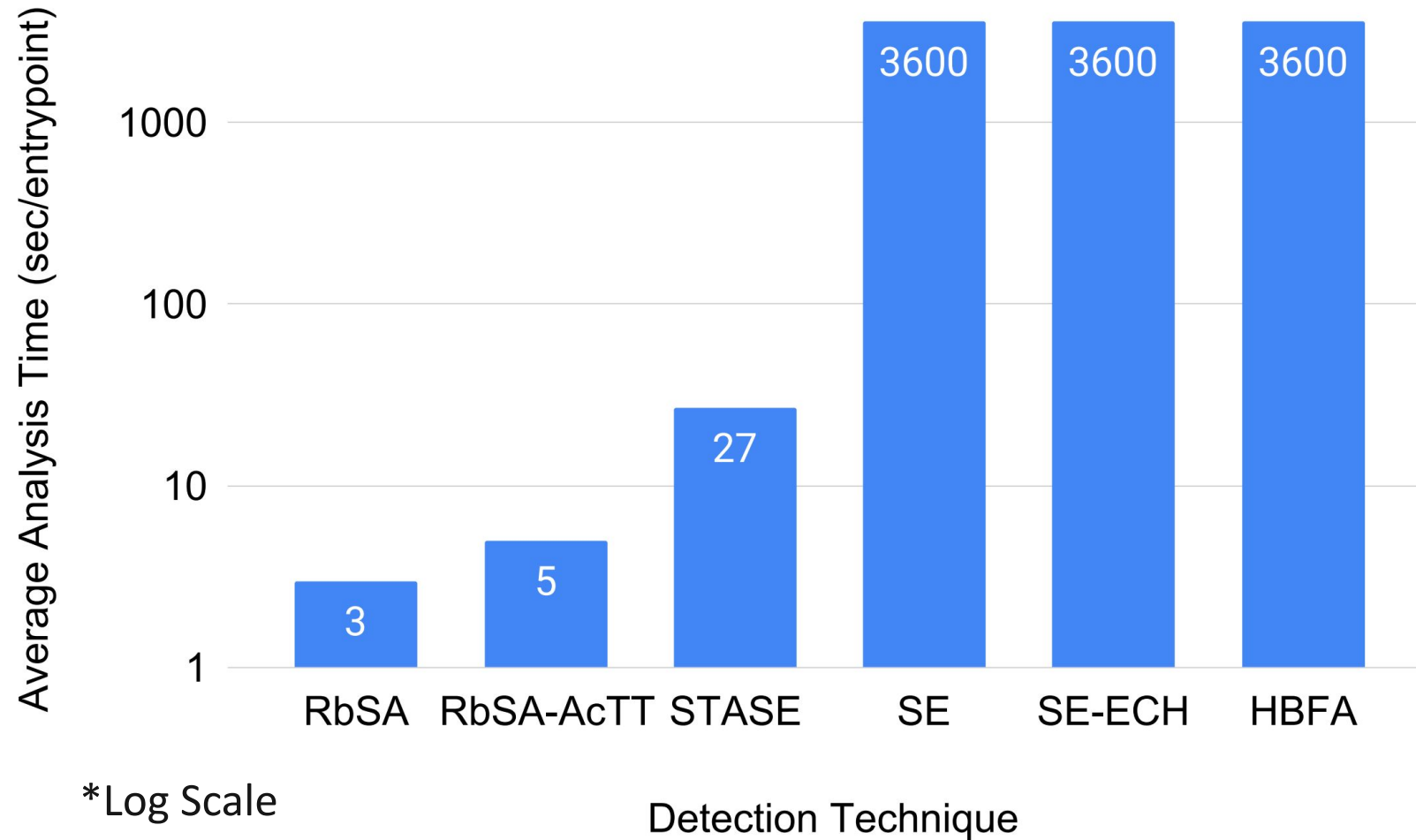


Paths Explored per Entrypoint



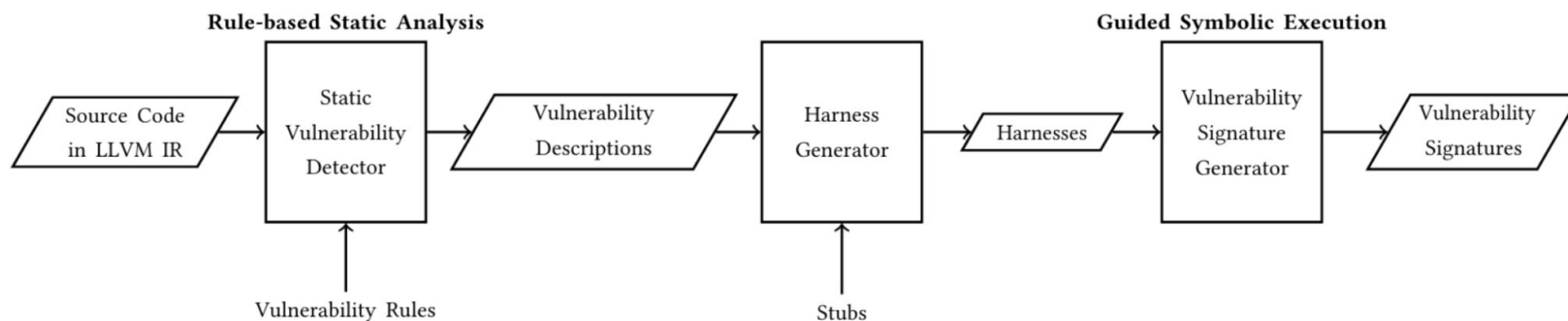
Constraint Solving Queries per Entrypoint

# Detection Time Reduction



# Summary

- **Rule-based static analysis** locates potential vulnerabilities with vulnerability descriptions
- **Vulnerability descriptions** generates harnesses to ensure scalability without manual intervention
- **Guided symbolic execution** eliminates false positives and generates vulnerability signatures
- **STASE** combines static analysis and symbolic execution to achieve best of both worlds



THANK YOU

Tool: <https://github.com/shafiuzzaman-md/stase>

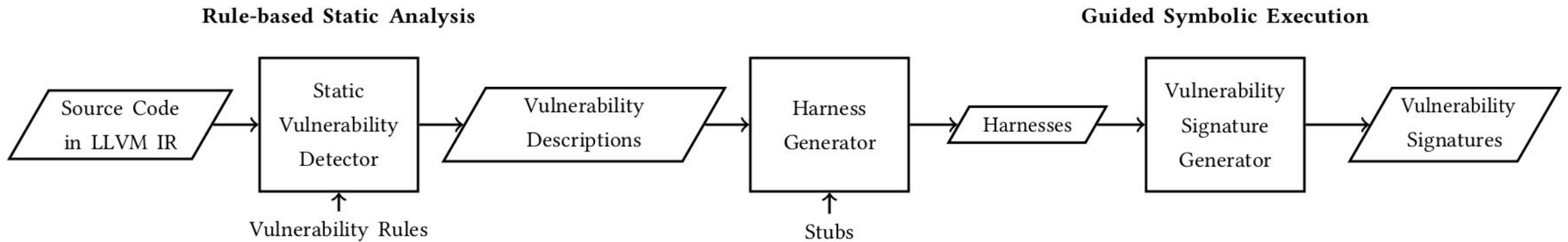
# Comparison of Vulnerability Detection and Average Analysis Time (seconds) Per Entrypoint (EP)

Dataset	#EP	SE		SE-ECH		HBFA		STASE	
		#Vul	Time	#Vul	Time	#Vul	Time	#Vul	Time
EDK II SMM	21	0	3600	0	3600	1	3600	13	24
EDK II Network	5	0	3600	0	3600	0	3600	5	47
HARDEN Demo1	27	0	3600	0	3600	1	3600	30	20
HARDEN Demo2	32	0	3600	2	3600	0	3600	18	28
Injected EDK II	21	0	3600	2	3600	1	3600	45	27
<b>Total</b>	106	0		4		3		111	

# EXTRA: Contributions

- (1) **Integration of rule-based static analysis with symbolic execution:** A novel approach for combining static analysis that reduces the false positives generated by static analysis and improves the scalability of symbolic execution.
- (2) **Rule-based static analysis for UEFI:** A rule-based and extensible static analysis approach to detect potential UEFI vulnerabilities.
- (3) **Automated harness generation for symbolic execution via rule-based static analysis:** An automated technique for generating symbolic execution harnesses to deploy symbolic execution without manual intervention.
- (4) **Scalable and precise vulnerability signature generation:** Guiding symbolic execution using static analysis effectively narrows the execution path, improving scalability and reducing analysis time, and outputs the vulnerability signatures that extract pre- and post-conditions of the vulnerabilities.

