Vulnerability Patch Verification for Military Software Systems Through AI-Driven Code-Level Rule Generation

Siam Shibly Antar
Research Assistant
School of Computing
Queen's University
Kingston, ON, Canada
siamshibly.antar@queensu.ca

Philippe Charland
Defence Scientist
Mission Critical Cyber Security Section
Defence Research and Development Canada
Quebec, QC, Canada
philippe.charland@drdc-rddc.gc.ca

Steven H. H. Ding
Assistant Professor
School of Information Studies
McGill University
Montreal, QC, Canada
steven.h.ding@mcgill.ca

Benjamin C. M. Fung
Professor
School of Information Studies
McGill University
Montreal, QC, Canada
ben.fung@mcgill.ca

Abstract: Patch verification is critical in military systems to ensure that known vulnerabilities are effectively addressed, preventing them from being exploited. Without proper verification, unpatched software could allow adversaries to exploit vulnerabilities, leading to unauthorized access, compromised operations, or even mission failure. In high-stakes environments such as military operations, patch verification is essential for maintaining the security, integrity, and readiness of both software and firmware, particularly in systems that manage sensitive information or control mission-critical equipment.

Traditional methods that rely on version strings to verify vulnerability patching are often insufficient. For example, the Heartbleed vulnerability (CVE-2014-0160) affected OpenSSL versions 1.0.1 through 1.0.1f. A system running OpenSSL 1.0.1f might still be flagged as vulnerable, even if a custom patch was applied, in the event that the version string was not updated by the software maintainer fixing the vulnerability. This will lead to false positives in the vulnerability detection process. Conversely, a system may appear secure based on the version string, but if the patch was not correctly implemented, the vulnerability will remain, resulting in false negatives. To address these limitations, this paper presents a new scalable, artificial intelligence-based code-level verification system. By leveraging large language models to generate rules that analyze the actual executable code, this approach verifies whether vulnerabilities have been properly fixed, regardless of version metadata. Additionally, it can pinpoint the exact location of exploitable code as a more accurate and reliable method for detecting and confirming patches. Our experiment, involving 1,466 vulnerable software records with over 4,000 instances, demonstrates that the rule generation system is both accurate and robust.

Keywords: *patch verification, vulnerability detection, firmware analysis, artificial intelligence, large language models*

1. INTRODUCTION

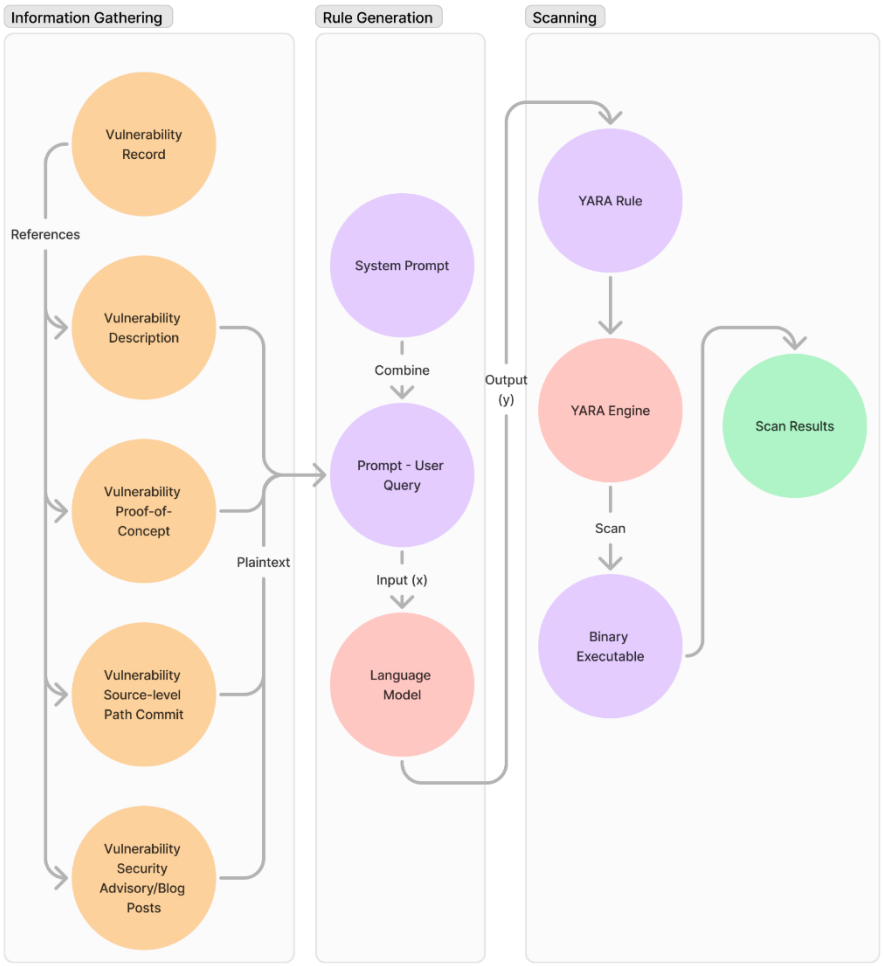
Vulnerability patch verification is a critical process in maintaining the security and reliability of software systems, particularly in high-stakes environments such as military operations. It ensures that known vulnerabilities, such as newly disclosed common vulnerabilities and exposures (CVEs) or any in-house new vulnerability records, have been effectively addressed or evaluated to mitigate potential risks. For example, when a new vulnerability is publicly announced through a CVE or found by internal threat teams, organizations must rapidly assess the associated risks and confirm that their systems are not vulnerable. This process is important for preventing unauthorized access, data breaches, and operational disruptions that could compromise mission-critical systems. An example of this urgency was seen with the Shellshock vulnerability (CVE-2014-6271), where exploits targeting this Bash flaw appeared within hours of its disclosure, affecting millions of devices reliant on Bash for system-level operations [1]. Even after almost 10 years, it remains one of the most exploited vulnerabilities, despite a patch being available [2]. This underscores the importance of swift response and thorough verification of newly discovered vulnerabilities.

