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

Md Shafiuzzaman†
University of California, Santa Barbara
Santa Barbara, CA, USA
mdshafiuzzaman@ucsb.edu

Achintya Desai †
University of California, Santa Barbara
Santa Barbara, CA, USA
achintya@ucsb.edu

Laboni Sarker
University of California, Santa Barbara
Santa Barbara, CA, USA
labonisarker@ucsb.edu

Tevfik Bultan
University of California, Santa Barbara
Santa Barbara, CA, USA
bultan@ucsb.edu

ABSTRACT

Since its major release in 2006, the Unified Extensible Firmware Interface (UEFI) has become the industry standard for interfacing a computer's hardware and operating system, replacing BIOS. UEFI has higher privileged security access to system resources than any other software component, including the system kernel. Hence, identifying and characterizing vulnerabilities in UEFI is extremely important for computer security. However, automated detection and characterization of UEFI vulnerabilities is a challenging problem. Static vulnerability analysis techniques are scalable but lack precision (reporting many false positives), whereas symbolic analysis techniques are precise but are hampered by scalability issues due to path explosion and the cost of constraint solving. In this paper, we introduce a technique called STatic Analysis guided Symbolic Execution (STASE), which integrates both analysis approaches to leverage their strengths and minimize their weaknesses. We begin with a rule-based static vulnerability analysis on LLVM bitcode to identify potential vulnerability targets for symbolic execution. We then focus symbolic execution on each target to achieve precise vulnerability detection and signature generation. STASE relies on the manual specification of reusable vulnerability rules and attacker-controlled inputs. However, it automates the generation of harnesses that guide the symbolic execution process, addressing the usability and scalability of symbolic execution, which typically requires manual harness generation to reduce the state space. We implemented and applied STASE to the implementations of UEFI code base. STASE detects and generates vulnerability signatures for 5 out of 9 recently reported PixieFail vulnerabilities and 13 new vulnerabilities in Tianocore's EDKII codebase.

KEYWORDS

Static Analysis, Symbolic Execution, Vulnerability Detection, Vulnerability Signatures, Firmware

ACM Reference Format:
Md Shafiuzzaman, Achintya Desai, Laboni Sarker, and Tevfik Bultan. 2024. STASE: Static Analysis Guided Symbolic Execution for UEFI Vulnerability Signature Generation. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695543

1 INTRODUCTION

UEFI has become an integral component in modern computer systems, replacing the traditional BIOS to provide enhanced support and stability. From 2013 to 2021, over 2.3 billion personal computer systems were shipped with UEFI, showcasing its widespread adoption [43]. Unfortunately, the frequency of firmware attacks targeting UEFI has also increased significantly [36], and there have been many reported examples of such attacks in recent years that affected millions of users [1–9, 11, 12, 35]. These attacks leverage vulnerabilities in UEFI to escalate privileges or gain unauthorized access. Vulnerability exploits in UEFI offer attackers a critical advantage since UEFI operates before the system boots, making exploits resilient to system reboots and operating system cleanups. Consequently, a compromised system remains vulnerable even if the operating system or hard disk is replaced. This inherent resistance to traditional mitigation techniques, combined with UEFI's widespread use, makes it an attractive target for attackers.

Detecting and characterizing vulnerabilities in UEFI is crucial for timely remediation and preventing attackers from generating exploits. However, this task is challenging for several reasons. First, UEFI exploits often arise from silent corruptions that do not crash the system but escalate privileges to perform unauthorized actions [49]. Additionally, the behavior causing the vulnerability may not manifest under normal operation but only under specific or unusual conditions [50]. Furthermore, attackers may exploit a combination of minor weaknesses to compromise the entire system, making detection and characterization even more difficult.

*This material is based on research supported by DARPA under the agreement number N66001-22-2-4037, by NSF under grants CCF-1901136, and CCF-2008660, and by an Amazon Research Award. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.
†Both authors contributed equally to this research.

This work is licensed under a Creative Commons Attribution International 4.0 License.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695543

For vulnerability detection and characterization, automated techniques such as fuzzing, symbolic execution, and static analysis have been used in many domains. However, firmware development, particularly for UEFI, faces unique challenges due to its specific execution environment and the extensive interaction with hardware [48]. Existing methodologies exhibit notable trade-offs that limit their effectiveness in comprehensive firmware analysis. For instance, while static analysis offers scalability, it often lacks precision and is prone to generate false positives due to its reliance on approximation methods [21]. Conversely, symbolic execution provides high-precision but struggles with scalability and the generation of effective harnesses when applied to complex firmware codebases that involve numerous hardware interactions [28]. Additionally, the architecture of UEFI presents challenges to traditional fuzzing techniques, as vulnerabilities typically manifest as silent corruptions rather than crashes, making them difficult to detect [49]. Furthermore, current fuzzing tools fail to accommodate the interdependencies between different UEFI entry points and lack a nuanced understanding of the complex input structures and execution contexts specific to the targets they are testing [49].

The integration of expert guidance with automated testing and verification techniques enhances the effectiveness of vulnerability analysis [33] such as directing symbolic execution or fuzzing towards potential error sites. However, the traditional manual generation of symbolic execution harnesses is both time-consuming and error-prone, particularly due to the intricate nature of UEFI's integration with hardware and its complex configuration settings. Additionally, this process requires deep expertise in symbolic execution and a thorough understanding of UEFI. Hence, manual harness generation is a significant barrier for developers and limits the adoption of symbolic execution in production environments [19].

In response to these challenges, this paper introduces a scalable and precise technique named Static Analysis guided Symbolic Execution (STASE), which leverages expert inputs as static analysis rules. These rules, derived from common vulnerability patterns, are extensible by developers' domain knowledge and enables the automatic generation of symbolic execution harnesses from the static analysis output. Figure 1 illustrates the components of the STASE framework. The approach begins with a rule-based static analysis to identify potential vulnerabilities, producing a comprehensive vulnerability description that includes entry points, attacker-controlled sources, vulnerability locations, affected program instructions (sinks), and an assertion to confirm the vulnerability. This description is subsequently utilized to generate the symbolic execution harness, thereby eliminating the need for manual intervention and significantly enhancing scalability by directing the symbolic execution towards specifically vulnerable locations within the program. Finally, symbolic execution employs this framework to generate detailed vulnerability signatures, encompassing both the preconditions and postconditions of the identified vulnerabilities.

Our overall technical contributions can be summarized as follows:

(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.

2 UEFI VULNERABILITY SIGNATURES

UEFI comprises millions of lines of code from various sources, including EDKII, the open-source implementation by TianoCore [44], along with contributions from motherboard firmware companies and manufacturers. This diverse supply chain results in a large attack surface for UEFI [24]. Due to UEFI's privileged role in the boot process and its ability to bypass security protections, vulnerabilities within it present serious security risks, making UEFI a prime target for exploitation [36].

Among the most critical components of UEFI are the System Management Mode (SMM) drivers, which operate in an isolated processor environment hidden from the operating system and applications. Access to SMM is controlled by System Management Interrupts (SMIs), with code execution confined to System Management RAM (SMRAM), a memory space that firmware must keep inaccessible to other CPU modes. However, vulnerabilities in SMM drivers can compromise this isolation, leading to real-world attacks [34, 35, 37, 38, 47]. In addition, issues like buffer overflows or integer underflows can escalate into significant security risks. A notable example is PixieFail, a set of vulnerabilities recently disclosed by Quarkslab [39], affecting the network module of EDKII.