# EXTRA: Technique



**Figure 1: STASE Technique**

# EXTRA: Vulnerability Example

```
2165 SmramProfileHandlerGetInfo (
2166     IN SMRAM_PROFILE_PARAMETER_GET_PROFILE_INFO *
        SmramProfileParameterGetInfo) { ...
2181     SmramProfileParameterGetInfo->ProfileSize      =
        SmramProfileGetDataSize ();} ...
2290 /*@param CommBuffer A pointer to a collection of data in memory
        that will be conveyed from a non-SMM environment into an
        SMM environment.
2291 @param CommBufferSize The size of the CommBuffer.*/
2300 ...
2301 SmramProfileHandler ( IN EFI_HANDLE DispatchHandle, IN CONST
        VOID *Context OPTIONAL, IN OUT VOID *CommBuffer OPTIONAL,
        IN OUT UINTN *CommBufferSize OPTIONAL) { ...
2341     switch (SmramProfileParameterHeader->Command) {
2342     case SMRAM_PROFILE_COMMAND_GET_PROFILE_INFO: ...
2349         SmramProfileHandlerGetInfo ((
            SMRAM_PROFILE_PARAMETER_GET_PROFILE_INFO *) (UINTN)
            CommBuffer);
2350         break;
2351         ...}
```

Listing 1: SMRAM Write Vulnerability Example

Assume that we added an SMRAM Write vulnerability condition on Line of Code (LOC) 2181 and received an assertion failure.

**Precondition:** The precondition defines the necessary conditions that must be satisfied to navigate from the entry point of the SMM driver (LOC 2301) to the vulnerability location (LOC 2181), where an assertion violation occurs. The path to this vulnerability is characterized by a series of constraints on the system state. An example path constraint is `CommBuffer->Header.Command = 1` and `*CommBufferSize == 24` and `mSmramReadyToLock == 0`

**Code Segment:** This code segment is accessing and manipulating the `CommBuffer` that starts from the entry point at LOC 2301 in `SmramProfileRecord.c` and proceeds to the assertion check at LOC 2181.

**Postcondition:** The postcondition describes the system state when the assertion fails at LOC 2181, indicating a failure to maintain the integrity constraints regarding SMRAM boundaries.

**SMRAM Write:**  $\neg(*CommBufferSize \leq SMRAM\_BASE + SMRAM\_SIZE \wedge CommBuffer \leq SMRAM\_BASE + SMRAM\_SIZE \wedge (*CommBufferSize = 0 \vee CommBuffer + *CommBufferSize \leq SMRAM\_BASE + SMRAM\_SIZE))$

**Vulnerability Signature:** Putting it all together, the vulnerability signature for this vulnerability can be expressed as a Hoare triple:

*{Precondition} CodeSegment {Postcondition}*

# EXTRA: Vulnerability Rules

- SMRAM Read vulnerability (based on CVE-2022-35896)
- SMRAM Write vulnerability (based on CVE-2022-23930)
- SMM Callout (based on CVE-2022-36338)
- Integer Underflow vulnerability (based on PixieFail CVE-2023-45229)
- Integer Overflow vulnerability
- Division by zero vulnerability
- Buffer Overflow vulnerability
- Out-of-bounds access vulnerability
- Use after Free vulnerability

# EXTRA: Slicing

*3.2.3 Program Slicing.* To run symbolic execution for each potential vulnerability, we want to identify the program locations that can be safely stubbed out. Since symbolic execution faces path explosion, it is crucial that we identify and remove program behaviors that are not related to the vulnerability target. We use program slicing [46] to achieve this. The slicing criteria are determined by the potentially vulnerable instruction and the taint-sink. We implemented the two-pass program slicing algorithm from [31], which is based on the dependence graph. We construct the dependence graph using control dependencies from [26] and data dependencies by tracking def-use chains and memory read-writes using pointer analysis from cclyzerpp. Note that our slicing implementation currently does not support mutual recursion, which can be added using the approach discussed in [15].