Traditional methods of vulnerability verification often rely on information provided by software vendors. This approach is fraught with challenges. The complexity of modern software supply chains, combined with the increasing prevalence of supply chain attacks—such as the SolarWinds attack [3], which exploited software updates to introduce malicious code into thousands of organizations, or the attack on Kaseya VSA [4], where a compromised IT management tool led to widespread ransomware infections—undermine the reliability of vendor-provided data. A one-day or multi-day delay in vendors validating and addressing newly disclosed CVEs can leave systems exposed. Additionally, open-source software (OSS) further complicates this landscape [5], as its decentralized nature can lead to inconsistencies in patch deployment and versioning. An example would be the Log4j vulnerability (CVE-2021-44228), which impacted millions of devices globally [6]. This flaw in the widely used Log4j library, part of the Apache Logging Services, exposed systems to remote code execution attacks, highlighting how a single OSS vulnerability can have widespread consequences. Vendors may customize OSS libraries or fail to update version strings, making it difficult to determine whether a vulnerability has been patched. Vendor information cannot always be trusted.

To address these shortcomings, a zero-trust approach has gained traction, involving three key steps: generating a software bill of materials (SBOM) [1], monitoring new CVEs that match the SBOM catalog, and analyzing systems for unpatched vulnerabilities. An SBOM provides a detailed inventory of software components, including their origins, versions, and dependencies, enabling organizations to map CVEs to their systems. For example, consider a system running OpenSSL with the version string “OpenSSL 1.0.1f.” This version string can be linked to product details, allowing tools to identify known vulnerabilities such as the Heartbleed vulnerability (CVE-2014-0160), which affects OpenSSL versions 1.0.1 through

1.0.1f [2]. FACT [7], EMBA [8], CVE Binary Tool [9], and ERS0 [10] follow such an approach. However, reliance on version string matching introduces significant risks. Vendors may adopt OSS or software from sub-vendors with altered version string patterns, complicating the identification process. Additionally, vendors may implement in-house patches for CVEs without updating the version string, especially when other parts of the code remain unchanged. A patched version of OpenSSL, for instance, might still appear vulnerable if the version string remains unmodified. Conversely, systems might seem secure based on metadata, while still harboring unpatched vulnerabilities due to incomplete fixes or custom versions. These limitations highlight the need for a more robust and precise method.

Figure 1: The process begins with information gathering, which serves as the input for a language model. The generated prompts lead to the creation of YARA rules, which are subsequently utilized for scanning binary files, culminating in the generation of scan results



A novel approach bypasses these metadata-based limitations by directly verifying the presence of unpatched CVEs in software binaries (see Figure 1). Using publicly available CVE patch information, such as source code commit logs, pattern-matching rules can be

generated to identify instances of unpatched vulnerabilities in executable code. YARA rules [11], a flexible and performance-optimized pattern-matching framework, have been selected for this purpose. Commonly used in malware detection and triage, YARA rules enable efficient scanning of binaries, making them well-suited for large-scale vulnerability analysis.