As an example, let us discuss an "SMRAM Write" UEFI vulnerability. This type of vulnerability arises from improper handling of memory within the SMM, where non-SMM code, such as the operating system kernel or user applications, is able to write to SMRAM. To identify such vulnerabilities, we can define conditions (stated as assertions) that characterize the intended behavior of the system, such that their violation indicates a vulnerability. Negation of such an assertion can be used to identify the "vulnerability condition" which indicates a vulnerability when it holds. The vulnerability condition for the SMRAM Write can be expressed as follows: $\neg(\texttt{BufferSize} \leq \texttt{SMRAM\_BASE} + \texttt{SMRAM\_SIZE} \wedge \texttt{Buffer} \leq \texttt{SMRAM\_BASE} + \texttt{SMRAM\_SIZE} \wedge (\texttt{BufferSize} = 0 \vee \texttt{Buffer} + \texttt{BufferSize} \leq \texttt{SMRAM\_BASE} + \texttt{SMRAM\_SIZE}))$ This condition validates whether or not the buffer provided to the SMI handlers overlap with SMRAM. The negation here reflects the violation of the expected condition.

Consider the code snippet in Listing 1 from `SmramProfileRecord.c` file of EDKII. This code performs a write operation on the buffer (`CommBuffer`) passed to an SMI handler (`SmramProfileHandler`). Writing to the `CommBuffer` through the `SmramProfileHandler` function without any overlap check could potentially overwrite SMRAM if the `CommBuffer` overlaps with SMRAM, leading to unintended behavior.

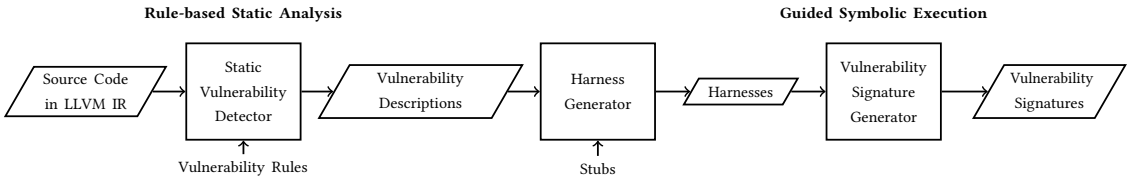


Figure 1: STASE Technique

We can characterize this unintended behavior to extract the vulnerability signature that holds the preconditions required to reach the vulnerable location, the code segment containing the vulnerability, and the postconditions resulting from the vulnerability.

```
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:

$$\{Precondition\}\ CodeSegment\ \{Postcondition\}$$

3 RULE-BASED STATIC ANALYSIS

The application of traditional static analysis techniques to vulnerability detection often results in many false positives due to their reliance on approximation methods [21]. In recent years, rule-based static analysis techniques have enhanced the precision of vulnerability analysis while enabling specialization across various domains. These approaches leverage domain-specific languages for rule specification, with Datalog—a declarative logic programming language and syntactical subset of Prolog—being widely used to specify program analysis rules [14, 16, 30, 41, 45]. Program analysis queries correspond to fixed-point computations which can be captured as evaluations of logic programs. Hence, once static analysis rules are expressed in Datalog, a Datalog engine can be used to compute the results of the static analysis. In this paper, we use Soufflé [32, 42], which is an expressive dialect of Datalog and provides a performant logic program evaluation engine for efficient rule-based static analysis. Furthermore, we use an open-source cclyzerpp [16, 17] tool built on Soufflé which specifies rules for field and structure-sensitive pointer analysis for LLVM-IR source code.

3.1 Semantics and Rules

In this section we briefly overview the LLVM-IR semantics and describe Datalog-based static analysis rules.

3.1.1 LLVM-IR Semantics. We define the stack frames and program states based on the formalization of LLVM-IR [51] as follows:

Definition 3.1. A frame Σ is defined as $(f, l, r, t, \Delta, \alpha)$ where f tracks the current function corresponding to the frame, l is LLVM block label, r is the next instruction in the current LLVM block to be executed next, t is termination instruction in the current block, Δ is the mapping function to keep track of the local variables to their values in LLVM and α is the memory block stack that is allocated within the function.

Definition 3.2. A program state $S_P \in \mathbb{S}_P$, where $\mathbb{S}_P$ is the state space for a given program P, is defined as $S_P \equiv (M, \overline{\Sigma})$ where $M = (N, B, C)$ is a memory state as defined in [51], N is the next block to allocate, B maps a valid block identifier to the size of the block, and C maps a block identifier and an offset within the block to a memory cell (if the location is valid) and $\overline{\Sigma} = \{\Sigma_1, \ldots, \Sigma_{nc}\}$ is a stack of frames, with Σ_1 being top, that keeps track of local variables and current instruction, where nc is the number of frames in the stack.

Let $\text{Attr}(S_P)$ and $\text{Val}(S_P)$ be two sets defined over a program state S_P that denote all the attributes and their values in S_P. For example, the function field of the current frame $\Sigma.f$ is a member of $\text{Attr}(S_P)$ and its value, say `main`, is a member of $\text{Val}(S_P)$.

We define facts and relations based on the program states:

Definition 3.3. A relation Q_T is defined as the collection of attributes $\{a_1, \ldots, a_n\}$ representing a program property T with each attribute being individually mapped to an attribute of the program state: $Q_T = \{a_1, \ldots, a_n\}$ where $\forall i \in [n], a_i \in \text{Attr}(S_P)$.

Definition 3.4. A fact $\mathsf{F}(Q_T)$ is defined as a set of tuples that correspond to the values of the attributes in relation Q_T in some program state:

$$\mathsf{F}(Q_T) = \left\{(b_1, \ldots, b_n) \mid \exists S_P \in \mathbb{S}_P, \forall i \in [n], a_i = b_i \wedge b_i \in \mathrm{Val}(S_P) \wedge a_i \in Q_T\right\}$$

For example, the membership property of a function with respect to an LLVM instruction can be expressed as relation $\{a_1, a_2\}$ where $a_1 = S_P \cdot \Sigma_1 \cdot r$ and $a_2 = S_P \cdot \Sigma_1 \cdot f$ where $S_P \in \mathbb{S}_P$ is a program state. Fact, $\mathsf{F}(\{S_P \cdot \Sigma_1 \cdot r, S_P \cdot \Sigma_1 \cdot f\})$ say `instr_func`, would be the collection of values for the instruction and function attributes in program states. A fact can be considered as the set of witnesses for the attribute relation. Note that symbol $\cdot$ is used to access a member (field) of a structure. For example, $S_P \cdot \Sigma_1 \cdot f =$ `GetSmramProfileContext()` means that `GetSmramProfileContext()` function is associated with attribute $S_P \cdot \Sigma_1 \cdot f$, denoting current function f in the top stack frame Σ_1 of program state S_P.

