FirmWall: Directed Symbolic Execution of Firmware Binaries for Defending against Unauthorized System Calls

Aruna Jayasena, *Student Member, IEEE,* and Prabhat Mishra, *Fellow, IEEE,*

***Abstract*—Modern computing devices rely on root-of-trust (RoT) to ensure confidentiality and integrity of both application code and data while satisfying a wide variety of user requirements. The RoT provides essential cryptographic and security functions as services (implemented as system calls) to the host system, supporting the execution of both trusted and untrusted applications. It also enables a secure boot process for the host operating system and other functionalities to establish a trusted execution environment for user applications. The complexity of RoT implementation often introduces vulnerabilities, such as privilege escalation and code injection risks, which affect the security of user data during execution. In this paper, we propose a RoT firmware verification framework that acts as a firmware firewall (FirmWall) to enhance the overall security of the system. Specifically, we perform directed symbolic execution focused on system calls to verify RoT firmware binaries against security specifications, facilitating targeted patching to mitigate potential vulnerabilities. Our framework demonstrated significantly better coverage compared to state-of-the-art symbolic execution. It also confirmed the presence of multiple vulnerabilities (CVEs) in recent versions of ARM Trusted Firmware-M implementations.**

I. Introduction

Trusted execution environments [1] (TEE), popularly known as security enclaves, securely manage security-sensitive operations involving user data, user code, and critical decision-making algorithms. Figure 1 shows a typical TEE implementation that utilizes a Root-of-Trust (RoT) module to provide services to the host system to handle both trusted and untrusted applications based on their security specifications. The RoT module, also known as trusted platform module, consists of both hardware and firmware layers. The firmware layer (RoT firmware) provides trusted services as system calls to cryptographic operations, including encryption, decryption, key exchange, and hashing, which are performed by the accelerators in the hardware layer (RoT hardware). The RoT firmware also supports key management, firmware authentication, secure boot, and other applications, to provide services to the host machine that executes the operating system. These services are implemented as system calls on the firmware so that the access restrictions can be applied based on the security requirements.

A. Security Concerns with RoT Firmware

Since firmware is a critical layer that interacts directly with the hardware, any vulnerabilities in firmware can be exploited by attackers to gain access to the system. These vulnerabilities can potentially bypass higher-level security measures, such as access control, secure communication mechanisms, and execution privileges of the firmware. Although Common

A. Jayasena and P. Mishra are with the Department of Computer & Information Science & Engineering, University of Florida, Gainesville, Florida.

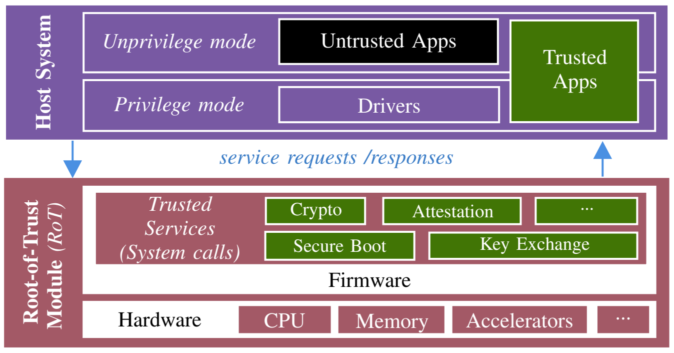


Fig. 1: Typical implementation of a root-of-trust module that serves as a co-processor to protect sensitive operations and user data. Firmware on the co-processor handles the root-of-trust configurations and manages the trusted services.

Weakness Enumeration (CWE) examples are available from MITRE [2] and NIST-NVD [3], these vulnerabilities are hard to track in the production version due to practical constraints, such as unavailability of the source code, limited visibility into the firmware's internal operations, and the complexity of the firmware's interactions with hardware components.

Most of the recent firmware-related security incidents, covering large network servers (*CVE-2024-3400, CVE-2024-22729*) as well as everyday devices such as printers and routers (*CVE-2022-30078/79, CVE-2023-43128 , CVE-2023-43204, CVE-2022-38308/535/534, CVE-2024-23625, CVE-2024-23628, CVE-2024-22529*), are results of commonly known weaknesses, such as *CWE-77* (command injection), *CWE-125* (out-of-bounds read), *CWE-787* (out-of-bounds write), *CWE-122* (heap-based buffer overflow), *CWE-306* (missing authentication for critical function), and *CWE-310* (cryptographic issues) [4]. Figure 2 illustrates that the top five CWEs that contributed to known exploited vulnerabilities (KEVs) in 2023 are all from the well-known weaknesses outlined above. This also highlights how these simple bugs can escape into production firmware and lead to security breaches.

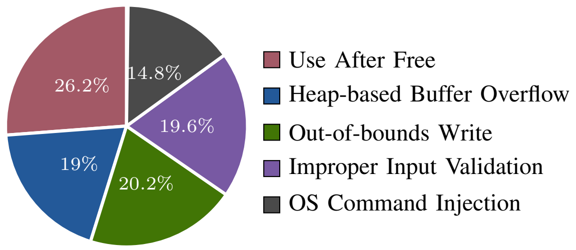


Fig. 2: Top five common weakness enumerations (CWEs) that contributed to known exploited vulnerabilities (KEV) [2].

It is easy for a programmer to pass a regular expression, for example, through a wrapper to a system call that may have higher execution privileges. This can inadvertently create security vulnerabilities if the input is not properly sanitized before reaching the system call or if the system call itself does not perform security checks. The problem arises because the system firmware that implements the system calls may assume that the input has already been sanitized by earlier stages of the firmware. This assumption is often made for the sake of performance, as system firmware needs to efficiently service multiple or even hundreds of client applications simultaneously. As firmware continues to evolve, the challenge of maintaining robust security while ensuring high performance becomes increasingly difficult, requiring more sophisticated design practices and thorough testing to mitigate risks.

B. State-of-the-Art and Its Limitations

There are promising approaches for symbolic simulation of software programs [5]–[7]. There are also prior efforts for verification of Android firmware images [8], application-specific firmware verification [9]–[11], and functional verification of hardware-firmware interactions [12], [13]. The existing approaches are not suitable for symbolic execution of modern firmware binaries to defend against unauthorized system calls due the following limitations. (1) Existing source code analysis techniques require the original source code, which necessitates tracking down the downstream repositories and verifying them individually. (2) Existing binary analysis techniques are often specific to the application scenario and designed specifically for simple microcontroller applications (such as 8-bit systems) and lack the ability to be customized for RoT firmware. (3) Unlike software entry points, which typically rely on a preconfigured operating environment, firmware must establish this environment from scratch, ensuring that all necessary hardware components are correctly configured and operational before higher-level software can execute. Section II-B describes existing efforts and highlights their fundamental limitations.

C. Our Contributions

Since vulnerable system calls are buried deep inside the firmware, traditional symbolic execution techniques struggle to progress beyond the initial firmware configuration routines, as illustrated in Figure 3a. This figure shows how conventional symbolic execution often becomes trapped in early-stage initialization paths, with no effective guidance toward deeper execution paths where system calls reside. In contrast, our proposed approach, shown in Figure 3b, introduces a directed RoT firmware validation framework, referred as *FirmWall*. It steers symbolic execution toward security-critical system call functions by defining a region of interest, enabling deeper and more targeted analysis. The symbolic execution region is selectively controlled based on the verification requirements, allowing our method to bypass irrelevant paths and focus computational resources where they matter most. *FirmWall* is intended for device vendors to evaluate their final firmware releases (tethered update or over-the-air update) for security vulnerabilities associated with system calls. In order to make it happen, this paper makes the following contributions:

- Our framework enables directed symbolic execution of firmware binaries based on the system calls to identify vulnerabilities, facilitating the implementation of necessary security patches.
- We decompose complex firmware into smaller verification scenarios based on system calls to make symbolic execution tractable. We utilize path vector routing to mitigate state explosion during symbolic execution.
- We propose a large language model (LLM)-assisted methodology to translate RoT firmware security specifications into security rules. We also introduce an automated approach for state preparation and checker integration using specifications for symbolic execution.
- Static control flow graph (CFG) is efficient for storage and analysis, but it may not be complete. While dynamic (unrolled) CFG is complete, it can introduce exponential time and storage complexity. We propose a lazy construction that combines the advantages of static and dynamic CFG construction while avoiding their pitfalls.

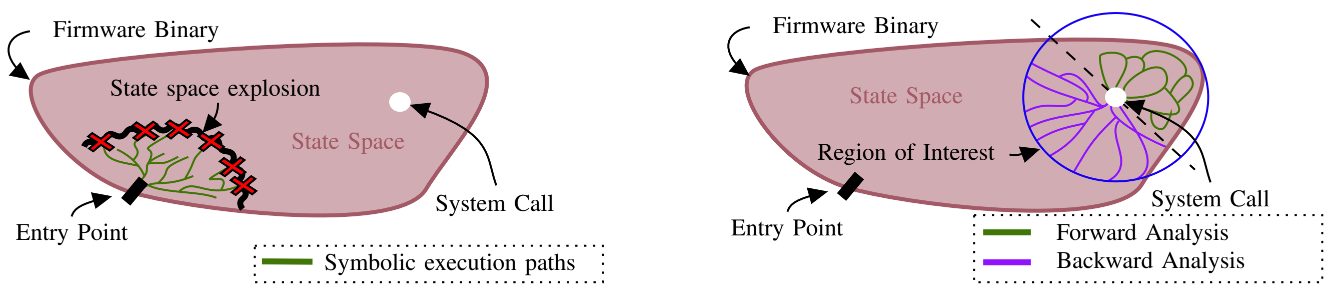
Our framework introduced key technical innovations: (1) a hybrid CFG reconstruction method that incrementally builds control flow via symbolic execution to handle indirect jumps and path-sensitive branches; (2) a rule extraction pipeline using large language models to translate high-level specifications into symbolic rules; and (3) a system call-directed execution strategy that limits analysis to semantically relevant paths, improving scalability while maintaining security coverage. Since source code is often unavailable for deployed systems, our method operates directly on binaries, making it suitable for real-world analysis. This captures compiler-induced transformations and reveals low-level hardware interactions, such as memory-mapped I/O and firmware-specific behavior, that source-level analysis typically misses.