While YARA rules are traditionally crafted manually, this process is time-consuming and does not scale to the volume of newly released CVEs. To address this, we propose a novel approach leveraging language models to automate the generation of YARA rules for unpatched CVEs. By taking CVE information, including proof-of-concept (PoC) exploits and patch commit logs as the input, the system generates YARA rules to detect the corresponding vulnerabilities in binary executables. This automated method not only accelerates the process but also enhances explainability, as the generated rules clearly delineate where vulnerabilities exist and why they remain unpatched. Preliminary investigations reveal that existing language models struggle to produce high-quality YARA rules. To overcome this limitation, we introduce a two-phase training methodology designed to improve the quality of the generated rules. The contributions of this paper are as follows:

- We propose a fast and reliable method for vulnerability patch verification and risk assessment, adopting a zero-trust approach that does not depend on vendor-provided information.
- We present a two-phase training framework for language models to generate high-quality vulnerability detection rules conforming to YARA specifications.
- We benchmark various language models for their effectiveness in generating vulnerability-matching rules, demonstrating the efficacy of our proposed approach.

This paper is structured as follows: Section 2 reviews related works. Section 3 formally defines the research problem. Section 4 outlines our methodology for model training. Section 5 details the experimental results. Finally, Section 6 provides the conclusion.

2. RELATED WORKS

Vulnerability detection involves identifying software flaws that can be exploited by attackers. It can be broadly categorized into static and dynamic approaches. Static vulnerability detection analyzes the source code, binaries, or intermediate representations without executing the program. Model-based approaches, such as taint analysis [12], track the flow of potentially malicious inputs through the program to identify insecure patterns. Data-driven methods leverage deep learning models trained on large datasets of vulnerable and non-vulnerable code to predict flaws [13]. While static methods provide comprehensive coverage, they may produce false positives due to the lack of runtime context.

Dynamic vulnerability detection, on the other hand, analyzes the software during execution to identify vulnerabilities that arise only under specific runtime conditions. Widely used

techniques include fuzz testing [14], which provides random or malformed inputs to the program, and symbolic execution [15], which systematically explores execution paths. While these methods are effective in finding runtime-specific vulnerabilities, they can be resource-intensive and may miss issues that are not triggered during testing.

Vulnerability scanning, the paradigm under which this work falls, focuses on identifying known vulnerabilities within software systems. This approach often utilizes an SBOM to map vulnerabilities to specific components within a system [7–10]. Another common method involves assembly code clone detection, which identifies code similarities to known vulnerable software [16, 17]. While assembly code clone approaches provide fuzzy matching results, typically in the form of a matching score between 0 and 1, they face challenges such as determining appropriate thresholds and requiring manual verification to finally confirm vulnerabilities. Despite these challenges, code clone techniques have advantages in identifying vulnerabilities at the binary level, but require disassembly, which can increase complexity.

Figure 2: Example YARA rule for detecting unpatched CVE-2017-9049

```
rule Detect_Unpatched_Vulnerability
{
    meta:
        description = "Detects presence of unpatched vulnerability in raw x86 binary executables"
        date = "2025-01-03"
        reference = "Based on provided CVE information"

    strings:
        // Look for patterns removed in the patch
        $vulnerable_code_1 = { E8 ?? ?? ?? ?? 83 C4 04 85 C0 75 0F }
        // Removed function call
        $vulnerable_code_2 = { 8B 45 FC 89 45 F8 83 7D F8 00 75 }
        // Pattern near vulnerable logic
        $missing_buffer_check = { 3B ?? ?? 7C ?? E8 ?? ?? ?? ?? }
        // Missing check for input buffer

        // string indicating lack of a new error message
        $no_error_message = "unexpected change of input buffer" ascii wide

        // ASCII strings commonly found in libxml binaries
        $libxml2_string1 = "xmlParseDoc" ascii
        $libxml2_string2 = "xmlReadMemory" ascii

    condition:
        // Match conditions for unpatched code presence
        ($libxml2_string1 or $libxml2_string2) and
        (two of ($vulnerable_code_1, $vulnerable_code_2, $missing_buffer_check)) and
        not $no_error_message
}
```

Our work diverges by emphasizing fast triage through explainable matching patterns. Unlike binary code clone approaches, which prioritize detailed matching at the cost of performance, our method focuses on generating transparent and actionable vulnerability rules. This approach balances precision and efficiency, providing a scalable method for rapid vulnerability scanning and verification as a standalone solution, or a complement to existing

clone search-based methods. We are among the first to adopt this strategy, combining explainability and speed to address the challenges of vulnerability verification in a novel and effective way.