```
; Function Attrs: noimplicitfloat noinline noredzone nounwind
      uwtable
define dso_local %struct.MEMORY_PROFILE_CONTEXT_DATA*
      @GetSmramProfileContext() #0 {
  %1 = load %struct.MEMORY_PROFILE_CONTEXT_DATA*, %struct.
       MEMORY_PROFILE_CONTEXT_DATA** @mSmramProfileContextPtr,
       align 8
  ret %struct.MEMORY_PROFILE_CONTEXT_DATA* %1
}
```

Listing 2: An example function in LLVM-IR

Consider the LLVM-IR code snippet from Listing 2. Based on the code snippet, `instr_func` fact will hold the following fact:
`(<SmramProfileRecord.bc>:GetSmramProfileContext:0, <SmramProfileRecord.bc>:GetSmramProfileContext)`
where the first entry represents LLVM instruction (index 0) on line 3, and the second entry represents the function to which the instruction belongs.

3.1.2 Rules for Static Analysis. In Datalog-based static analysis, rules are specified as horn clauses. Formally, a declarative static analysis rule is defined as follows (see [42] for rule syntax):

Definition 3.5. We define a rule $\mathsf{R}(C)$ as a horn clause over a set of terms $C = \{c_1, \ldots, c_n\}$ where term $c_1 = \{q_1, \ldots, q_n\}$ is a collection of attributes used to represent the underlying program property, and the rest are either a relation or negation of a relation or a constraint or a rule in itself: $\mathsf{R}(C) \equiv c_1 \leftarrow c_2 \wedge c_3 \ldots \wedge c_n$ where rule $\mathsf{R}(C)$ means that: If $c_2 \wedge c_3 \ldots \wedge c_n$ holds, then c_1 can be inferred.

For a given rule $R(C)$, the set of program states where c_1 can be inferred is categorized as rule qualifying set $P_{\text{set}}^{\mathsf{R}(C)}$ under rule-relations. Internally, the datalog engine views these rules as relations and computes them over the facts and constraints till a fixed point is reached which results in $P_{\text{set}}^{\mathsf{R}(C)}$ for the corresponding rule. The attributes of these states are extracted and mapped onto the vulnerability descriptions are defined at the end of this section.

3.2 Vulnerability Rules for UEFI

Our rule-based static analysis involves specifying the vulnerability rules, taint tracking, and program slicing to generate vulnerability descriptions.

3.2.1 Vulnerability Rules. We use facts from cclyzerpp [17] on LLVM-IR to design customizable and reusable vulnerability detection rules. In addition to writing rules for common vulnerability types such as division by zero and buffer overflow, we extend our rule set with UEFI-specific rules to detect specification violations known to lead to vulnerabilities from CVEs related to UEFI. As demonstrated in previous work[18], UEFI contains over ten categories of vulnerabilities, each with distinct variations and exploitation points. The following is a list of vulnerabilities for which we provide rules with the current version of the STASE tool:

- 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

Our vulnerability rules are expressed over facts and constraints, with facts serving as the fundamental building blocks that enable static vulnerability analysis. To generate facts, we use cclyzerpp's built-in LLVM passes that populate fact relations with program facts based on the abstract syntax tree (AST) of the LLVM modules. For example, the fact relation `instr_func(?instr, ?func)` holds the ordered tuple `(?instr, ?func)`, where `?instr` represents an LLVM instruction, and `?func` corresponds to the function containing that instruction. These fact relations abstract away program-specific details such as variable and function names from vulnerability rules, ensuring that the rules remain generalizable and customizable.

In isolation, facts are insufficient for conducting complex vulnerability analysis. As described earlier, vulnerability rules allow us to impose conditions that facts must satisfy to qualify as a program property. These conditions are formulated using fact relations, other vulnerability rules, or constraints on attributes. For example, to capture the three variations of LLVM division instructions, we define a rule called `divisioninstructions(?divid, ?divis, ?instr)`, where `?instr` identifies a potential division instruction, and `?divid` and `?divis` represent the dividend and divisor associated with the instruction, respectively. We specify the rule for detecting the `udiv` instruction as follows:

```
divisioninstructions(?divid, ?divis, ?instr):-
  udiv_instr(?instr),
  udiv_instr_first_operand(?instr,?divid),
  udiv_instr_second_operand(?instr,?divis).
```

where `udiv_instr(?instr)` is a fact relation that contains `udiv` instructions within the LLVM file. `udiv_instr_first_operand(?instr, ?divid)` and `udiv_instr_second_operand(?instr,?divis)` are also fact relations that match the `udiv` instruction to the first and second operators respectively. Based on LLVM syntax, the first and second operands of `udiv` instruction map to dividend and divisor, respectively. Following the same pattern, we extend the rule to include the `sdiv` and `fdiv` instructions. Furthermore, the `divisioninstructions` rule can be expanded to detect function calls that perform division internally

by leveraging Soufflé's `substr` constraint, allowing for the identification of division operations within function bodies. This approach ensures comprehensive coverage of division-related vulnerabilities in the program.

Based on this rule, we define our divide-by-zero vulnerability rule by adding more facts such as `instr_pos` that maps the LLVM instruction to the line number in the source code using LLVM debug information. The final divide by zero vulnerability rule is as follows:

```
division_primitive(?func, ?divid, ?divis, ?instr, ?line):-
  instr_func(?instr, ?func),
  divisioninstructions(?divid, ?divis, ?instr),
  instr_pos(?instr,?line,?col).
```

It is important to note that we do not perform any value flow analysis to determine whether the divisor could potentially be zero. This choice is intentional to maintain the tool's lightweight and scalable design, though this may result in false positives, which are filtered out during symbolic execution. Crucially, the vulnerability rules themselves remain unchanged even as the underlying facts vary between different programs. Therefore, once a vulnerability rule is written, it can be reused to analyze any code.

3.2.2 Attacker-controlled Taint Tracking (AcTT). One missing aspect of the vulnerability rules discussed above is exploitability. While we may detect a division by zero vulnerability (e.g., when the divisor is set to 0), it might not be exploitable if the attacker cannot control the divisor's value. To focus on exploitable vulnerabilities, we need to disregard cases that cannot be triggered by an attacker. To achieve this, we incorporate attacker-controlled taint-tracking into our vulnerability analysis.

Taint tracking works by defining attacker entry points and attacker-controlled inputs. Entry points are functions an attacker can invoke, and attacker-controlled inputs are variables, often function arguments or global variables, that the attacker can manipulate. In the context of SMI handlers, triggering an SMI through a kernel module, such as writing to a port that triggers SMIs, can be considered an entry point. This approach refines our analysis to focus on vulnerabilities that are truly exploitable by an attacker.

In our vulnerability rules, we designate SMI handlers as attacker entry points. Since SMM code is protected in the SMRAM region, the only way to interact with it is through a communication buffer, specifically the `CommBuffer` and `CommBufferSize` parameters of the SMI handler. Thus, we treat `CommBuffer` and `CommBufferSize` as attacker-controlled inputs. By doing so, we can refine our vulnerability rules to detect potential vulnerabilities triggered by the contents of these parameters. The attacker entry points and controlled inputs are defined manually as expert input, based on domain specifications and CVE descriptions. Let us consider an example. where we tag an SMI handler named `SmramProfileHandler` as an attacker entry point, `CommBuffer` as attacker-controlled input and use the following rule:

```
entrypoint(?func):-
  func_name(?func, "@SmramProfileHandler").
entryinput(?taintentry):-
  entry_point(?func),
  func_param(?func, ?taintentry, 2).
```

where `func_param` is a fact relation that maps a function to its argument at the given index.

For taint tracking, we use attacker-controlled input as the taint source. We define a taint sink as the variable at the vulnerable instruction. It usually varies depending on the specific vulnerability. For instance, in a division by zero vulnerability, the taint sink is the divisor operand, while in an integer overflow vulnerability, the taint sink could be either operand. This flexibility allows our taint-tracking approach to adapt to different vulnerability behaviors.

To perform taint-tracking, we use pointer analysis of cclyzerpp, which is based on field and structure-sensitive Andersen-style pointer analysis [16]. Using the points-to information, the taint-tracking rule adds the condition that both taint-source and taint-sink should point to the same allocation site in the program. The taint-tracking rule is written as follows:

```
tainttacking(?taintsource,?taintsink):-
  entryinput(?taintsource),
  subset.var_points_to(_,?samealloc,_,?taintsource),
  subset.var_points_to(_,?samealloc,_,?taintsink).
```

Then the modified division-by-zero vulnerability rule with taint-tracking is specified as follows:

```
divisor_tainted_division_primitive(?func, ?divid, ?divis, ?taintsource,
?instr, ?line) :-
  tainttacking(?taintsource, ?divis),
  instr_func(?instr, ?func),
  divisioninstructions (?divid, ?divis, ?instr),
  instr_pos(?instr,?line,?col).
```

Notice that the flow-insensitive nature of pointer analysis from cclyzerpp results in over-approximation. It is possible that the taint-source and taint-sink no longer points to the same allocation site at the potentially vulnerable location. However, taint-tracking still reduces the number of false positives and contributes to the vulnerability description used for symbolic execution harness generation for each potential vulnerability by providing a starting point for symbolic execution in the form of an entry point, potential symbolic variables in the form of taint-sources, and variables to be used in assertion template in the form of taint-sinks.

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].

3.2.4 Vulnerability Descriptions. Once the static vulnerability detector identifies a potential vulnerability target, it produces a vulnerability description. For each vulnerability target, a vulnerability description provides target-specific information as defined below. Later, the harness generator component uses the vulnerability description to guide symbolic execution for each vulnerability target.

Definition 3.6. Given a program P and a series of program states $(S_P^e, \ldots, S_P^t)$ where S_P^e is the program state at the attacker-controlled

entry point of program P and S_P^t is the program state at vulnerability target, we define the vulnerability description as a 7-tuple representation $V_{S_P^t} = \langle P, E, I, A, K, L, U \rangle$ where

- 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 $I = \{v_1, \ldots, v_m \mid \forall i \in [m],\ v_i = S_P^e \cdot \Sigma_1 \cdot \Delta(v_i)\}$
- A is the assertion template corresponding the vulnerability category, which is defined as $A = \texttt{assert}(B(l_1, \ldots l_n))$ such that $\forall i \in [n], l_i = S_P^t \cdot \Sigma_1 \cdot \Delta(v_i)$ where $B(l_1, \ldots l_n)$ are conditions over LLVM variables $(l_1, \ldots l_n)$
- 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 $U = \{S_P^i \cdot \Sigma_1 \cdot r \cdot \text{LOC} \,|\, S_P^i \in (S_P^e, \ldots, S_P^t) \wedge S_P^i \notin \text{Slice}(S_P^e, \ldots, S_P^t)\}$

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 = `injected_Tcg2Smm.c`, E = `TpmNvsCommunciate`,
K = < `injected_Tcg2Smm.bc` >: `TpmNvsCommunciate : 32`,
A = `assert(TempCommBufferSize ≠ 0)`, I = `CommBufferSize`,
L = `injected_Tcg2Smm.c : 70`, U = {`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

4 HARNESS GENERATION

The symbolic execution harness establishes the environment and supplies the necessary context for symbolic execution. It has two main components: the environment configuration and the path exploration guidance. Environment configuration involves setting up the system's initial state and context to reflect realistic execution conditions. This includes modeling the external libraries, system calls, and hardware interactions.

Path exploration guidance involves directing the symbolic execution toward the program behaviors likely to expose bugs or vulnerabilities. This includes identifying suitable entry points and exit conditions, determining symbolic and concrete variables, and adding bounds on depth and loops.

A STASE symbolic execution harness consists of an Environment Configuration Harness (ECH), which is created once for a UEFI codebase, and a Path Exploration Harness (PEH), which is automatically generated using the vulnerability descriptions generated during the static analysis phase of STASE. Typically, developers generate these harnesses manually which requires in-depth knowledge of symbolic execution and the system under investigation. STASE automates this process, improving the usability and scalability of symbolic execution.

4.1 Environment Configuration Harness (ECH)

STASE leverages basic UEFI domain knowledge, which can be extracted from UEFI specification [29], and provides an Environment Configuration Harness (ECH) for UEFI vulnerability analysis. This sets up the symbolic execution configuration once, establishing a baseline environment that accurately emulates UEFI configurations and hardware interactions.

4.1.1 Global Table: Global system and services tables (e.g., EFI System Table, Boot Services Table, and Runtime Services Table) provide standardized interfaces for UEFI applications, drivers, and the underlying system to interact. However, direct interaction with these global tables are neither practical nor necessary during symbolic analysis because the symbolic execution environment does not execute the entire UEFI runtime environment at once. Instead, communication and data manipulation are abstracted, focusing on the program logic. Internal functions or variables typically accessed via global tables are exposed through file references. This abstraction is essential for accurately simulating the firmware's behavior in a controlled environment, allowing the analysis to concentrate on the code's logic and functionality without being constrained by the specifics of the UEFI execution environment.

4.1.2 PCD-dependent Variable: The Platform Configuration Database (PCD) enables developers to customize the firmware for specific operational requirements. However, during the symbolic analysis, the exact values of PCD-dependent variables may be unknown. Moreover, relying on a particular PCD value for symbolic execution can limit the analysis to predetermined configurations, potentially missing vulnerabilities that may arise under different settings. To address these limitations and enhance the comprehensiveness of the symbolic execution, symbolic values are assigned to PCD-dependent variables. This enables PCD-dependent variables to simulate various configuration scenarios the firmware might encounter in real-world settings. For instance, the PCD variable `PcdReclaimVariableSpaceAtEndOfDxe` determines whether a certain memory space is reclaimed at the end of the DXE phase. By assigning a symbolic value to this variable, the symbolic execution can explore both scenarios: one where the memory space is reclaimed and one where it is not.

4.1.3 Firmware Parameter: Assigning symbolic values to firmware parameters decouples the analysis environment from the hardware infrastructure. This approach allows the symbolic execution engine to explore a broader range of hardware configurations without relying on a fixed setup. For example, the SMRAM base address and size, which dictate the memory region reserved for system management functions, can be treated as symbolic. By doing so, the analysis can consider various possible values for the SMRAM base address and size, enabling the exploration of different memory configurations. Similarly, other critical parameters on memory boundaries, such as `mSmmMemLibInternalMaximumSupportAddress`, can also be treated symbolically to explore their implications on firmware security. This approach ensures a comprehensive examination of the firmware under diverse conditions, identifying potential vulnerabilities that may not be apparent with static configurations.

4.1.4 Protocol GUID:. A GUID (Globally Unique Identifier) in UEFI is a 128-bit number that is used to identify items such as devices and protocols. Protocol GUIDs are essential for identifying and interacting with the protocols defined within the UEFI specification. Since symbolic execution harness abstracts parts of the system behavior, during symbolic execution specific runtime values for

the protocol GUIDs are not generated. Predefining these GUIDs with default values ensures that the symbolic execution engine can accurately simulate the firmware's interactions with various UEFI protocols. The UEFI Specification documents [29], maintained by the UEFI Forum, contains detailed information about the protocol GUIDs. These specifications include a list of protocol GUIDs along with their corresponding values and descriptions. STASE UEFI ECH contains the default values for protocol GUIDs obtained directly from the `inf` files (Information Files that are used to describe the metadata and build configuration for UEFI modules or drivers).

4.2 Path Exploration Harness (PEH)

STASE uses static analysis output to automate the path exploration harnesses. As discussed earlier, each potential vulnerability detected by the static analysis phase of STASE is characterized as a vulnerability description $V_{S_P^t} = \langle P, E, I, A, K, L, U \rangle$. Vulnerability descriptions are mapped to the symbolic execution harness to guide symbolic execution towards the vulnerability location (L), ensuring a targeted and efficient exploration of program behaviors.

4.2.1 Symbolic Analysis Entrypoint: To invoke the symbolic execution without any manual intervention, STASE uses the same entry points for symbolic execution used during static analysis. This alignment allows for a seamless transition from static analysis to symbolic execution where E is the field of the vulnerability description $V_{S_P^t}$ is used as the *symbolic analysis entry point.*

4.2.2 Symbolic Variables: Symbolic values are allocated to the taint sources identified during static analysis, allowing the symbolic execution engine to explore how external inputs (untrusted data locations) can affect the firmware's execution paths and potentially lead to vulnerabilities. UEFI external inputs typically use a structured data type comprising multiple fields, static analysis reveals the specific fields contributing to the vulnerability. To make the symbolic execution effective, only the contributing fields are defined as symbolic. The I field of the vulnerability description $V_{S_P^t}$ identifies the *symbolic variables* for PEH.

Consider an example involving an SMI handler, `SmramProfileHandler`, which processes data from a non-SMM environment into an SMM environment and uses `CommBuffer` as the external input. `CommBuffer` uses a structure data type comprising multiple fields; however, static analysis reveals that only `Command` and `ReturnStatus`, contributes to the targeted vulnerability. Based on this insight, these fields are designated as symbolic within the symbolic execution harness.