II. Background and Related Work

Root-of-Trust (RoT) is a sub-system of intellectual property (IP) cores that consist of diverse components, including a processor core, cryptographic modules, and a secure storage, as shown in Figure 1. Host system interacts with the RoT system via service requests and responses that are implemented via shared memory. When the host wants to send data to the RoT module, it will create a service request by encoding the request into a structure similar to Figure 4, place it on the shared memory, and set the mailbox registers. A similar process is followed by the RoT module when responses are ready from the system calls. In this section, we first survey existing RoT specifications and their implementations. Next, we discuss the related efforts on firmware verification and their limitations.

A. RoT Firmware Specifications

Due to the foundational role of acting as the trust anchor, the RoT is designed to be secure and resistant to tampering, ensuring that all subsequent layers of the system can be trusted. Therefore, RoT must maintain the public key component of the vendor keys in an immutable manner so that it can authenticate future firmware updates. RoT firmware implements various



(a) Existing techniques start from the entry point, and fail before reaching the system call due to state space explosion.

(b) Proposed system call directed symbolic execution creates intermediate entry points for each system call to mitigate state explosion.

Fig. 3: Comparison between existing symbolic execution and our proposed framework. We show only one system call verification scenario in this example. in a typical RoT system, there are many such system calls associated with different services.

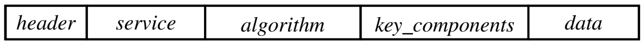


Fig. 4: Sample service request packet structure.

services (trusted code) that the host applications (untrusted code) can utilize to establish trust, as illustrated in Figure 5. We first discuss architecture-specific RoT solutions. Next, we briefly describe industrial RoT implementations.

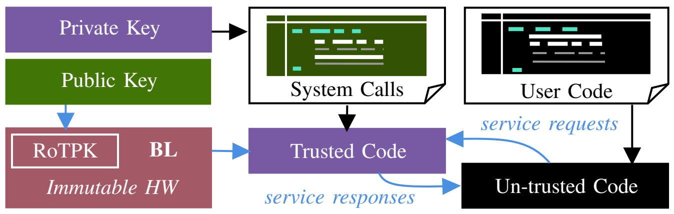


Fig. 5: Simplified illustration of a chain-of-trust.

ARM Platform Security Architecture: Trusted Firmware-M [14] is the RoT specification from ARM's Platform Security Architecture (PSA) [15] intended to be used in the ARM computing ecosystems. PSA comprises a comprehensive framework that includes threat models, security analysis methods, hardware and firmware architecture specifications, and an open-source firmware reference implementation. PSA offers a methodology based on industry best practices, enabling consistent security design at both hardware and firmware levels. This includes specifications and reference implementations for various services such as cryptographic services, secure storage services, attestation services, and platform-supporting drivers that need to be present in an RoT based on ARM architecture. The PSA cryptography API offers access to a range of cryptographic primitives and serves two primary purposes. First, it can be utilized on a PSA-compliant platform to develop services like secure boot, secure storage, and secure communication. Next, it can be used independently of other PSA components on any platform based on the requirement.

RoT Solutions for Intel Architecture: Intel offers implementations, such as Intel Boot Guard, Intel Platform Trust Technology (PTT), Intel Trusted Execution Technology (TXT), and Intel Converged Security and Management Engine (CSME) [16]. Intel Boot Guard ensures that the system boots only with BIOS firmware trusted by the platform manufacturer by verifying the initial boot block. Intel PTT provides Trusted Platform Module (TPM) functionality, securely stores keys, and performs cryptographic operations to verify boot firmware integrity. Intel TXT establishes a root of trust with hardware-based measurements, ensuring that the system firmware, operating system, and hypervisor remain untampered. Intel CSME acts as a root of trust for platform security, performing initial boot code verification and firmware integrity checks to ensure that only authenticated firmware executes during boot.

RoT Solutions for AMD Architecture: AMD offers solutions such as the AMD Secure Processor (AMD-SP), which serves as a firmware configurable hardware RoT by providing secure boot capabilities, cryptographic operations, and secure key management [17]. Additionally, AMD Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) protect memory data and virtualized environments, while AMD Firmware TPM provides TPM functionality for secure boot and platform integrity measurements.

Industrial RoT Implementations: Synopsys tRoot [18], for example, is a hardware-firmware RoT solution designed for a range of applications, including IoT and automotive systems. It provides secure boot, secure key management, and cryptographic acceleration, ensuring that only authenticated firmware is executed. The Trusted Computing Group's (TCG) TPM [19] standard offers a widely adopted RoT, providing secure storage for cryptographic keys and measurements that verify system integrity. There are also commercial implementations of hardware security modules (HSMs) that offer secure boot and cryptographic services for embedded systems. These solutions contribute to a more secure computing environment by ensuring the integrity and authenticity of firmware and protecting sensitive operations across various industries.

B. Related Work

There are promising prior efforts in verification of both application programs and compiled binaries. We survey related efforts in four major categories and discuss their limitations.

Symbolic Simulation of Software Programs: There are several software-level application binary analysis frameworks for performing symbolic execution [5]. For example, `angr` [6] is a binary analysis framework that excels in both static and dynamic analysis of executables. It supports tasks such as symbolic execution, control flow analysis, and data dependency tracking, providing various constructs for vulnerability identification and automated exploit generation. Similarly, `Manticore` [7] is a symbolic execution engine focused on binary analysis and automated vulnerability discovery, particularly in smart contracts and software binary exploitation. Although these methods are promising for verifying generic software binaries, they are not directly applicable to firmware verification due to the specialized constraints and embedded environments in which firmware operates.

Verification of Android Firmware Images: Android devices have two components to their read-only-memory firmware: Linux firmware subsystem and Android system image. An automated tool to detect privilege-escalation vulnerabilities in the pre-installed Android system images is proposed [8]. The authors focus on the functionalities provided by Android system firmware applications that can be triggered without the user's knowledge. The authors extract the system image to obtain the Dalvik executable for each of the system applications and perform taint analysis with injected manually created security rules. Although this technique is promising in detecting privilege escalations, it is only limited to Android firmware images due to the structure of the Dalvik executable.

Control Flow Recovery: Control flow recovery plays a critical role in binary analysis, especially for embedded systems. Tsang et al. propose FFXE [20], a dynamic forced execution technique to reconstruct accurate control flow graphs in firmware binaries, particularly targeting indirect branches such as callback functions in interrupt-driven systems. This dynamic approach complements existing static techniques by observing execution behavior in situ. Similarly, Nguyen et al. [21] present a hybrid method that integrates static disassembly with dynamic test case generation, allowing the recovery of indirect jump targets. Both approaches address a fundamental challenge in binary analysis—resolving dynamic control transfers that are difficult to handle through static means alone.

Vulnerability Detection in Binary Code: Recent research has advanced automated vulnerability detection through slicing and semantic analysis. Wu et al. introduce UltraVCS [22], which performs ultra-fine-grained variable-based slicing to isolate vulnerability-prone code regions. This technique significantly improves precision in identifying vulnerabilities in complex binaries. Yamaguchi et al. [23] complement this with Chucky, a tool that highlights missing sanity checks in source code—one of the most common root causes of security vulnerabilities. At the firmware level, Wen et al. propose FirmXRay [24], a static analysis tool for detecting Bluetooth link layer vulnerabilities from bare-metal binaries, illustrating the value of domain-specific techniques in uncovering deep-seated flaws in embedded systems.

Binary Rewriting and Dynamic Execution: Binary rewriting and patch generation have become increasingly feasible with hybrid lifting techniques. Reiter et al. [25] demonstrate a method to decompile and patch vulnerable binary code via partial recompilation, bridging the gap between binary-only and source-assisted mitigation. On the other hand, Pang et al. [26] explore the difficulties in establishing ground truth for binary disassembly, revealing critical issues in evaluating disassembly accuracy—especially when evaluating CFG-based tools. For improved code coverage during analysis, Peng et al. propose X-Force [27], a dynamic forced execution engine that explores multiple execution paths by manipulating conditional branches, supporting deeper inspection of binaries without requiring full program inputs.

Vulnerability Detection with Machine Learning: Machine learning-based approaches have emerged as powerful tools for vulnerability detection of software applications. Zou et al. introduce μVulDeePecker [28], which uses deep learning models to detect multiple vulnerability types in code gadgets extracted from binaries. Li et al. extend this direction with SySeVR [29], which identifies semantic vulnerability candidates and applies neural network models to classify them effectively. Furthermore, VulDeeLocator by Li et al. [30] aims for fine-grained localization of vulnerabilities within code snippets, enabling actionable outputs for developers. These approaches highlight the growing role of AI in automating binary analysis tasks that were previously manual and error-prone.