3. PROBLEM DEFINITION

The problem involves transforming vulnerability record information into actionable detection rules for binary files. The input consists of a released CVE's details, including its description and all related data available under the references section formatted as plain text and denoted as x. For example, on the National Institute of Standards and Technology national vulnerability database, there is a "References to Advisories, Solutions, and Tools" section for each CVE record.

This information is collected using automated crawlers that retrieve relevant details such as threat advisories, descriptions, source code commits of patches, PoC exploits, and blog posts analyzing the vulnerability. Leveraging this diverse data source ensures a comprehensive understanding of the vulnerability and its exploitation patterns for rule generation.

Figure 3: System prompt design

```
You are a cybersecurity expert specializing in Yara rule creation. Given detailed information about a CVE, generate a
Yara rule that can accurately detect the unpatched vulnerability in binary files. Ensure the rule adheres to Yara
syntax specifications and includes:

- A meta section describing the rule purpose, CVE reference, and author details.

- A strings section with carefully chosen patterns that reflect patch code content, ensuring inclusion of:

    - Byte or text strings confirming the target library or functionality (e.g., JSON processing or XML processing,
    etc.) to reduce false positives.

    - Byte or text strings reflecting changes introduced in the patch, such as new function call instructions, new
    constants, or updated error messages, etc.

    - Byte or text strings directly related to the specific patch or fix implementation.

- A condition section combining the patterns logically, with checks to:

    - Confirm the library or functionality targeted.

    - Verify the location of the patch within the binary.

    - Ensure the patch is not present, distinguishing unpatched instances.

Following is an example Yara Rule:
...

Write down your thinking and reflection process here:
### begining of the thinking process

### end of the thinking process

Write down your final rule here:
### begining of the yara rule
```

In this paper, we focus on public records to build the required input dataset. These records include advisories from official CVE databases, Git repositories documenting patch implementations, security researchers' PoC codes (optional), and technical blogs discussing

the vulnerability's scope and impact. While our approach is based on public data, the same methodology can be applied to in-house vulnerability records, where organizations can gather similar information internally through proprietary systems and sources.

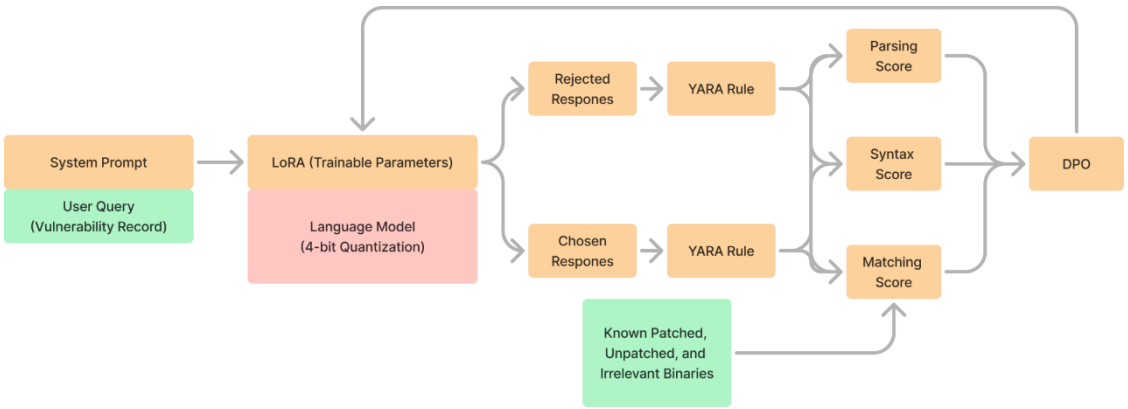
The goal is to generate a YARA rule, denoted as y, capable of identifying unpatched instances of the vulnerability in binary executables (see Figure 2). YARA rules provide explainable and precise matching patterns that facilitate rapid detection and verification of vulnerabilities across diverse systems. By automating this process, we aim to enhance scalability, while maintaining high levels of accuracy and interpretability for vulnerability detection.

4. METHODOLOGY

A. Prompt Engineering for YARA Rule Generation

The initial step starts with designing effective prompts to guide the language model in generating YARA rules (see Figure 4). Prompts typically consist of two parts: the system prompt and the user query [18]. The system prompt provides a detailed set of instructions and context for the model, such as "Generate a YARA rule for detecting a vulnerability based on the provided CVE details. Ensure the rule adheres to YARA specifications and includes meaningful identifiers and conditions." This part sets the task's scope and quality expectations. The user query, by contrast, supplies the specific input data for the task. For example, a query might state: "Based on CVE-2021-44228, generate a YARA rule. The CVE details are as follows: ### start of CVE details .... ### end of CVE details."