# EXTRA: Vulnerability Description Example

- $P$  is the program in which the potential vulnerability exists
- $E$  is the attacker-controlled entry point, which is defined as  $E = S_P^e \cdot \Sigma_1 \cdot f$
- $I$  is the attacker-controlled inputs, which are defined as  $I = \{v_1, \dots, v_m \mid \forall i \in [m], v_i = S_P^e \cdot \Sigma_1 \cdot \Delta(v_i)\}$
- $A$  is the assertion template corresponding the vulnerability category, which is defined as  $A = \text{assert}(B(l_1, \dots, l_n))$  such that  $\forall i \in [n], l_i = S_P^t \cdot \Sigma_1 \cdot \Delta(v_i)$  where  $B(l_1, \dots, l_n)$  are conditions over LLVM variables  $(l_1, \dots, l_n)$
- $K$  is the vulnerable LLVM instruction, which is defined as  $K = S_P^t \cdot \Sigma_1 \cdot r$
- $L$  is the vulnerability location in the source code (LOC: line of code), which is defined as  $L = S_P^t \cdot \Sigma_1 \cdot r \cdot \text{LOC}$
- $U$  is the list of source code locations that can be safely stubbed out, which is defined as  $U = \{S_P^i \cdot \Sigma_1 \cdot r \cdot \text{LOC} \mid S_P^i \in (S_P^e, \dots, S_P^t) \wedge S_P^i \notin \text{Slice}(S_P^e, \dots, S_P^t)\}$

Based on the code snippet in listing 3, the vulnerability description generated for the target is as follows:  $\langle P, E, I, A, K, L, U \rangle$

$P = \text{injected\_Tcg2Smm.c}, E = \text{TpmNvsCommunciate},$   
 $K = < \text{injected\_Tcg2Smm.bc} >: \text{TpmNvsCommunciate} : 32,$   
 $A = \text{assert}(\text{TempCommBufferSize} \neq 0), I = \text{CommBufferSize},$   
 $L = \text{injected\_Tcg2Smm.c} : 70, U = \{\text{injected\_Tcg2Smm.c} : 60\}$

```
47  EFI_STATUS EFIAPI TpmNvsCommunciate (IN EFI_HANDLE
    DispatchHandle, IN CONST VOID *RegisterContext, IN OUT
    VOID *CommBuffer, IN OUT UINTN *CommBufferSize){
59  ...
60  DEBUG ((DEBUG_VERBOSE, "%a()\n", __func__));
61  ...
69  TempCommBufferSize = *CommBufferSize;
70  mMcSoftwareSmi = mMcSoftwareSmi/TempCommBufferSize;
71  ...
72  }
```

Listing 3: Division by Zero Vulnerability Example

# EXTRA: Vulnerability Signature Example

```
1)Precondition:-
(Private->SelectIndex > 0 and Private -> SelectIndex - 1 <
  PXEBC_OFFER_MAX_NUM and Status = 16)
or (Private -> SelectIndex > 0 and Private -> SelectIndex - 1 <
  PXEBC_OFFER_MAX_NUM and Status = 0 and Private ->
  IsProxyRecved != 0 and Private -> IsOfferSorted != 0 and
  Private -> SelectProxyType < PxeOfferTypeMax and Private ->
  DhcpAck . Dhcp4 -> OptList != 0)
or (Private->SelectIndex > 0 and Private -> SelectIndex - 1 <
  PXEBC_OFFER_MAX_NUM and Status = 14 and Private ->
  IsProxyRecved = 0 and Private -> DhcpAck . Dhcp4 -> OptList
  != 0)
2)Code Segment:-
  Entrypoint: PxeBcHandleDhcp4Offer@PxeBcDhcp4.c:1013
  Symbolic Argument: *Private
  Assertion Location: PxeBcParseVendorOptions@PxeBcDhcp4.c:232
3)Postcondition:-
  !(arrayIndex >= 0 and arrayIndex < bitmapSize) at the program
  location PxeBcParseVendorOptions@PxeBcDhcp4.c:232
```

**Listing 8: Vulnerability Signature**