Verification of Application-Specific Firmware: FIE [9] is a firmware-focused (for MSP430 microcontrollers) source code symbolic analysis tool that uses KLEE [10] symbolic execution engine. It converts the firmware source code into LLVM bytecode instructions and performs the symbolic analysis on top of the intermediate representation. FirmUSB [11] framework analyzes USB device firmware on MCS-51 family (8051) microcontroller by extending the FIE framework. Instead of relying on the source code, the authors reconstruct the source code from the binary and perform symbolic execution using KLEE [10]. This approach can handle only a small search space: 8-bit MCS-51 instruction set with 4KB ROM and 128-byte RAM. It is not applicable for contemporary microcontrollers that typically require 32/64-bit instruction set with both RAM and ROM in the range of megabytes (MB).

Verification of Hardware-Firmware Interactions: Firmware symbolic evaluation requires hardware interpreters to simulate the firmware correctly. A hint-based symbolic execution framework using hardware description as the symbolic interpreter is proposed in [12]. The authors compile the firmware and inline it with the flattened hardware description of the SoC to achieve the objective, however, manual hint-based guidance is required for the symbolic execution tool to avoid state space explosion. *HIVE* [13] extends this idea for automated hint extraction by utilizing a test plan to automatically derive the functional verification scenarios so that the constraint solver is focused on one verification scenario at a time. These approaches have two major limitations: (i) they require the hardware description of the implementation, and (ii) they are suitable for only functional verification.

Limitations of Existing Methods: Although the above methods are promising for verifying generic software binaries as well as specific firmware implementations, several limiting factors make them unsuitable for RoT firmware verification. (1) Existing source code analysis techniques require

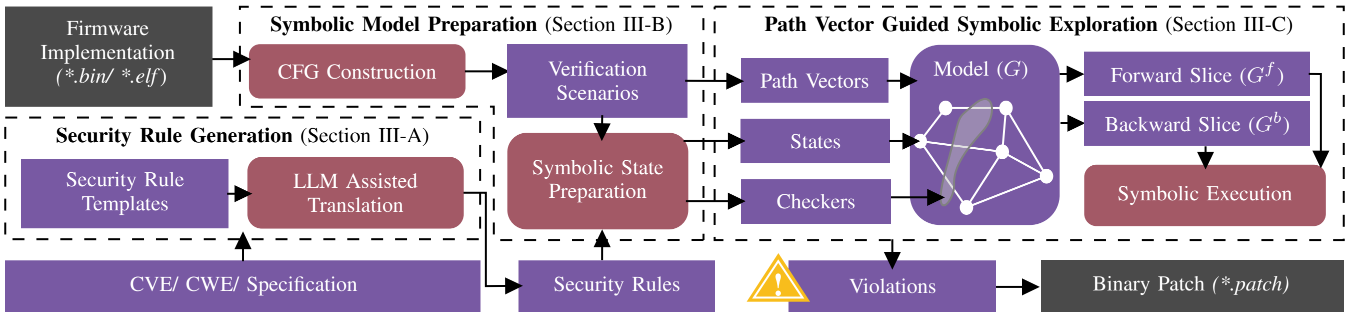


Fig. 6: Overview of the *FirmWall* symbolic execution framework. It contains three primary stages: i) specification parsing and LLM-assisted security rule translation, ii) verification scenario-based symbolic model preparation, and iii) path vector-guided symbolic execution. Once violations are detected, patches should be generated to defend the firmware against vulnerabilities.

the original source code, which necessitates tracking down the downstream repositories and verifying them individually. However, integration time changes and patches cannot be monitored. (2) Existing binary analysis techniques are often specific to the application scenario and designed specifically for simple microcontroller applications (such as 8-bit systems) and lack the ability to be customized for RoT firmware. Specifically, RoT firmware binary entry points are far more complex compared to software entry points due to the need to manage various hardware configuration routines, including initializing low-level peripherals, setting up memory controllers, configuring clock settings, and performing hardware-specific tasks such as power management and security checks. (3) Unlike software entry points, which typically rely on a pre-configured operating environment, firmware must establish this environment from scratch, ensuring that all necessary hardware components are correctly configured and operational before higher-level software can execute. This complexity is further aggravated by the need to handle platform-specific variations and ensure compatibility across different hardware configurations. As an example, prior hybrid CFG construction approaches are not suitable for obfuscated or interrupt-driven firmware since they are prone to path explosion due to the lack of context sensitivity. These limitations highlight the need for directed symbolic execution [31] for RoT firmware validation.

III. FirmWall Methodology

Figure 6 illustrates the primary stages of the proposed *FirmWall* (firmware firewall) framework that performs system call directed symbolic execution to find firmware vulnerabilities: i) security rule translation from natural language specifications and vulnerability databases, ii) verification scenario-based symbolic model preparation, iii) path vector-guided symbolic exploration. Once our framework identifies security rule violations, they can be used to facilitate the construction of binary patches so that the final firmware is mitigated from potential vulnerabilities. Algorithm 1 shows the three major steps of our framework, where R contains the generated rules from the specification, and S is the symbolic model. The algorithm return the violations identified during the symbolic execution. The remainder of this section describes these steps. To facilitate the discussion, we define the following two terms:

Region of Interest: It refers to a specific region of the firmware that is crucial for the verification context (e.g., activation of a system call). *A region-of-interest should cover from the service listener interrupt to the system call* .

Reversed Graph ($R(G)$): Given a directed graph such that $G = (V, E)$, corresponding reversed graph $R(G)$ can be obtained by altering the directions of each edge $e \in E$.

A. Security Rule Generation

As discussed in Section I-A, most firmware vulnerabilities arise from common implementation mistakes and oversights in the firmware source code. These checks cannot be directly applied to source code (in *C* or *C++*) or compiled binaries. The main reason is that the expected conditions for monitoring firmware must be translated into the implementation context, which requires specification details. This process is tedious and manual. Instead of writing manual security rules, we can create security rule templates and take assistance from large language models (LLM) and natural language translation techniques to translate them into design-specific checks. Note that LLMs are used solely to significantly reduce the manual effort involved in generating these security rules.

Algorithm 1 Overview of FirmWall

```
Require: Binary $B$, specification $D$
Ensure: Violations $V$
1: function FIRMWALL($B, D$)
2:     $R$ = TRANSLATE_RULES(D)
3:     $S$ = SYMBOLIC_MODEL($B$, $R$)
4:     $V$ = SYMBOLIC_EXECUTION($B$, $S$)
5:     return $V$
```

Algorithm 2 shows the main steps of security rule generation. We first outline different security rule templates that can be derived from specification sources. Next, we describe how LLM utilizes these templates to derive security rules.

1) Security Rule Templates: Security rules can be divided into two main categories: desired rules representing expected behaviors and undesired security rules representing unexpected scenarios. Desired security rules define the properties, attributes, or conditions that a secure system must exhibit. Undesired security rules identify the properties, attributes, or conditions that a secure system should not exhibit, highlighting

Algorithm 2 Security Rule Generation

```
Require: Specification $D$
Ensure: Security Rules $R$
1: function TRANSLATE_RULES($D$)
2:     $T$ = security rule templates
3:     $R$ = LLM.QUERY($T$, $D$)
4:     return $R$
```

the weaknesses or flaws that should be absent. In the context of RoT, security rules depend on both the functional and security specifications. Specifically, we must consider various factors, such as the types of system call services provided by the RoT firmware, their capabilities and limitations, the available privilege levels for applications, and how memory is managed during a system call. We have identified the following security rule categories.

Algorithmic Constraint Rules: Cryptographic operations have inherent limitations on features, such as symmetric key sizes, public and private key component lengths, point validity, and data length. These properties are detailed in cryptographic standard documentation and can be translated into security rules to be checked during symbolic execution.

Example 1 (Algorithmic Constraints): *Advanced Encryption Standard (AES) has a supported key length of 128-bits and 256-bits. Similarly, RSA public key encryption key components of n, d, e should be in the range of 2048-bits to 4096-bits and the message length should be less than or equal to the key component size length.* ■

Physical Constraint Rules: Hardware consists of physical limitations due to area, power, and performance constraints. With physical constraint rules, we try to capture information such as architecture register size (16/32/64 bits), communication bus width, and physical accelerator implementation-related constraints into properties that can be translated into security rules to be checked during symbolic execution.

Example 2 (Physical Constraint Rules): *In an ARM Cortex-A53 SoC, this type of rules can be constructed using constraints such as register width of 64 bits, and AMBA AXI (Advanced Extensible Interface) with a 128-bit bus width. In addition, the firmware must account for wait delays when fetching data from the bus, as different components like the memory and peripherals may operate at different speeds.* ■

Security Architecture Rules: These rules are usually available in security specifications and are specific to the RoT firmware implementation. These rules describe privilege levels, outlining the kind of privileges allowed with different system calls, cryptographic implementation-related details, key storage, attestation-related details, and trusted and untrusted memory regions.