Figure 4 The overall training workflow and reward score calculation



We first draft a base system prompt that incorporates key elements such as YARA rule structure, syntax requirements, and general considerations about rule quality. This base prompt is then iteratively refined using outputs from a separate language model. Manual feedback is employed to evaluate the generated rules for alignment with predefined standards, such as syntactic validity and contextual accuracy. This iterative refinement involves adjusting the phrasing, e.g., inclusion, and input-output formats of the prompts to

optimize the model's ability to produce high-quality and consistent YARA rules. Figure 3 shows our example system prompt. Contextual information about the vulnerabilities will be used as the user query prompt.

B. Language Model Initial Setup

To enhance the efficiency and scalability of rule generation, we employ low-rank adaptation (LoRA) and 4-bit quantization (see Figure 4). These optimization methods enable the effective adaptation of pre-trained language models to the specialized tasks, in our case YARA rule generation, while minimizing computational and resource overhead. Especially for model fine-tuning, the reduced overhead enables us to train the model in faster iterations.

LoRA is a fine-tuning method that optimizes pre-trained models by injecting additional learnable parameters into low-rank matrices within specific layers of the model [19]. This approach focuses on training only the newly introduced parameters, while leaving the pre-trained weights untouched. By reducing the number of trainable parameters, LoRA significantly decreases memory and computational requirements compared to traditional fine-tuning. This makes LoRA particularly useful for tasks requiring domain-specific adaptation, such as cybersecurity applications, where the model can efficiently specialize in YARA rule generation without losing its general-purpose capabilities.

In 4-bit quantization, a model is compressed by representing its weights with 4 bits instead of the typical 16 or 32 bits, reducing the model size drastically [20]. This compression allows for faster inference times and enables deployment on hardware with limited computational power, such as edge devices or low-resource servers. Despite the reduction in precision, modern quantization techniques use algorithms to maintain the model's accuracy, ensuring that it performs well even under these constraints. For YARA rule generation, 4-bit quantization ensures that the model is efficient enough for real-time and large-scale applications in varying application scenarios.

C. Iterative Sampling for YARA Rule Syntax Correction

The language model may fail to generate syntactically correct YARA rules due to issues such as:

- Including extraneous explanation text or code snippets outside the designated response area, leading to extraction errors.
- Producing YARA rules that are not syntactically valid.

To address these challenges, we consider two methods for training the existing language model: direct preference optimization (DPO) and proximal policy optimization (PPO). DPO is a stable and efficient approach to reward-based fine-tuning, while PPO uses reinforcement learning to iteratively improve outputs based on reward signals.

PPO [21] optimizes the model by iteratively interacting with a reward function. It evaluates outputs based on defined metrics, such as accuracy or syntax validity, and adjusts the model to maximize expected rewards. A clipping mechanism in PPO prevents overly large updates to the model parameters, ensuring training stability. However, PPO requires well-defined reward functions, extensive hyperparameter tuning, and significant computational resources, making it complex and resource-intensive for this application.

DPO [22], in contrast, simplifies the process by focusing directly on sampled preferences without requiring explicit reinforcement signals. DPO trains the model to rank outputs based on their quality, as determined by a reward function. This method avoids complex policy adjustments and uses a more straightforward sampling-based approach to refine outputs. DPO requires less computational overhead and delivers more stable results, making it well-suited for tasks such as generating syntactically correct YARA rules. Typically, the training dataset consists of a pair of different text responses given the same query: the chosen response and the rejected one. The chosen response has a higher award score than the rejected response.

In our case, we use DPO due to its simplicity, stability, and reduced resource requirements (see Figure 4). DPO provides a straightforward and interpretable optimization process, making it especially effective in scenarios with limited labeled data and tasks requiring high precision. We define the reward function as:

$$R(y) = \alpha \cdot P(y) + \beta \cdot S(y)$$

where:

- $R(y)$: The reward score for the generated response y.
- $P(y)$: A response format validity score (1 if the YARA rule y can be successfully extracted from the response template, 0 otherwise).
- $S(y)$: A binary validity score (1 if the YARA rule y is syntactically valid, 0 otherwise).
- (α, β): Weighting factors to balance the importance of syntax validity and semantic alignment.