```
typedef struct { SMRAM_PROFILE_PARAMETER_HEADER Header;
BOOLEAN RecordingState;
} COMMBUFFER_STRUCT;
COMMBUFFER_STRUCT *CommBuffer=malloc(sizeof(COMMBUFFER_STRUCT));
CommBuffer->Header.Command=klee_int("CommBuffer->Header.Command");
CommBuffer->Header.ReturnStatus=klee_int("CommBuffer->Header.
    ReturnStatus");
```

Listing 4: Declaring Symbolic Variables

4.2.3 Global Variables: As mentioned earlier, the symbolic execution engine uses the same entry points as static analysis. However, the global variables explored by symbolic execution can be updated in other parts of the code and are accessible from different entry points. This is particularly relevant to cross-handler interactions, where different event handlers and protocols interact through shared global variables. Therefore, relevant global variables must be declared symbolic to allow the symbolic execution engine to reason about the influence of external inputs and internal state changes that occur outside the immediate scope of the currently analyzed code segment.

STASE identifies the relevant global variables as part of its static analysis technique by implementing program slicing to isolate portions of the code relevant to the verification goal. These global variables are then used as the taint sources and included in the I field of the vulnerability description $V_{S_P^t}$. Hence, these global variables are also included in PEH's symbolic variable list.

For example, the global variable `mVariableBufferPayload`, accessed by multiple handlers and protocols, is identified as a taint source for various entry points. This allows the symbolic execution engine to use it as symbolic and consider various possible states of `mVariableBufferPayload`, thereby simulating the effects of external inputs and internal state changes across different parts of the code.

4.2.4 Call Depth: STASE optimizes the call depth of symbolic execution by managing the breadth of the firmware's codebase, allowing for a more targeted analysis. This optimization minimizes exploring the functions that do not impact the vulnerability under investigation. This process is implemented by taking the output of slicing (the U field of vulnerability description $V_{S_P^t}$) and replacing the functions with stubs. These stubs mimic the basic interface and effects of the original functions but exclude the program logic that is non-critical to the specific vulnerabilities being explored. Consider the following example within the context of SMRAM profile management in the EDKII codebase:

```
1 ContextData = GetSmramProfileContext();
2 if (ContextData == NULL) { return;}
3 SmramProfileGettingStatus  = mSmramProfileGettingStatus;
4 mSmramProfileGettingStatus = TRUE;
5 CopyMem(&SmramProfileGetData,SmramProfileParameterGetData,sizeof(
      SmramProfileGetData));
```

Listing 5: Code snippet containing Non-critical Function

In this example, `GetSmramProfileContext()` is responsible for obtaining context data critical for SMRAM profile management. However, for the purposes of vulnerability analysis related to `CopyMem`, the detailed operations within `GetSmramProfileContext()` are not directly relevant.

```
MEMORY_PROFILE_CONTEXT_DATA* EFIAPI GetSmramProfileContext(VOID) {
// Initialize a symbolic variable for the context data
MEMORY_PROFILE_CONTEXT_DATA *symbolicContextData;
klee_make_symbolic(&symbolicContextData, sizeof(
    symbolicContextData), "symbolicContextData");
return symbolicContextData;
}
```

Listing 6: Stub Implementation

The stub implementation replaces the original function, restricting the call depth within the function. Instead of engaging with detailed firmware data structures, this stub returns a symbolic variable, `symbolicContextData`. As illustrated in the stub, the minimum implementation is crucial because it maintains control flow integrity. If we remove the function call entirely, the subsequent null check `(ContextData == NULL)` will fail, disrupting the control flow and leading to inaccurate analysis.

4.2.5 Assertions. Static analysis marks the locations as vulnerable where tainted data—data derived from untrusted sources can influence the behavior of the firmware, potentially leading to security breaches. Assertions are added before the taint sinks to enforce constraints or expectations about the program's state at those code points. This helps to eliminate the false positives generated by the static analysis. If an assertion fails during symbolic execution, it indicates a possible vulnerability where attacker-controlled input could manipulate the program's state in unexpected and potentially harmful ways. However, if symbolic execution does not detect an assertion violation, we know that the vulnerability does not exist (within the execution depth explored by symbolic execution).

Steps to add Assertions:

(1) Identify the location: Assertion locations are identified as part of the vulnerability description. This information is provided in the L field of the vulnerability description $V_{S_P^t}$.
(2) Formulate assertions: The static analysis phase of STASE outputs an assertion template (as illustrated in the listing 7) with each vulnerability description $V_{S_P^t}$. The A field holds the assertion template.
The assertion template is filled with the taint sinks to generate the assertion. The A field of vulnerability description $V_{S_P^t}$ holds the taint sinks that can be extracted from the assertion template.

```
Assertion Template: assert(%i >= 0 && %i < Sizeof(%j))
Assertion: klee_assert(arrayIndex >= 0 && arrayIndex <
      bitmapSize) ;
```

Listing 7: Example of Assertion Template & Assertion

(3) Implement assertions in code: The assertions are integrated within the firmware code being analyzed.

4.2.6 Loop Bounds. Symbolic execution cannot handle unbounded loops. During the static analysis phase of STASE, loops that do not have a constant loop bound are identified and during symbolic execution they are limited to a fixed number of loop iterations.

5 GUIDED SYMBOLIC EXECUTION

Given a vulnerability description $V_{S_P^t} = \langle P, E, I, A, K, L, U \rangle$, the Environment Configuration Harness (ECH) and the Path Exploration Harness (PEH) instruments the program P and generate an instrumented code segment Θ with the following properties:

- Θ is derived from the program P, containing only the sliced code without line numbers in U, and has specific call depth and loop bounds as specified in PEH.
- Θ includes a designated program entry point E with a set of symbolic arguments I.
- An assertion generated using template A is injected within Θ at the location L.

Guided symbolic execution takes Θ as input and operates as follows:

(1) **Initialization:** Execution begins with an initial program state S_P^e containing the entry point and symbolic variables.
(2) **Targeted Exploration:** As execution progresses, the symbolic execution engine explores paths with the aim of reaching the location L. This involves continually updating through the program state $S_P^i = (M, \overline{\Sigma}) \in \{S_P^e, \ldots, S_P^t\}$ by modifying M (memory state) and $\overline{\Sigma}$ (stack of frames) to reflect only those paths that are relevant to reach instruction K. The exploration is constrained by the specified call depth and loop bounds.
(3) **Assertion Validation and State Logging:** : Upon reaching K, the assertion A is evaluated. If the assertion fails, indicating a vulnerability, the current program state S_P^t is logged. This failure captures the memory state M at the point of failure and adjusts $\overline{\Sigma}$ to reflect the path constraints π that led to the vulnerability.

Guided symbolic execution outputs the confirmed vulnerabilities and their signatures. A vulnerability signature consists of the precondition, instrumented code segment containing the vulnerability, and the postcondition describing the immediate program state after triggering the vulnerability.

Precondition: The precondition, denoted as Π, is constructed by combining the set of path constraints for the paths that reach the vulnerability location and results in assertion violation such that $\Pi = \pi_1 \vee \pi_2 \vee \ldots \vee \pi_n$ where each π_i is a path constraint.

Instrumented Code Segment: The instrumented code segment Θ is included in the vulnerability signature.

Postcondition: The postcondition, denoted as Ω, corresponds to the assertion failure at the vulnerability location.

Combining these components, the vulnerability signature for a specific vulnerability can be represented as a Hoare triple:

$$\{\Pi\} \quad \Theta \quad \{\Omega\}$$

As an example consider an out-of-bounds access vulnerability in the `PxeBcHandleDhcp4Offer` function of EDKII with the following properties:

- Entrypoint: `PxeBcHandleDhcp4Offer@PxeBcDhcp4.c:1013`
- Assertion Location: `PxeBcParseVendorOptions@PxeBcDhcp4.c:232`
- Assertion Condition: `arrayIndex >= 0` $\wedge$ `arrayIndex < bitmapSize`

After running a symbolic execution engine on Θ, an assertion failure occurs, indicating the vulnerability. The engine logs three path constraints leading to the assertion condition, which is reported as the precondition (Π). It also logs the stack traces that reported as the postcondition(Ω):

```
1)Precondition:-
(Private->SelectIndex > 0 and Private -> SelectIndex - 1 <
      PXEBC_OFFER_MAX_NUM and 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

6 EXPERIMENTAL EVALUATION

STASE targets UEFI source code written in C. We use clang-14 to compile the UEFI source code into LLVM-IR. As mentioned

above, STASE implementation uses cclyzerpp [17] and Soufflé [32] for rule-based static analysis. We write our vulnerability rules on top of cclyzerpp's analysis rules. We use scripts to generate the environment configuration harness. We automatically generate a driver (path exploration harness) for each entry point based on the vulnerability descriptions computed during the static analysis phase. For symbolic execution, we use KLEE [13], and for constraint solving, we use Z3 [27]. To construct the vulnerability signatures, we use KLEE-generated smt2 [25] and KQuery [10] expressions.