Example 3 (Security Architecture Rules): *In ARM PSA there are definitions such as access control and privilege levels. Cryptographic services within a RoT module should not be accessible by any user with any privilege. Instead, access should be restricted to specific, authorized entities. Only processes or users with appropriate privilege levels, such as system or root-level access, should be able to interact directly with the cryptographic services. Regular user applications should have access only through controlled interfaces.* ■

Memory-related Security Rules: Most of the memory-related security properties are related to common weakness enumerations and design-specific rules that can be found in the security specification. These rules implement memory safety checks that must be maintained by preventing buffer overflows, use-after-free errors, double-free errors, and null dereferences. Pointers must be valid, meaning they should be properly initialized and any pointer arithmetic should not lead to invalid memory access. Memory leaks need to be avoided by ensuring all allocated memory is appropriately freed. Data integrity should be preserved by correctly handling data within memory arrays and ensuring proper initialization. Control flow integrity is essential, particularly in the safe handling of function pointers and indirect jumps or calls. In some cases, security architecture specifications may outline restrictions on the heap to store data or force to use the memory stack with predefined buffer limits.

Example 4 (Memory-related Security Rules): *ARM PSA recommends implementing bounds checking to protect against buffer overflows, which can lead to security vulnerabilities. It is recommended to carefully manage heap allocations in secure contexts to avoid unintended data leakage or vulnerabilities. For example, sensitive data should not be stored in dynamically allocated memory if it can be avoided, as heap memory may be more prone to certain types of attacks.* ■

2) LLM-Assisted Security Rule Translation: Specifications are typically described in natural language such that they can be understood by developers and engineers. Therefore, all of the above constraints can be manually constructed into the security rule format. However, large language models (LLMs) [32] are competent for parsing descriptions in natural language into properties when prompted with predefined templates. For instance, transformer-based models [33], such as text-to-text and sequence-to-sequence models, are effective for parsing templates and generating security rules. Although the generated security rules are accurate in most of the cases, they need to be manually validated before proceeding to the next step. In this section, we discuss the specific steps of handling custom data types and function specifications with security rule templates so that LLMs can translate the specifications to security rules.

```
-structure:
  name: [[struct name]]
  num_items: [[no. of members]]
  size: [[size in bytes]]
  item_i: [[Member i info]]
    name: [[name]]
    type: [[type]]
    size: [[size]]
  ...
```

(a) Data structure template

```
- structure:
    name: psa_invec
    num_items: 2
    item_1:
      name: base
      type: ptr
      size: 32
    item_2:
      name: len
      type: size_t
      size: 32
```

(b) LLM translated security rule

Fig. 7: Sample data structure template (a) is given to the LLM with the security specification document to produce the security rule in *YAML* format (b) for `psa_invec` structure.

```
-function:
  name: [[function name]]
  num_args: [[no. of args]]
  - arg_i: [[arg i info]]
     name: [[name]]
     type: [[type]]
     size: [[unit size]]
     constraints:
     - min: [[min val]]
     - max: [[max val]]
     - not: [[!= val]]
     - eq: [[= val]]
     - precondition : [[ var, value ]]
  num_vars: [[no. of vars]]
  - var_i:  [[var i info]]
     name: [[name]]
     ...
  output: [[output info]]
    type: [[type]]
    size: [[unit size]]
```

(a) Function specification

```
- function:
  name: psa_hash_compare
  num_args: 5
  - arg_1:
      name: alg
      type: psa_algorithm_t
      size: 32
      constraints:
        - min: 1
        - max: 0xFFFFFFFF
        - not: 0
  ...
  num_vars: 0
  output:
    type: psa_status_t
    size: 32
    constraints:
      ...
```

(b) Translated security rules

Fig. 8: Simplified function-level security specification template and security rules translated from the specification of the `psa_hash_compare()` function in *YAML* format.

Handling Custom Data Types: Security rules must be implemented as symbolic variables before integrating them as checks inside our symbolic model. Although generic data types such as 8/16/32/64-bit integers, floating-point numbers, and pointers can be represented directly as symbolic values, custom data structures that are typically used in firmware cannot be directly represented. However, these custom constructs are created using multiple generic data types. To process such data structures, we implement separate specification templates. This describes the types of data available in each data structure and their constraints. Figure 7a shows an example template for translating the data structure specifications using LLMs to produce the security rule shown in Figure 7b.

Example 5 (Data Structure Template): *In TrustedFirmware-M, the* `psa_invec` *struct is a custom data structure that uses generic data types. This structure describes an input vector for inter-process communication calls. The base member is a pointer to the start of the input data, while the* `len` *member specifies the size of this data in bytes. Figure 7b illustrates the security rule translated from the specification into the YAML format utilizing the template shown in Figure 7a.* ■

Security Rule Translation: Once custom data structures and security templates (discussed in Section III-A1) are available, LLM can translate them to security rules. Functional inputs and output types can now be selected from both generic and custom data types, as the composition of the custom data types has been captured by the data structure specifications. Figure 8a illustrates the template for constructing security rules using the function-level specification. Conditions for variables and arguments used in functions, such as min and max ranges, can be specified in the constraints section under the corresponding variable. This format captures the precondition for different actions that are required to interpret security rules with conditional constraints.

Example 6 (Security Rule Template): *In TrustedFirmware-M, the function* `psa_hash_compare()` *contains five arguments including two constant byte arrays and their length. Figure 8b illustrates the security rule translated from the specification into the YAML format. Note that LLM-assisted translation requires manual inspection due to potential hallucinations from different LLM configurations.* ■

B. Symbolic Model Preparation

Once we have the security specification converted into security rules, we need to prepare the symbolic execution model. First, we construct the control flow graph (CFG) of the entire firmware. A CFG can represent the possible execution paths through an application. Each node in the graph corresponds to a basic block, which is a straight-line sequence of instructions with no branches except at the entry and exit points. The directed edges between nodes represent the control flow transitions between these basic blocks. There are two approaches for constructing the CFG from the firmware: static analysis, which involves examining the code without executing it, and dynamic analysis, which involves executing the code and observing the actual control flow during runtime. Algorithm 3 shows four key steps: (1) constructing the control flow graph (CFG) of the firmware, (2) generating a symbolic model for each system call, (3) appending security rules as function hooks, and (4) preparing symbolic execution states for each system call. The remainder of this section describes these steps in detail.

Algorithm 3 Symbolic Model Preparation

```
Require: Firmware binary $B$, Security rules $R$
Ensure: Region-of-Interest subgraphs $S$
1: function SYMBOLIC_MODEL($B, S$)
2:     $CFG$ = LAZY_CONSTRUCTION($B$)
3:     $S = \{\}$
4:     for system call $i \leq I$ do
5:         $\{G_i^f, G_i^b\}$= SPLIT($CFG.copy(), u_i$)
6:         $hooks_i \leftarrow$ GENERATEHOOKS(R)
7:         $states_i \leftarrow$ PREPARESTATE(R)
8:         $S$.append( $\{G_i^f, R(G_i^b)\}$, $hooks_i$, $states_i$)
9:     return $S$
```

1) Lazy Construction of Control Flow Graph : Static analysis does not work effectively for building a CFG for RoT firmware due to several inherent limitations. RoT firmware often contains optimized and obfuscated code to enhance security, making it difficult for static analysis to accurately interpret the control flow. Static analysis also lacks path sensitivity and context sensitivity, meaning it cannot distinguish between different execution paths or track function calls and returns in specific contexts, which are common in firmware implementations to prevent tampering and enhance security. On the other hand, dynamic CFG construction based on symbolic execution is not a viable solution due to the state space explosion as illustrated by Figure 9.

FirmWall adopts a lazy construction of CFG to leverage the advantages of both static and dynamic analysis while mitigating their limitations. Specifically, lazy construction enables symbolic execution to only explore possible paths from an execution point at a time, which does not maintain the state space

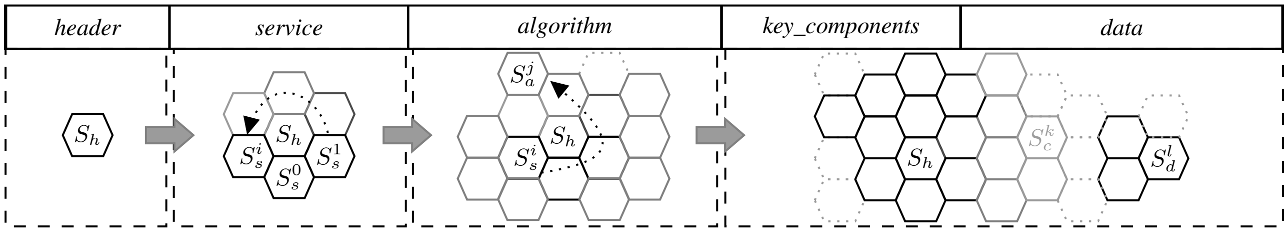


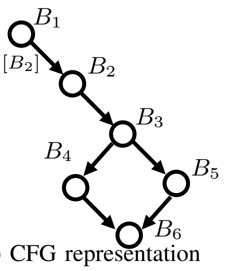
Fig. 9: Illustration of the growth of state space based on a request packet from the host system to RoT via shared memory. Each parameter in the packet contributes to the increase in the state space. Note that this is an example of a single request. In reality, the RoT firmware kernel runs several threads serving multiple host requests at a time, which can lead to state explosion.

throughout the execution. This creates a complete CFG for the subsequent steps. For this, we start by loading the binary and finding the entry point to the binary to start the symbolic execution, interpreting each instruction and maintaining a symbolic state that tracks register and memory values. When it encounters branches, it explores all feasible paths by forking the state, thus capturing both direct and indirect jumps and calls. This allows it to handle indirect control flows and dynamic behaviors accurately compared to static analysis. This way we can build the CFG incrementally, creating nodes for basic blocks and edges for control transitions based on actual execution paths. This process includes static analysis techniques for function detection and loop handling, while dynamic analysis techniques resolve control flows that depend on runtime values. Additionally, we incorporate context sensitivity by tracking function call and return contexts, ensuring an accurate mapping of inter-procedural flows.