We propose an iterative sampling and training algorithm for our YARA rule generation task:

- Step 1: Initialize the model temperature (τ) to encourage diverse responses.
- Step 2: For each CVE in the training set, gather the query data in plain text format.
- Step 3: Use the system prompt and query to generate a response.
- Step 4: Parse the response, extract the YARA rule, and assign a score for the response based on the reward function.

- Step 5: Repeat Steps 3 and 4 five times, leveraging non-zero temperature to explore diverse responses. Retain only the response that has the largest difference in score compared to the response in Step 4.
- Step 6: Form m training pairs using valid and invalid responses by repeating Step 5. Record the number of syntactically incorrect trials in Step 4 as n.
- Step 7: Train the model with these m training pairs, reducing the temperature exponentially based on n.
- Step 8: Repeat the process until τ is zero, with updated τ to refine the model's ability to consistently generate valid YARA rules.

The temperature adjustment in Step 7 follows an exponential decay formula, expressed as:

$$\tau_{i+1} = \tau_i \cdot e^{-\lambda n}$$

where (τ_i) is the current temperature at iteration (i), (λ) is the decay rate constant, and (n) is the number of trials. This ensures that the model progressively focuses on generating more precise outputs as training progresses, making bigger adjustments at the beginning and smaller ones when converging.

D. Iterative Sampling for Rule Matching Quality Improvement

Building upon the syntax correction framework, this step focuses on optimizing the matching quality of YARA rules. Instead of validating syntax alone, the reward mechanism evaluates the effectiveness of the rules in identifying vulnerabilities. The training set includes binaries categorized as containing known CVEs, patched known CVEs, and irrelevant binaries. The reward function for matching quality is defined as:

$$RM(y) = \gamma \cdot R(y) + \delta \cdot F1(y)$$

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

where:

- $RM(y)$: Reward score for the generated response y.
- $R(y)$: The syntax and parsing score from the previous step.
- $F1(y)$: F1 score evaluating the balance between precision and recall when tested on the labeled binary dataset.
- (γ, δ): Weighting factors to balance the importance of target matching and overall F1 score.

The iterative sampling and training algorithm involves:

- Step 1: Initialize the model temperature (τ) to encourage diverse responses.
- Step 2: For each CVE in the training set, gather the query data in plain text format and gather all the testing binaries.
- Step 3: Use the system prompt and query to generate a response.
- Step 4: Parse the response, extract the YARA rule y, and assign the matching score for the response based on the above reward function, by matching the rule y against the known binaries with labels.
- Step 5: Repeat Steps 3 and 4 five times, leveraging non-zero temperature to retrieve different responses. Keep only the response that has the smallest non-zero difference in score compared to the response in Step 4.
- Step 6: Form m training pairs using valid and invalid responses by repeating Step 5. Record the number of syntactically incorrect trials in Step 4 as n.
- Step 7: Train the model with these m training pairs, reducing the temperature (τ) exponentially based on n.
- Step 8: Repeat the process until τ is zero, with updated τ to refine the model's ability to consistently generate valid YARA rules.

This sampling algorithm is similar to the one above for syntax correction, except that the reward score is estimated based on the F1 score, which evaluates the effectiveness of matching the training binaries. Additionally, instead of selecting the pair with the largest score difference as the chosen and rejected responses, we choose the pair with the smallest non-zero difference. This strategy is justified because smaller non-zero differences indicate borderline cases where the model struggles to differentiate quality. Optimizing for such cases helps refine the decision boundary and improves the model's sensitivity to subtle distinctions, ultimately leading to better performance across diverse scenarios.

5. EXPERIMENT EVALUATION

A. Sample Set Building

We start by constructing a CVE vulnerability instance repository containing labeled binaries extracted from well-established firmware images, which serves as a robust testbed for evaluating the system's performance. Our dataset consists of two components: The first comprises popular open-source utility libraries, while the second includes Android OS built-in library vulnerabilities derived from the AOSP dataset [23].

Utility libraries play a significant role in software development, binary analysis, and multimedia processing, but they often exhibit a range of security vulnerabilities. Tools such as addr2line, as, and elfedit, which are respectively used for debugging, assembly, and

executable and linkable format (ELF) file manipulation, demonstrate critical flaws across various versions. For instance, addr2line includes vulnerabilities such as CVE-2018-18605, allowing buffer overflows, and CVE-2018-12697, leading to out-of-bounds reads. Similarly, the GNU assembler (as) has been affected by vulnerabilities such as CVE-2017-7230, an integer overflow issue, and CVE-2018-1000019, a stack overflow vulnerability, both of which could enable arbitrary code execution. Multimedia libraries such as ffmpeg, freetype, and libpng also show significant risks, with vulnerabilities such as heap buffer overflows (CVE-2017-7862 in ffmpeg) and use-after-free issues (CVE-2015-8126 in libpng), potentially leading to crashes or remote code execution. These vulnerabilities, arising from issues such as improper input validation and poor memory management, emphasize the need for rigorous security assessments of utility libraries. Table I presents the number of identified vulnerabilities, corresponding library versions, and confirmed CVEs for the open-source utility libraries.