We ran our experiments on a machine with 13th Gen Intel Core i9-13900K CPU at 3.00GHz and 192 GB of RAM running Ubuntu 22.04.4 LTS. STASE implementation is publicly available on GitHub[1] We experimented on five sets of UEFI implementations to evaluate STASE for UEFI vulnerability detection and characterization. The first and second sets of programs are from the EDKII [44], released as UEFI's "Foundation Code" under an open-source license. These two sets contain the following EDKII (release: edk2-stable202311) modules:

EDKII SMM Drivers: SMM drivers can access highly privileged data and control low-level hardware. SMI Handlers are the most important components of SMM drivers as they are the only channel to receive data from kernel-space programs. This makes them a potential target for attackers who can exploit vulnerabilities in the handlers.

EDKII Network Module: The network module processes data received over the network, and it is susceptible to attacks where an attacker can send malicious network packets to exploit vulnerabilities in the network stack. The third and fourth sets of programs (**HARDEN Demo 1**, **HARDEN Demo 2**) were collected from the challenges provided by the Defense Advanced Research Projects Agency (DARPA) for its Hardening Development Toolchains against Emergent Execution Engines (HARDEN) program. These challenge sets use the EDKII source (based on commit af8859bc) and add their own modules and handlers to inject vulnerabilities. In the final set of programs (**Injected EDKII**), we injected five types of vulnerabilities (Use After Free, SMRAM Read and Write, Integer Underflow, Buffer Overflow, and Division by Zero) in the EDKII. We ran our experiments on these five sets of programs using STASE and the following techniques (to compare with STASE's performance):

- **RbSA:** Rule-based Static Analysis with vulnerability rules
- **RbSA-AcTT:** Rule-based Static Analysis with Attacker-controlled Taint Tracking
- **SE:** Symbolic execution using KLEE
- **SE-ECH:** Symbolic execution using KLEE with Environment Configuration Harness
- **HBFA:** Intel's Host-Based Firmware Analyzer [40]

For symbolic execution and HBFA, we limited the experiment to an hour for each entry point. The big challenge in applying symbolic execution or fuzzing to firmware code is generating the harnesses, while STASE is designed to generate harnesses automatically. HBFA contains harnesses for some of the SMI handlers; still, we had to manually add harnesses to HBFA for other SMI handlers and module codes to make it comparable with STASE.

We experimentally evaluate STASE based on the following research questions:

[1]STASE implementation: https://github.com/shafiuzzaman-md/stase

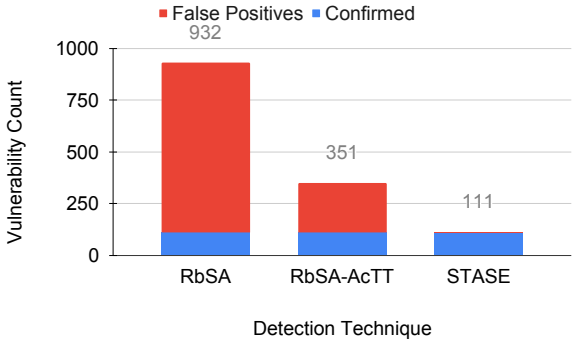
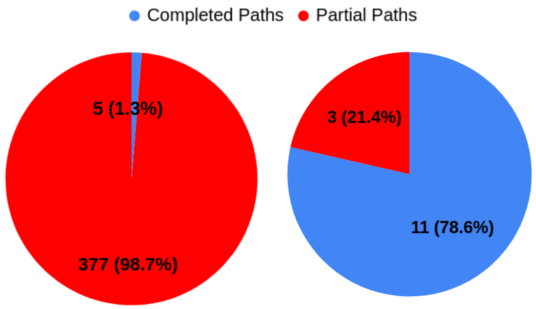


Figure 2: False Positive Reduction by STASE

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
Total	106	0		4		3		111	

Table 1: Comparison of Vulnerability Detection and Average Analysis Time (seconds) Per Entrypoint (EP)



(a) SE-ECH **(b) STASE**

Figure 3: Paths Explored per Entry Point

- **RQ1:** To what extent does STASE decrease the false positive rate compared to the rule-based static analysis?
- **RQ2:** Does STASE alleviate the path explosion problem in symbolic execution?
- **RQ3:** Does STASE significantly reduce the time required to identify vulnerabilities compared to traditional methods?
- **RQ4:** How effectively can STASE detect vulnerabilities compared to symbolic execution and fuzzing techniques?
- **RQ5:** Can STASE precisely identify vulnerability signatures compared to traditional methods?

RQ1: False positivity reduction by STASE: Figure 2 shows that STASE eliminates the false positives reported by rule-based static analysis (RbSA), enhancing the precision of vulnerability detection. We can also see that the false positivity rate is reduced significantly when the vulnerability rules are constrained with attacker-controlled taint tracking (RbSA-AcTT) without reducing

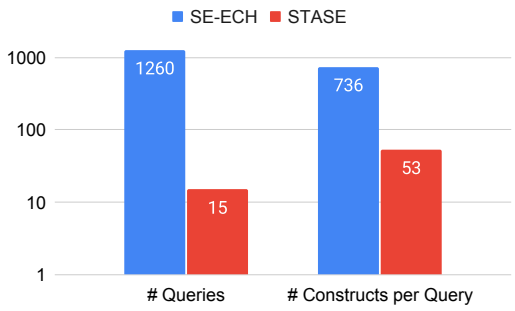


Figure 4: Constraint Solving Queries per Entry Point

the detection count for true positives. However, this is not enough to achieve full precision, which is only achieved after guided symbolic execution. We also tested Vandalir tool [41] on these datasets. The only matching category of Vandalir with our tool was buffer overflow. It was able to identify 152 targets with none of them being true positive. This is an expected result as Vandalir is not tailored for UEFI.

Table 2 demonstrates that STASE effectively eliminates false positives across all examined vulnerability categories, consistently maintaining a 0% false positivity rate. While rule-based static analysis also achieves 0% in very noticeable patterns such as SMM Callout (triggering EFI_BOOT_SERVICES inside code running in SMM), STASE's superior performance becomes evident in more generic categories like SMRAM READ and WRITE, where it reduces false positives by over 50%. Furthermore, STASE significantly enhances precision in detecting general security vulnerabilities such as integer overflow or buffer overflow, where the static analysis often yields numerous false positives. Considering all vulnerability types together, STASE reduces false positive rate from 88% for RbSA and for 68% for RbSA-AcTT to 0%.