We represent CFG as a directed graph $G = (V, E)$, where V is the set of vertices, each corresponding to a basic block in the original firmware binary, and E is the set of directed edges, each representing a control flow transfer between blocks (e.g., jumps, conditional branches, return statements). For example, if a basic block $B_1 \in V$ conditionally jumps to another basic block $B_2 \in V$, a directed edge $(B_1 \rightarrow B_2) \in E$ is created in the graph, where $\rightarrow$ corresponds to the direction of the jump. This graph structure allows verification scenario identification, efficient traversal, and analysis of the firmware control flow.

```
#include <stdio.h>
int write_reg(int x) { ← [B1]
 volatile int *data_reg =
            (int *)0x40000000; ← [B2]
 if (x == 0) { ← [B3]
  *data_reg = 1; ← [B4]
 } else {
  *data_reg = 0; ← [B5]
 }
 return 0; ← [B6]
}
```

(a) Simple function



(b) CFG representation

Fig. 10: An illustrative example of a firmware function converted to the CFG representation in the form of a directed graph using lazy construction of CFG.

Example 7 (Firmware CFG Construction): *Figure 10a illustrates a typical function, written in **C** language, from a firmware implementation. Figure 10b illustrates the corresponding CFG representation. Note that the branch B_3 creates two edges based on the two possible paths in the code.* ■

2) Generating Verification Scenarios: Symbolic execution of software implementations typically begins from the entry point of the program. As discussed in Section II-B, if we start the firmware analysis from the entry point, the underlying SMT solver will get stuck and eventually fail due to state space explosion. Figure 9 illustrates a sample request message from the host machine to RoT via the shared memory. Each parameter in the request message increases the state space with respect to its parameter width. Note that the RoT firmware kernel typically handles multiple such requests concurrently, and this increases the state space further. The number of target states increases exponentially with the increase of the width of these parameters. In other words, the increase in parameter width increases the possible paths that need to be explored symbolically, eventually making it an intractable problem for the symbolic simulator [13]. Instead of starting from the entry point, we target one system call at a time. In this section, we focus on creating region-of-interest subgraphs of the CFG based on the system calls, which are treated as verification scenarios for the RoT firmware.

In order to generate verification scenarios, we need to identify the system calls in the CFG. To achieve this, we first extract the symbols available in the compiled binary, which include function names and their corresponding addresses (basic blocks are within address offsets of the functions). Next, we locate the corresponding address on the CFG. Note that the system call may appear in multiple places within the CFG due to authorization mechanisms and the context sensitivity of the implementation. Next, we take multiple CFG copies such that each of them corresponds to a different system call. Let's assume that in our RoT implementation we have a total of I system calls. We iterate through each system call i such that $1 \leq i \leq I$ and split the corresponding CFG at the vertex (u_i) that contains the system call function, as illustrated in Figure 11. This creates two slices of subgraphs, forward slice (G_i^f) and backward slice (G_i^b), for each system call i. Next, we construct the reversed graph of the backward slice (G_i^b) to obtain $R(G_i^b)$ so that we are able to backtrack the stack call sequence that leads to the system call. This process is repeated for all the system calls to obtain the final graph set S as illustrated in Equation 1. Set S serves as the starting point for constructing the path vectors.

$$\forall (i \in I) : (G_i \text{ split at } u_i) \Rightarrow G_i^f, G_i^b$$
$$S = \bigcup_{i=1}^{I} (G_i^f \cup R(G_i^b)) \tag{1}$$

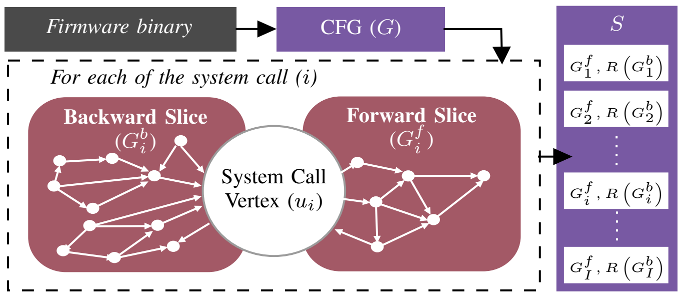


Fig. 11: Illustration of verification scenario generation for each system call in the firmware by splitting the CFG (G) at the corresponding vertex (u_i), resulting in two graph slices, G_i^f and G_i^b. The slices $\{G_i^f,\ R(G_i^b)\}$ are added to the set S.

3) Security Monitors via Function Hooks: In order to monitor the security rules during symbolic execution, the security rules need to be incorporated into the binary implementation. However, traditional assertion-based monitoring is not an option since the execution is performed at the binary level. Instead, we generate function hooks and attach them to the symbolic simulator to perform the security rule checks. We consider two types of checkers for RoT firmware: i) checkers that were defined under the constraints section of each variable and function argument of the security rules, and ii) checkers related to dynamic memory-related components, where memory allocations, accesses, resizes, and de-allocations are monitored. While the first type of checkers are generated by LLMs, the second type of checkers are manually created due to their generic applicability across diverse firmware implementations. Security rule-related checkers get attached to their corresponding basic block address while memory-related checkers are attached to the function that are responsible for memory operations. During symbolic execution, a function hook will trigger a warning if the check is not satisfied when a particular address is in the execution state of the execution manager, as illustrated in Figure 12.

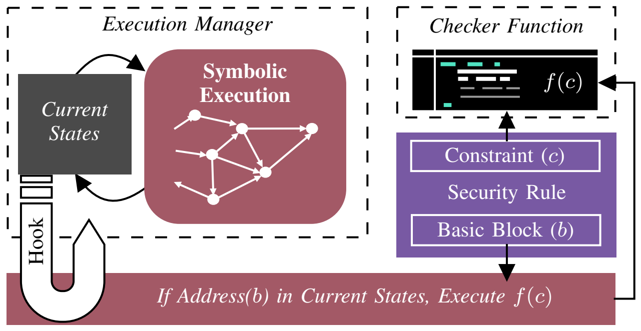


Fig. 12: An overview of how a checker is implemented from the security rule and corresponding function hook that attaches the checker to the execution manager.

4) Symbolic State Preparation: As discussed in Section II-B, a major limitation of applying software-based symbolic execution for firmware verification is that the complex procedures in the firmware implementation can lead to state space explosion. *FirmWall* mitigates this problem by starting the execution from a region of interest rather than starting the execution from the entry point of the firmware. However, in order to start the execution from a region of interest, we need to prepare an internal entry point with a state preparation step. State preparation involves setting up the initial conditions for a firmware execution in a controlled analysis environment. State preparation contains two main steps: i) memory-related states, and ii) function argument-related states. If any memory related data structures are required, they need to be initialized as symbolic variables. Argument registers are then initialized with their corresponding symbolic values, often corresponding to the firmware startup state or based on the specific system call that is being analyzed. For each of the system calls, this process is repeated so that during the symbolic execution, it can start from these states to perform security checks.

C. Path Vector Guided Symbolic Execution

In this section, we discuss how to assemble all the components together to perform the analysis. We also need to define the region of interest surrounding the system call. This determines both the starting state for the analysis and the region boundaries on which the symbolic execution should be focused. To achieve both of these objectives, we implement path vector-guided symbolic execution. Path vector routing is a network routing protocol used to determine the best path for data to travel across a network [34]. It is primarily used in border gateway protocol, which is the protocol responsible for routing data between autonomous systems (AS) on the Internet. The key idea behind path vector routing is to maintain a path (a sequence of nodes) that data packets should follow to reach a destination. We borrow the idea from the path vector routing to our *FirmWall* framework to create a region of interest and avoid the symbolic execution effort so that it reduces the state space using the following techniques.

Path Tracking: Just as in path vector routing, where the entire path to a destination is recorded, in symbolic execution, we maintain a sequence of execution states or branches taken to reach a particular point in the program. This helps the analysis of the backward slice where we need to lead the symbolic analysis towards the system call.

Path Concretization and Abstraction: With this technique, we apply constraints based on the current state of the symbolic simulator and the path vector. For example, paths that fall outside the region of interest will be abstracted out since they are less relevant for security analysis so that we can prioritize the paths that are directed toward the system calls.

Specification-based Optimization: By analyzing the security specification, we can optimize the exploration strategy for different areas in the firmware. For example, if certain paths are known to lead to errors or vulnerabilities, the symbolic execution engine can prioritize these paths by abstracting the other paths to find bugs more quickly.

The remainder of this section describes three important steps: finding region of interest, finding path vectors, and symbolic execution of the RoT firmware.

1) Region-of-Interest Construction: In order to find the region of interest in the firmware, we implement a technique known as N-hop destination extraction. These hops are computed considering the system call as the hub, and hop count (n) is a parameter that can be adjusted based on the verification requirements. To implement N-hop destination extraction, we use the region-of-interest subgraphs S (discussed in Section III-B2) to obtain the possible destination functions that can be reached from each system call by just traversing n edges (hops from the system call).