Table I: Aggregated summary of utility libraries, versions, and confirmed CVEs

Library	Versions	CVEs	Example CVEs
addr2line	7	72	CVE-2017-14129, CVE-2014-8738, ...
as	2	2	CVE-2017-72.30, ...
elfedit	3	4	CVE-2018-20623, CVE-2017-15996, ...
exif	3	10	CVE-2012-2814, CVE-2012-2840, ...
expat	3	3	CVE-2015-1283, CVE-2012-6702, ...
ffmpeg	45	54	CVE-2017-14059, CVE-2016-7562, ...
freetype	7	63	CVE-2014-9656, CVE-2010-2807, ...
objcopy	2	5	CVE-2018-12699, CVE-2018-12700, ...
objdump	5	16	CVE-2017-8421, CVE-2017-14934, ...
openssl	18	75	CVE-2016-6306, CVE-2015-0289, ...
png	4	6	CVE-2015-8126, CVE-2015-7981, ...
qemu	10	30	CVE-2024-9594, CVE-2024-8612, …
readelf	2	7	CVE-2017-7209, CVE-2017-9042, ...
sftp	3	3	CVE-2010-4755, CVE-2017-15906, ...
ssh	4	8	CVE-2014-2653, CVE-2011-0539, ...
sshd	7	10	CVE-2016-3115, CVE-2013-4548, ...
tcpdump	3	90	CVE-2017-12902, CVE-2017-13035, ...
xml2	8	38	CVE-2015-8035, CVE-2017-9048, ...

Networking and file-sharing libraries are similarly impacted by security flaws. Tools such as objdump and objcopy contain vulnerabilities such as improper file handling (CVE-2018-6543), which can lead to denial-of-service conditions. Cryptographic libraries, like OpenSSL, suffer

from vulnerabilities such as CVE-2016-6306, where improper handling of certificates may result in man-in-the-middle attacks. XML parsing libraries, such as expat and xml2, are also prone to vulnerabilities, including buffer overflows (CVE-2017-9233) and out-of-bounds reads (CVE-2015-8241), which compromise application security. Furthermore, FTP and SSH tools are affected by input handling flaws and directory traversal vulnerabilities, enabling unauthorized access, denial of service, and remote code execution. These widespread vulnerabilities across utility libraries highlight the importance of implementing robust security measures to mitigate evolving threats.

The AOSP dataset [23], hosted on GitHub by Quarkslab, provides a detailed collection of CVEs tailored to the Android operating system. Given Android's extensive integration into various devices, including Internet of Things (IoT) platforms, its security plays a critical role in ensuring device protection. This dataset focuses on vulnerable binary executables, omitting Java-related issues, and allows for an in-depth comparison of pre-patch and post-patch binaries based on source code commit data. With coverage of more than 50 Android system components, such as Media Framework, System, Bluetooth, and SurfaceFlinger, the dataset captures a range of vulnerabilities and their potential effects on device operations. Table II presents the number of vulnerabilities identified in the top 15 built-in libraries of Android system components.

Table II: Top 15 components by Android OS built-in library vulnerability count

Component	CVEs	High Severity	Critical Severity	Example CVEs
System	202	146	53	CVE-2019-2115
Media Framework	201	106	80	CVE-2019-2176
Mediaserver	136	64	44	CVE-2015-3864
Framework	40	32	5	CVE-2019-2123
libstagefright	21	6	14	CVE-2015-1538
Audioserver	11	9	0	CVE-2017-0418
Libraries	10	6	0	CVE-2016-1839
Bluetooth	8	4	0	CVE-2016-0850
Framework APIs	5	5	0	CVE-2016-3750
system UI	5	5	0	CVE-2017-0638
Binder	4	4	0	CVE-2015-1528
Debuggerd	4	0	2	CVE-2016-2420
Expat	4	1	0	CVE-2012-6702
LibUtils	4	0	4	CVE-2016-3861
OpenSSL & BoringSSL	4	0	2	CVE-2016-0705