RQ2: Alleviating path explosion with STASE: STASE is more effective than symbolic execution in path exploration (both for SE: symbolic execution with KLEE, and SE-ECH: symbolic execution with KLEE using Environment Configuration Harness). STASE explores significantly less number of paths, and majority of the paths it explores are completed (meaning that symbolic execution phase of STASE reaches an exit condition such as program termination or assertion failure). SE without ECH runs indefinitely without progressing due to missing assumptions (not modeled or stubbed) about the environment. This prevents the symbolic execution engine from completing its path exploration, producing no information about explored or completed paths. Hence, we compare STASE with SE-ECH which is able to complete some paths. Figure 3 shows the average number of paths explored by SE-ECH and STASE per entry point. Figure 3(a) shows that SE-ECH explores a total of 382 paths per entry point while only 1.3% of the explored paths are completed (corresponding to 5 completed paths per entry point on average), while a vast majority (98.7%) are partially explored. Hence, SE-ECH initiates many paths, but fails to explore them to completion, leading to inefficient resource usage. In contrast, Figure 3(b) shows that STASE explores only 14 paths per entry point but a substantial percentage (78.6%) of them are completed, corresponding to 11 completed paths per entry point on average. The integration of rule-based static analysis with program slicing and automated harness generation within STASE narrows down the scope of path exploration phase and enables it to focus on potentially vulnerable and relevant paths. This methodology not only reduces unnecessary computational overhead but also enhances the depth and relevance of the analysis, establishing STASE as a significantly more efficient tool for vulnerability detection.

Furthermore, due to its more effective path exploration strategy, STASE significantly reduces the number of satisfiability queries to the constraint solver, and therefore reduces the cost of the symbolic path exploration. Figure 4 shows that STASE generates only 15 queries per entry point whereas SE-ECH generates 1,260 queries. Moreover, the average size of the queries SE-ECH generates is higher than STASE (736 vs. 53 constructs per query). By reducing both the number and size of the satisfiability queries, STASE significantly improves the cost of symbolic path exploration.

RQ3: Vulnerability detection time reduction by STASE: Figure 5 compares the average analysis time across different techniques, indicating that STASE finishes exploration on average 27 seconds per entry point which is significantly faster than SE, SE-ECH, and HBFA (all timed out in all cases). Although static analysis methods are faster than STASE, they produce many false positives, as discussed earlier and shown in Figure 2.

Based on the data provided in Table 1, we can also compute the analysis time per reported vulnerability. Symbolic execution (SE) by itself is unable to find any vulnerabilities after 382K seconds of total analysis time. Symbolic execution with environment configuration harness (SE-ECH) uses 95K seconds of analysis time per vulnerability reported, and Intel's host-based firmware analyzer (HBFA) uses 127K seconds of analysis time per vulnerability reported, whereas STASE uses only 25 seconds of analysis time per vulnerability reported. This corresponds to 3,800X and 5,000X improvement in analysis time per reported vulnerability compared to SE-ECH and HBFA, respectively.

RQ4: Vulnerability detection improvement by STASE: Table 1 highlights the vulnerability detection improvement by STASE, revealing that STASE detected 111 vulnerabilities across the five datasets, whereas SE, SE-ECH, and HBFA detected far fewer or none. These results highlight that STASE strikes a good balance between reducing the overall analysis time and increasing the number of vulnerabilities detected compared to traditional symbolic execution and fuzzing methods, outperforming both in UEFI vulnerability analysis.

RQ5: Vulnerability signature generation by STASE: STASE stands out in its ability to identify vulnerability signatures without requiring any additional analysis time. In our evaluation, STASE successfully identified vulnerability signatures for all 111 vulnerabilities. In contrast, SE-ECH was able to generate vulnerability signatures for only 4 vulnerabilities, while SE failed to generate any. Fuzzing methods (that do not use symbolic analysis) are not able to generate signatures, as fuzzing typically provides a single input and lacks the capability to generate the precondition and postcondition needed for signature generation. Static analysis methods are not able to generate vulnerability signatures either due to their inherent limitations in capturing the preconditions. This demonstrates the superior precision and effectiveness of STASE in detecting and characterizing vulnerabilities compared to other approaches.

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%

Table 2: Comparison of False Positives (FP) among RbSA, RbSA-AcTT, and STASE

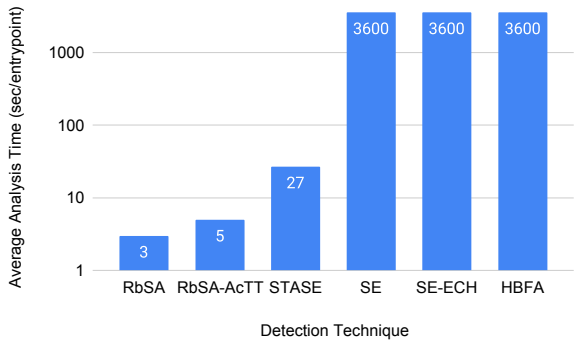


Figure 5: Comparison of Analysis Time across the Techniques

Discovered Vulnerabilities: For HARDEN Demo1, HARDEN Demo2, and Injected EDKII, STASE successfully identifies all the injected vulnerabilities. For EDKII SMM drivers, STASE identifies 13 new bugs that we reported to Tianocore. The 5 identified bugs in the EDKII network module correspond to the Pixiefail vulnerabilities.

Threats to validity: Our experimental results may be affected by hardware checks, which STASE assumes are sidestepped with specified attacker-controlled inputs. STASE's effectiveness relies on the accuracy of vulnerability rules; if they do not cover a vulnerability, STASE will not be able to identify it. Accurate specification of attacker-controlled entry points and inputs is crucial; otherwise, STASE will identify non-exploitable vulnerabilities. Additionally, the symbolic execution depth and loop bounds we set may also limit the exploration of potential vulnerabilities, as deeper or unbounded loops might contain hidden issues that remain undetected.

7 RELATED WORK

UEFI firmware vulnerabilities, though critical, receive less attention than OS and software vulnerabilities. Recent approaches to UEFI vulnerability detection include:

Symbolic execution. Bazhaniuk et al. [19] used symbolic execution to find vulnerabilities in UEFI firmware by analyzing SMRAM snapshots, generating 4000 test cases in 4 hours. However, no real bugs were reported, and the manual creation of test harnesses posed a significant challenge. State explosion and high computational costs remain obstacles.

Fuzzing. HBFA [40] from Intel has been developed to test UEFI drivers. It supports fuzzing frameworks like AFL, Libfuzzer, and Peach. HBFA's effectiveness is limited by the complexity of adding harnesses for each driver. We compared our approach with HBFA using AFL++ as the underlying fuzzing engine.

Hybrid analysis. A hybrid approach called RSFUZZER [49] has been proposed recently, combining fuzzing and symbolic execution; however, it does not produce the vulnerability signatures required for identifying exploitable vulnerabilities. This tool's emulation overhead results in slow performance, achieving only 18-42 executions per second compared to thousands by native fuzzers like AFL. RSFUZZER's effectiveness is also constrained by the availability and completeness of UEFI firmware images, as update bundles often miss important modules. Additionally, it struggles to detect silent vulnerabilities, which do not cause immediate system crashes. Although RSFUZZER claimed their tool would be public, they later indicated it could not be made public, preventing direct comparison.

Combining static analysis with symbolic execution. Symbiotic [23] combines static analysis with symbolic execution focusing on property verification and finding a witness that violates a property whereas STASE is tailored towards generating a vulnerability signature that can be used for characterization. Sys [20] merged user-extensible static checkers with under-constrained symbolic execution to achieve scalability and precision. Their static checkers rely on domain-specific knowledge to detect errors while reducing false positives. False positives still exist after under-constrained symbolic execution. Our technique sidesteps this issue with static and environment configuration harnesses. Recent work [22] on combining static analysis error traces with dynamic symbolic execution demonstrated a negative result. They concluded that the error traces do not provide enough guidance to the path exploration of DSE, and more often than not, static analysis techniques are imprecise. STASE shows improvement in both conclusions. Firstly, we show that combining rule-based static analysis and program slicing is enough to guide SE in achieving scalability rather than relying on traces. Moreover, we show that the accuracy of the static analyzers can be improved drastically by incorporating the notion of attacker-controlled taint-tracking.