In order to extract the destinations corresponding to each system call i, we first iterate over each directed subgraph from $S_i \in S$ (Section III-B2) and construct the corresponding adjacency matrix A_i. Let's assume that the subgraph vertex related to the system call i in graph S_i is u_i. Then we can compute the set of destinations (N_i) for each system call that can be reached with $n{-}hops$ using n^{th} *power of the adjacency matrix* theorem [35] as illustrated in Equation 2.

$$N_i = \forall v_j \text{ such that } (A_i^n[u_i][v_j] > 0) \tag{2}$$

If the condition $A_i^n[u_i][v_j] > 0$ is satisfied for a vertex in G_i, they are considered as the destination functions. Since the backward slice in S_i is a reversed graph $(R(G_i^b))$, this step computes the corresponding source vertices for the backward slice instead of the destinations.

2) Path Vector Extraction: At this stage, we have a set of subgraphs S and a set of $n - hop$ reachable destinations from the system calls for each subgraph for both the forward slice and the backward slice. Next, we have to extract the possible paths for each subgraph from the system call to the identified destination point (e.g., service request listener). Our final goal is to guide the underlying SMT solver of the symbolic execution and just focus on one of these paths at a time. The path leading from each of the sources u_i to $n_i \in N_i$ can be constructed using depth-first search on top of the adjacency matrix of the graph. Since the vertices contain information about the address corresponding to different basic blocks in the firmware, this serves as the path vector for the symbolic simulation to perform path prioritization during symbolic execution using a path pruning subroutine.

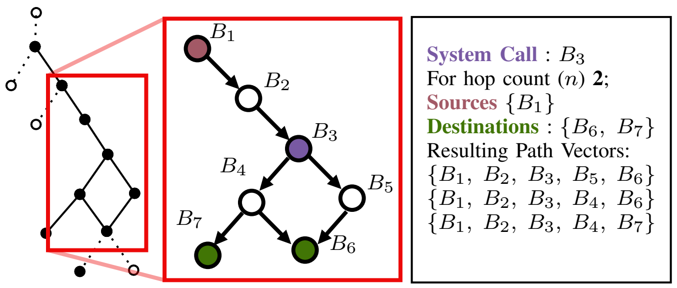

B_1
B_2
B_3
B_4
B_5
B_6
B_7
System Call : B_3
For hop count (n) **2**;
Sources $\{B_1\}$
Destinations : $\{B_6,\ B_7\}$
Resulting Path Vectors:
$\{B_1,\ B_2,\ B_3,\ B_5,\ B_6\}$
$\{B_1,\ B_2,\ B_3,\ B_4,\ B_6\}$
$\{B_1,\ B_2,\ B_3,\ B_4,\ B_7\}$

Fig. 13: Illustration of constructing the path vectors for the system call in the vertex B_3. With the hop count (n) 2, we get the source values to start the symbolic execution at the vertex B_1 from the backward slice. Similarly, from the forward slice, we have two vertices (B_6 and B_7) as the destination points. This results in three paths for the analysis as illustrated.

Example 8 (Path Vetor Extraction): *Figure 13 illustrates a simple example for the construction of path vectors based on the backward slice and the forward slice for the system call available in vertex B_3. In this example, we have the hop count (n) set to 2. Note that B_3 is deep inside the firmware binary. In order to construct the path vectors, first, we backtrack the vertex B_3, which provides the vertex B_1 as our source vertex. Similarly, from the forward slice we get two destination points of B_6 and B_7. Based on this we can construct the three possible path vectors, $\{B_1,\ B_2,\ B_3,\ B_5,\ B_6\}$, $\{B_1,\ B_2,\ B_3,\ B_4,\ B_6\}$, and $\{B_1,\ B_2,\ B_3,\ B_4,\ B_7\}$ to focus guide our symbolic analysis.* ■

3) Symbolic Exploration with Execution Manager: The previous sections prepared the three important components for symbolic execution: path vectors, symbolic states, and checker hooks. The goal of the final stage is to assemble all three components to perform symbolic execution of the RoT firmware without encountering state space explosion. In order to perform this, we create a module known as an execution manager that communicates with the underlying constraint solver. The execution manager is responsible for performing the symbolic execution while incorporating the path vectors, symbolic states, and checker hooks into the model, as illustrated in Algorithm 4. First, we initialize the symbolic execution manager by loading the firmware binary. Then for each of the subgraph pair $S_i \in S$ (the forward slice and the backward slice corresponding to each of the system call functions), we prepare and initialize the states separately. In case of the forward slice (G_i^f), we start the symbolic execution from the system call (vertex u_i) and set the target state as different destinations (set of N). In case of the backward slice (G_i^b), we start the symbolic execution at the destinations of the reversed graph (set of N) and set the target state as the system call function (vertex u_i).

We initialize the checker function hooks into the symbolic execution manager. Next, we attach the path vector to the execution manager so that it can cross-validate the address of the current basic block against the path vector, as illustrated by the function EXPLORE in Algorithm 4. In case the current state is not present in the path vector, it executes the path pruning procedure through path concretization and abstraction to reduce the effect of exploring paths that fall outside our region of interest. In other words, instead of treating path variables that do not lead to the system call as symbolic values, they are fixed with concrete values or will be given less priority with abstraction, which reduces the state space.

The symbolic execution progresses by following the program's control flow and exploring possible execution paths focused within the region-of-interest. At each step, the symbolic execution engine evaluates conditions and branches, evaluating checkers that implement security rules. The exploration continues until the destination state is reached. Throughout this process, the execution manager tracks and records the violations of security rules and memory operations, enabling the verification of properties or discovery of potential issues within the RoT firmware binary.

D. Vulnerability Mitigation

During symbolic execution, checker hooks are attached to the corresponding addresses of the firmware. These hooks create interrupts if any of the security rules are not satisfied.

Algorithm 4 Path Vector Guided Symbolic Execution

```
Require: Binary $B$, Region-of-Interest subgraphs $S$
Ensure: Violations $V$
 1: function SYMBOLIC_EXECUTION($B, S$)
 2:     EM ← $load(b)$                          ▷ Initialize the Execution manager
 3:     $V$ = {}
 4:     for each $S_i \in S$ do
 5:         if $G_i^f$ then                          ▷ handle forward slice
 6:             EM.init(state($S_i.u_i$))
 7:             EM.dest(state($S_i.N$))
 8:             EM.hook(checkers($S_i.checkers$))
 9:             EM.EXPLORE(($S_i.pathvector$, $V$)))
10:         if $R(G_i^b)$ then                       ▷ handle backward slice
11:             for each $n_i \in S_i.N$ do
12:                 EM.init(state($n_i$))
13:                 EM.dest(state($S_i.u_i$))
14:                 EM.hook(checkers($S_i.checkers$))
15:                 EM.EXPLORE(($S_i.pathvector$, $V$)))
16:     return $V$
17: function EXPLORE($P, V$)
18:     while addr == dest do
19:         if addr $\in P$ then
20:             addr = EXPLORE_PATH(addr)
21:             if (!$checker$) then
22:                 $V$.append($checker$)
23:         else                                 ▷ path vector guiding subroutine
24:             PRUNE_PATH(addr)
```

The execution manager records these failed checkers so that the appropriate corrections can be made by the user of the framework. Typically argument and variable-related hooks can be automatically patched by generating necessary assembly codes to implement the missing checks as patches. However, failed memory-related checkers can only point to the corresponding addresses of the firmware and require manual inspection to fix the issue. After fixing, the new changes can be constructed as a patch that can be applied to the production version of the firmware. Note that this process should be done prior to signing the firmware images for authentication.

IV. Experiments

In this section, we evaluate the performance and effectiveness of *FirmWall*. First, we outline the experimental setup. Next, we discuss the symbolic execution coverage achieved by *FirmWall*. Then, we evaluate the quality of LLM-assisted security rule translation. Finally, we discuss the security violations found in the benchmarks and relate them with common vulnerability exposures.

A. Experimental Setup

We have created automation scripts for the *FirmWall* using *GNU Make, bash*, and *Python* to streamline the process. For disassembling the binary, creating the CFG, and address decoding, we use `angr` [6] binary analysis framework with *cle* plugin to add the capability to decode ARM instructions. We used *VEX* [36] intermediate representation to represent the loaded binary as an executable to perform symbolic execution. To solve the constraints for the symbolic execution and CFG construction, we have used *Z3* [37] SMT solver. In order to connect *Z3* with `angr` front-end, we have used the *claripy* abstraction layer. All the experiments were performed on a machine with x86_64 Intel i7-9700 @ 3.00GHz CPU with 32GiB of memory and with an NVidia RTX 3090 GPU running GNU Linux 22.04.

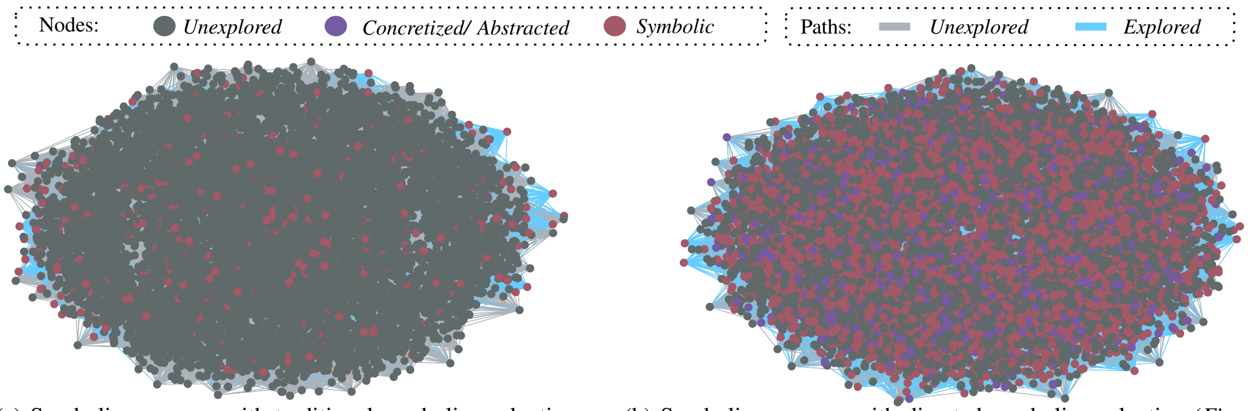
TABLE I: Four configurations of RoT firmware implementations created based on the ARM platform security architecture (PSA) reference benchmarks available in Zephyr OS [38].

Configuration		**C1**	**C2**	**C3**	**C4**
Services	*psa_crypto (+mbedTLS)*	✓	✓	✓	✓
	psa_protected_storage	✓	✓	✓	✓
	tfm_ipc	✓	✓	✓	✓
	tfm_secure_partition	✓	✓	✓	✓
mbedTLS Version		3.0	3.4	3.5	3.6
Size (Mb) of Firmware		2.0	2.1	2.1	2.2
No of Vertices (V) in CFG		13898	14224	15761	16176
No of System Calls (I)		127	143	157	157
Lines of Code (Source code)		137K	143K	155K	163K

We construct a RoT firmware benchmark using publicly available libraries. Specifically, we have selected Zephyr OS *tfm-integration* [38] modules that implement firmware for services specified by ARM platform security architecture (PSA). This provides multiple trusted services that implement different system calls from *psa_crypto, psa_protected_storage, tfm_ipc* and *tfm_secure_partition*. We have created four configurations (**C1** - **C4**, as illustrated in Table I) of RoT firmware implementation using different variations of the above four implementations. For computing the N value for the $N-Hop$ destination extraction, we have used the longest path length ($\approx$ 65) from system call functions to the service request listener function. We have selected the system calls from the algorithmic level interfaces of each of the trusted services. For example, in the case of *psa_crypto*, we have selected *psa_hash_compute, psa_cipher_update, psa_cipher_encrypt, psa_asymmetric_encrypt, psa_rsa_verify_hash,* etc. as the system calls. These system calls are managed through Zephyr OS [39] kernel and request response packet structure was created using ZCBOR [40] library. The final binary versions of these four variations are capable of running on microcontroller families based on the ARM Cortex-M platform ranging from microcontrollers *STMX*, *nRFXX*, *NXP LPX*, and the *QEMU* simulator that is implemented based on Cortex-M. Although it is possible to implement RoT firmware with either firmware and hardware-based accelerated cryptographic libraries, we used a firmware-based cryptographic library, *mbedTLS* [41].

B. Symbolic Execution Coverage

We designed this experiment to visually illustrate the symbolic execution coverage achieved by *FirmWall* compared to traditional symbolic evaluations that are designed for software binaries. For this purpose, we selected the CFG of the firmware configuration **C1** and placed a tag when explored by the execution manager. Figure 14 presents the symbolic execution coverage of the CFG for the RoT firmware configuration **C1**. Figure 14a illustrates the coverage achieved using traditional symbolic evaluation that starts the symbolic execution at the firmware entry point while Figure 14b illustrates the system call directed symbolic execution performed by our *FirmWall*



(a) Symbolic coverage with traditional symbolic evaluation (b) Symbolic coverage with directed symbolic evaluation (*FirmWall*)

Fig. 14: Symbolic execution basic block coverage for RoT firmware configuration **C1**. Directed symbolic execution (*FirmWall*) provides significantly higher coverage compared to state-of-the-art symbolic simulation.

framework. The basic blocks that were explored symbolically are represented by (●), blocks that were abstracted out or concretized are illustrated in (●), while basic blocks that were not explored are represented as (●). Corresponding transitions for explored and unexplored paths are illustrated as (—) and (—), respectively. Table II compares basic block coverage between traditional symbolic execution and our proposed framework for four RoT firmware configurations. While the traditional approach can provide only 23-30% coverage, our framework can achieve 65-78% coverage. It can be observed that compared to traditional symbolic execution, directed symbolic execution can achieve significantly better coverage with focused attention on security-sensitive regions.

TABLE II: Comparison of basic block coverage between traditional symbolic execution and our framework.

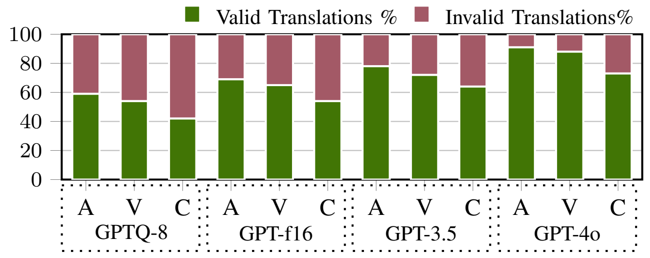
	Basic Block Coverage			
Configurations	**C1**	**C2**	**C3**	**C4**
Traditional Symbolic Execution	30	28	24	23
Proposed Framework (*FirmWall*)	73	78	69	65

C. *Quality of LLM Translated Security Rules*

This experiment evaluates the effectiveness of popular transformer-based LLM for security rule translation. For this purpose, we have obtained the ARM PSA specifications and mbedTLS specifications for the related implementation components. Figure 15 presents these results for two variants of offline *LocalGPT* [42] (GPTQ-8, GPT-f16) model and two variants of online *OpenAI ChatGPT* [43] (GPT-3.5, and GPT-4o) model. During this experiment, approximately 220 verified security rules were collectively generated. These properties primarily fall into three categories: *algorithmic constraints*, *physical constraints*, and *security architecture rules*. While the generated rules explicitly include these three broader types, they implicitly include some of the memory-related security constraints. The combined effort for automated generation and manual verification took about 1.5 person-months without considering the initial setup time. In contrast, purely manual development of a comparable set of rules is estimated to require around 4 to 6 person-months. Clearly, LLM is promising in automated generation of security rules. For example, GPT-4o is able to accurately generate 70-90% security rules, which can drastically reduce the manual effort. Although LLMs were able to successfully translate most of the cases in the specification, there were certain cases in which the generated security rules were not capturing the appropriate information. For example, in most of the constraints, the minimum and maximum bounds were not captured with the tightest possible bounds that were specified within the specification. Such scenarios should be manually inspected to ensure the accuracy of the generated rules.

Model	*LocalGPT*		*ChatGPT*	
Version	GPTQ-8	GPT-f16	GPT-3.5	GPT-4o
Parameters	7B	13B	175B	1800B

(a) LLM and their model parameters used in the evaluations.



(b) Percentage of valid and invalid security rule translations for function input arguments (**A**), variables (**V**), and constraints (**C**).

Fig. 15: Effectiveness of transformer-based LLM for security rule translation from specifications into pre-defined templates.

D. *RoT Firmware Security Violations*

This section describes the identified potential vulnerable places in evaluated four RoT firmware configurations. Table III illustrates the different security rules and security violations found in each configuration of the RoT firmware.

We have performed further analysis on the security violations identified by *FirmWall* against the existing CVEs [41] identified on the components used in assembling the **C1-C4** firmware configurations. Table IV presents the summary of the CVEs and corresponding RoT firmware configurations that contain the vulnerability. Note that most of these vulnerabilities can lead to high-severity exploitations, such as denial-of-service attacks, information leakage, and arbitrary code execution. We have analyzed the source code for the failed security

TABLE III: Summary of vulnerable places found in the four RoT firmware configurations. For this evaluation, we have included manually inspected LLM-translated security rules (**A**) as well as manually constructed checkers (**M**).

Rule ID	**Security Rule Description**	**Configuration**				**A/ M**
		C1	**C2**	**C3**	**C4**	
D_1	*Missing block cipher checks*	8	8	0	0	A
D_2	*Missing public cipher checks*	7	10	7	4	A
D_3	*Missing hash checks*	3	0	0	0	A
D_4	*Use-after-free*	9	9	9	9	M
D_5	*Heap inside system calls*	37	22	13	12	M
D_6	*Out-of-bound read*	39	31	21	12	A
D_7	*Out-of-bound write*	19	19	7	7	A
D_8	*Double free*	6	2	2	2	M
D_9	*Null dereferences*	7	0	0	0	M
D_{10}	*Request decoder checks*	6	4	0	0	A
D_{11}	*Using data from shared mem.*	6	6	6	2	M

rules to identify the underlying causes. In addition to missing checkers, the majority of the failed rules can be attributed to platform-specific configurations. In such cases, the root cause is the constructs, such as `#if defined (MACRO)`, which allows compile-time configurable firmware builds for different applications. For instance, a macro might conditionally include or exclude certain security features depending on the target platform. It would be infeasible to manually identify such scenarios from the source code, which highlights the need for our proposed framework.

TABLE IV: CVEs associated with security violations found on different RoT firmware configurations. (✓) indicate the presence of the vulnerability while (x) indicates the absence of the vulnerability in the particular firmware configuration.