8 CONCLUSION

STASE overcomes the challenges of detecting and characterizing UEFI vulnerabilities by combining the strengths of rule-based static analysis and guided symbolic execution. The static analysis phase identifies potential vulnerabilities and outputs detailed descriptions, including entry points, attacker-controlled inputs, and affected instructions. These descriptions are used to automatically generate symbolic execution harnesses, ensuring scalability and eliminating manual intervention. Guided symbolic execution then refines vulnerability detection, reduces false positives, and generates precise vulnerability signatures. STASE demonstrates its effectiveness by successfully identifying injected bugs, 13 new vulnerabilities, and 5 recently reported PixieFail vulnerabilities.

REFERENCES

[1] Cia vault 7 data leak: What do we know now? https://www.infosecinstitute.com/resources/hacking/cia-vault-7-data-leak-know-since-now/.
[2] Conti leaks reveal ransomware gang's interest in firmware-based attacks. https://thehackernews.com/2022/06/conti-leaks-reveal-ransomware-gangs.html.
[3] Cosmicstrand: the discovery of a sophisticated uefi firmware rootkit. https://securelist.com/cosmicstrand-uefi-firmware-rootkit/106973/.

[4] A deeper uefi dive into moonbounce. https://www.binarly.io/blog/a-deeper-uefi-dive-into-moonbounce.

[5] Finspy: the ultimate spying tool. https://usa.kaspersky.com/blog/finspy-for-windows-macos-linux/25559/.

[6] Hacking team spyware preloaded with uefi bios rootkit to hide itself. https://thehackernews.com/2015/07/hacking-uefi-bios-rootkit.html.

[7] Lojax uefi rootkit overview. https://h20195.www2.hp.com/v2/GetDocument.aspx?docname=4AA7-4019ENW.

[8] Malware delivery through uefi bootkit with mosaicregressor. https://usa.kaspersky.com/blog/mosaicregressor-uefi-malware/23419/.

[9] Mebromi bios rootkit. https://digital.nhs.uk/cyber-alerts/2018/cc-2565.

[10] The reference manual for the kquery language. https://klee-se.org/docs/kquery/.

[11] Trickbot malware gets uefi/bios bootkit feature to remain undetected. https://thehackernews.com/2020/12/trickbot-malware-gets-uefibios-bootkit.html.

[12] Uefi threats moving to the esp: Introducing especter bootkit. https://www.welivesecurity.com/2021/10/05/uefi-threats-moving-esp-introducing-especter-bootkit/.

[13] Klee 3.0, 2023.

[14] L. S. Amour. *Interactive Synthesis of Code-Level Security Rules*. PhD thesis, Northeastern University, 2017.

[15] M. Aung, S. Horwitz, R. Joiner, and T. Reps. Specialization slicing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):1–67, 2014.

[16] G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*, pages 84–104. Springer, 2016.

[17] L. Barrett and S. Moore. cclyzer++: Scalable and precise pointer analysis for llvm. https://galois.com/blog/2022/08/cclyzer-scalable-and-precise-pointer-analysis-for-llvm/, 2022.

[18] V. Bashun, A. Sergeev, V. Minchenkov, and A. Yakovlev. Too young to be secure: Analysis of uefi threats and vulnerabilities. In *14th Conference of Open Innovation Association FRUCT*, pages 16–24. IEEE, 2013.

[19] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer. Symbolic execution for BIOS security. In *WOOT*. USENIX Association, 2015.

[20] F. Brown, D. Stefan, and D. Engler. Sys: A {Static/Symbolic} tool for finding good bugs in good (browser) code. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 199–216, 2020.

[21] F. Brown, D. Stefan, and D. R. Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. In *USENIX Security Symposium*, pages 199–216. USENIX Association, 2020.

[22] F. Busse, P. Gharat, C. Cadar, and A. F. Donaldson. Combining static analysis error traces with dynamic symbolic execution (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 568–579, 2022.

[23] M. Chalupa, V. Mihalkovič, A. Řechtáčková, L. Zaoral, and J. Strejček. Symbiotic 9: String analysis and backward symbolic execution with loop folding: (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 462–467. Springer, 2022.

[24] J. Christensen, I. M. Anghel, R. Taglang, M. Chiroiu, and R. Sion. {DECAF}: Automatic, adaptive de-bloating and hardening of {COTS} firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1713–1730, 2020.

[25] D. R. Cok et al. The smt-libv2 language and tools: A tutorial. *Language c*, pages 2010–2011, 2011.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, 1989.

[27] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[28] J. Engblom. Finding bios vulnerabilities with symbolic execution and virtual platforms. Last-updated: 06/07/2019.

[29] U. Forum. Uefi specifications. https://uefi.org/specifications.

[30] B. Garmany, M. Stoffel, R. Gawlik, and T. Holz. Static detection of uninitialized stack variables in binary code. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II 24*, pages 68–87. Springer, 2019.

[31] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[32] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pages 422–430. Springer, 2016.

[33] I. B. Kadron, Y. Noller, R. Padhye, T. Bultan, C. S. Păsăreanu, and K. Sen. Fuzzing, symbolic execution, and expert guidance for better testing. *IEEE Software*, 41(1):98–104, 2023.

[34] X. Kovah and C. Kallenberg. Are you giving firmware attackers a free pass. In *Proceedings of the RSA Conference, San Francisco, CA, USA*, pages 20–24, 2015.

[35] ldpreload Yukari. BlackLotus UEFI Windows Bootkit. https://github.com/ldpreload/BlackLotus/, 2022.

[36] B. Mullen. Vulnerability management in uefi. https://uefi.org/sites/default/files/resources/Vulnerability%20Management%20in%20UEFI_Mullen.pdf.

[37] D. Oleksiuk. Exploiting ami aptio firmware on example of intel nuc, 2016.

[38] D. Oleksiuk. Thinkpwn. https://github.com/Cr4sh/ThinkPwn/, 2022.

[39] Quarkslab. Pixiefail: Nine vulnerabilities in tianocore's edk ii ipv6 network stack. https://blog.quarkslab.com/pixiefail-nine-vulnerabilities-in-tianocores-edk-ii-ipv6-network-stack.html.

[40] B. Richardson, C. Wu, J. Yao, and V. J. Zimmer. Using host-based firware analysis to improve platform resiliency. Technical report, Intel, 2019.

[41] J. Schilling and T. Müller. Vandalir: Vulnerability analyses based on datalog and llvm-ir. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 96–115. Springer, 2022.

[42] Souffle-Lang. Github - souffle-lang/souffle: Soufflé is a variant of datalog for tool designers crafting analyses in horn clauses.

[43] Statista. Global shipments of personal computers. https://www.statista.com/statistics/273495/global-shipments-of-personal-computers-since-2006/.

[44] tianocore. Edk ii. https://github.com/tianocore/tianocore.github.io/wiki/EDK-II.

[45] P. Tsankov. Security analysis of smart contracts in datalog. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV 8*, pages 316–322. Springer, 2018.

[46] M. Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[47] R. Wojtczuk and C. Kallenberg. Attacks on uefi security. In *Proc. 15th Annu. CanSecWest Conf.(CanSecWest)*, 2015.

[48] Z. Yang, Y. Viktorov, J. Yang, J. Yao, and V. Zimmer. Uefi firmware fuzzing with simics virtual platform. pages 1–6, 07 2020.

[49] J. Yin, M. Li, Y. Li, Y. Yu, B. Lin, Y. Zou, Y. Liu, W. Huo, and J. Xue. Rsfuzzer: Discovering deep smi handler vulnerabilities in uefi firmware with hybrid fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2155–2169. IEEE, 2023.

[50] J. Yin, M. Li, W. Wu, D. Sun, J. Zhou, W. Huo, and J. Xue. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1623–1637. IEEE, 2022.

[51] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 427–440, 2012.