CVE	**Rule ID**	**Severity**	**C1**	**C2**	**C3**	**C4**
CVE-2024-30166	D_6	High	x	✓	✓	x
CVE-2024-28960	D_{11}	High	✓	✓	✓	x
CVE-2024-45158	D_2, D_7	High	x	x	x	✓
CVE-2024-23775	D_2, D_7	Low	✓	✓	✓	x
CVE-2023-45199	D_2, D_7	High	✓	✓	x	x
CVE-2023-43615	D_1, D_6	Medium	✓	✓	x	x
CVE-2022-35409	D_3, D_6	Medium	✓	x	x	x
CVE-2021-44732	D_8	High	✓	x	x	x

V. Applicability and Limitations

In this work, our analysis was focused on RoT firmware implementations with respect to CWEs that are related to improper input sanitization and buffer-related properties. The scope can be extended beyond RoT firmware, specifically covering other real-time embedded system firmware implementations, such as automotive systems. Similar to expanding application domains, the monitoring rules can also be extended to cover more CWE types and other validation concerns, such as real-time guarantees, safety, functional coverage, etc. While our framework is well-suited for firmware verification in embedded systems, symbolic execution is likely to face state space explosion in verifying general-purpose systems, running large operating systems, such as Liunx or Windows.

Our experiments were limited to ARM Trusted Firmware-M due to several reasons: i) Trusted Firmware-M reference implementation is available to the public, ii) it is a widely used industrial solution, and iii) PSA specification of Trusted Firmware-M is well documented and complete, which facilitated LLM assisted security rule translation. However, *FirmWall* can be easily applied to other RoT firmware implementations with minimal changes if the specification and the implementation are accessible. LLM-assisted experiments were conducted without making any prior modifications to the transformer models outlined in Figure 15. It would be interesting to perform the experiments on these LLMs with careful tweaks to take optimized outputs with formal or statistical guarantees about translation accuracy.

Each of the steps in our framework are independent. For example, we used LLM to automate security rule translation. The security rules can also be derived manually or through other automated methods. Irrespective of how the security rules are derived, our framework will be able to detect the firmware vulnerabilities based on the security rules.

VI. Conclusion

Root-of-Trust (RoT) firmware provides trusted services through system calls to ensure confidentiality and integrity of both application code and user data while satisfying a wide variety of security requirements. While there are promising solutions for firmware verification, they either cannot verify vulnerabilities in firmware binaries or lead to state space explosion when dealing with modern (complex) firmware implementations. In this paper, we have presented a directed symbolic execution framework (*FirmWall*) to detect and mitigate firmware vulnerabilities. Specifically, we have developed efficient techniques to avoid state space explosion as well as automate various stages of our framework. For example, path vector guided symbolic execution framework drastically reduces the search space by ignoring non-relevant paths. Similarly, our lazy construction technique combines the advantages of static and dynamic CFG construction and thereby significantly reduces the CFG complexity. Moreover, we decompose large firmware binaries into smaller, more manageable, verification scenarios based on the region of interest to make symbolic execution tractable. We have performed extensive experiments on four RoT firmware configurations created based on ARM Trusted Firmware-M specifications and found numerous vulnerable places in the firmware implementations. These findings highlight the effectiveness of *FirmWall* in detecting critical vulnerabilities in the RoT firmware that should be mitigated before final deployment.

Acknowledgments

This work was partially supported by the Semiconductor Research Corporation (SRC) grant 2025-HW-3307.

References

[1] P. Jauernig, A.-R. Sadeghi, and E. Stapf, “Trusted execution environments: properties, applications, and challenges,” *IEEE Security & Privacy*, vol. 18, no. 2, pp. 56–60, 2020.

[2] “Common Weakness Enumeration: CWE,” https://cwe.mitre.org/, 2024.

[3] “NIST Vulnerability Database,” https://nvd.nist.gov/vuln, 2024.

[4] “CVE Details,” www.cvedetails.com/vulnerability-list/, 2024.

[5] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, “Sok:(state of) the art of war: Offensive techniques in binary analysis,” in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[6] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *IEEE Cybersecurity Development*, 2017, pp. 8–9.

[7] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[8] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FIRMSCOPE: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware," in *USENIX security symposium*, 2020, pp. 2379–2396.

[9] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "{FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *USENIX Security Symposium*, 2013, pp. 463–478.

[10] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[11] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2245–2262.

[12] A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Verifying hardware security modules with {Information-Preserving} refinement," in *USENIX Symposium on Operating Systems Design and Implementation*, 2022, pp. 503–519.

[13] A. Jayasena and P. Mishra, "HIVE: Scalable hardware-firmware co-verification using scenario-based decomposition and automated hint extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[14] "TF-M secure services documentation," https://tf-m-user-guide.trustedfirmware.org/design_docs/services, Jul 2024.

[15] "PSA certified apis," https://arm-software.github.io/psa-api/, Jul 2024.

[16] I. Security, "Intel Security Essentials," Intel Corporation, Tech. Rep. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/intel-security-essentials-solution-brief.pdf

[17] "AMD PRO Technologies," https://www.amd.com/en/products/processors/technologies/pro-technologies.html, 2024.

[18] "Synopsys tRoot Secure Hardware Root of Trust," www.synopsys.com/designware-ip/security-ip/root-of-trust.html, 2024, accessed: 2024-6-26.

[19] L. Duflot, F. Guihery, R. Findeisen *et al.*, "Trusted Platform Module Library," Trusted Computing Group, Tech. Rep., 2024.

[20] R. Tsang, D. Joseph, S. Salehi, P. Mohapatra, H. Homayoun *et al.*, "{FFXE}: Dynamic control flow graph recovery for embedded firmware binaries," in *USENIX Security Symposium*, 2024, pp. 5573–5590.

[21] M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa, "A hybrid approach for control flow graph construction from binary code," in *Asia-Pacific Software Engineering Conference*, 2013, pp. 159–164.

[22] T. Wu, L. Chen, G. Du, D. Meng, and G. Shi, "Ultravcs: ultra-fine-grained variable-based code slicing for automated vulnerability detection," *IEEE Transactions on Information Forensics and Security*, 2024.

[23] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *ACM SIGSAC conference on Computer & communications security*, 2013, pp. 499–510.

[24] H. Wen, Z. Lin, and Y. Zhang, "Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware," in *ACM SIGSAC conference on computer and communications security*, 2020, pp. 167–180.

[25] P. Reiter, H. J. Tay, W. Weimer, A. Doupé, R. Wang, and S. Forrest, "Automatically mitigating vulnerabilities in binary programs via partially recompilable decompilation," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[26] C. Pang, T. Zhang, R. Yu, B. Mao, and J. Xu, "Ground truth for binary disassembly is not easy," in *USENIX Security Symposium*, 2022, pp. 2479–2495.

[27] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "{X-Force}:{Force-Executing} binary programs for security applications," in *USENIX Security Symposium*, 2014, pp. 829–844.

[28] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μ vuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

[29] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[30] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2821–2837, 2021.

[31] A. Jayasena and P. Mishra, "Directed test generation for hardware validation: A survey," *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–36, 2024.

[32] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, pp. 1–45, 2024.

[33] K. Han, A. Xiao, E. Wu, J. Guo, C. Xu, and Y. Wang, "Transformer in transformer," *Advances in neural information processing systems*, vol. 34, pp. 15 908–15 919, 2021.

[34] Y.-C. Hu, A. Perrig, and M. Sirbu, "Spv: Secure path vector routing for securing bgp," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2004, pp. 179–192.

[35] N. Biggs, *Algebraic graph theory*. Cambridge university press, 1993, no. 67.

[36] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015.

[37] Microsoft, "Z3 theorem prover," https://github.com/Z3Prover/z3, 2023.

[38] "Trusted Firmware-M: Zephyr project documentation," https://docs.zephyrproject.org/latest/services/tfm/index.html, Jul 2024.

[39] "System Calls- Zephyr project documentation," https://docs.zephyrproject.org/latest/kernel/usermode/syscalls.html, Jul 2024.

[40] "CBOR library and tool providing code generation from CDDL descriptions." https://github.com/NordicSemiconductor/zcbor, Jul 2024.

[41] "Trusted Firmware-M Security Advisories," https://mbed-tls.readthedocs.io/en/latest/security-advisories/, 2024.

[42] PromptEngineer, "LocalGPT: Secure, local conversations with your documents," https://github.com/PromtEngineer/localGPT, 2024.

[43] OpenAI, "Chatgpt," https://openai.com/chatgpt, 2024.

[44] Y. Yao, W. Zhou, Y. Jia, L. Zhu, P. Liu, and Y. Zhang, "Identifying privilege separation vulnerabilities in iot firmware with symbolic execution," in *European Symposium on Research in Computer Security*, 2019, pp. 638–657.

Aruna Jayasena is a Ph.D student in the Department of Computer & Information Science & Engineering at the University of Florida. He received his B.S. in the Department of Computer Science and Engineering at the University of Moratuwa, Sri Lanka, in 2019. His research focuses on systems security, hardware-firmware co-validation, applied cryptography, trusted execution, side-channel analysis, and test generation.

Prabhat Mishra is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. He received his Ph.D. in Computer Science from the University of California at Irvine. His research interests include embedded systems, hardware security, formal verification, system-on-chip validation, machine learning, and quantum computing. He is an IEEE Fellow, an AAAS Fellow, and an ACM Distinguished Scientist.