Among the 1,000 CVEs, vulnerabilities are categorized by severity and type, including Elevation of Privilege and Remote Code Execution. Key components, such as Media

Framework and System account, form a significant portion of the dataset, each containing over 200 vulnerabilities, many of which are high severity. Examples include CVE-2019-2115 in System, a privilege escalation issue, CVE-2019-2176 in Media Framework, a remote execution risk, and CVE-2015-3864 in MediaServer, which impacts media rendering. Other components, such as libstagefright, highlight the risks associated with multimedia processing. This dataset serves as a foundation for understanding vulnerabilities within Android's binaries, aiding efforts to improve the security of IoT devices relying on this architecture. In total, our experiment covers 1,466 vulnerable software records, resulting in 4,218 instances of binary executables for analysis. Additionally, we included 10,000 irrelevant binaries in our experiment to evaluate the false positive rates of the methods.

B. Language Models

This experiment evaluates the performance of five state-of-the-art language models: LLaMA 3.3, Qwen 2, Gemma 2, and Mistral 0.3. These models represent advanced approaches in natural language processing and machine learning, demonstrating varying capabilities in understanding and generating complex patterns from data.

- LLaMA 3.3 [24]: A cutting-edge large language model designed for general-purpose natural language tasks. It focuses on efficiency and scalability, making it suitable for applications requiring high accuracy and rapid inference.
- Qwen 2 [25]: Known for its optimization in handling domain-specific language tasks, Qwen 2 leverages fine-tuned datasets to enhance contextual understanding and generate precise outputs, particularly in technical and specialized areas.
- Gemma 2 [26]: This model excels in multilingual and cross-lingual tasks, offering robust performance across diverse languages. It employs advanced transformer architectures to ensure consistency and coherence in its results.
- Mistral 0.3 [27]: A lightweight yet highly efficient model optimized for resource-constrained environments. Despite its smaller size, Mistral 0.3 delivers competitive performance, making it a practical choice for scalable applications.

The experiment was conducted on a server equipped with a 56-core Xeon Gold 2.3/3.9GHz processor, 100GB of RAM, and two NVIDIA GeForce RTX 6000 cards with (24 GB x 2) of VRAM. The training system was implemented using the HuggingFace Transformer Reinforcement Learning (TRL) library. To assess the performance of the methods, the following evaluation metrics were utilized:

- Precision: This metric measures the accuracy of positive predictions, indicating the proportion of true positives among all instances predicted as positive. It reflects the system's ability to avoid false alarms.

- Recall: Also known as sensitivity, recall evaluates the system's ability to identify actual positive instances, highlighting how well true positives are detected.
- F1 Score: The F1 score combines precision and recall into a single metric, providing a balanced measure of the system's accuracy, particularly when dealing with imbalanced datasets.

These metrics provide a detailed assessment of the system's performance. We conduct evaluations for each setup, including the original language model, the language model enhanced with syntax correction, and the language model further improved with syntax correction and quality improvement. This layered evaluation helps identify the impact of each enhancement on the model's accuracy and reliability.

Table III: Performance benchmark

Model (Solution)	Recall	Precision	F1
LLaMA 3.3	0.541	0.019	0.037
Qwen 2	0.622	0.012	0.024
Gemma 2	0.263	0.040	0.069
Mistral 0.3	0.342	0.024	0.045
LLaMA 3.3 (syntax correction)	0.620	0.451	0.522
Qwen 2 (syntax correction)	0.774	0.672	0.719
Gemma 2 (syntax correction)	0.561	0.836	0.671
Mistral 0.3 (syntax correction)	0.832	0.681	0.749
LLaMA 3.3 (syntax correction + quality improvement)	0.992	0.781	0.874
Qwen 2 (syntax correction + quality improvement)	0.986	0.893	0.937
Gemma 2 (syntax correction + quality improvement)	0.912	0.866	0.888
Mistral 0.3 (syntax correction + quality improvement)	0.956	0.901	0.928

C. Experimental Results

Table III shows the experimental results. The evaluation of the language models in their original configurations highlights their limited ability to handle the task effectively. Metrics for recall, precision, and F1 score remain low across all models. For example, LLaMA 3.3 achieves a recall of 0.541 but a precision of only 0.019, resulting in an F1 score of 0.037. Qwen 2, Gemma 2, and Mistral 0.3 show similar patterns, with F1 scores ranging from 0.024 to 0.069. These outcomes suggest that the models, in their initial states, struggle to balance the identification of true positives with the minimization of false positives.

When syntax correction is applied, significant improvements are observed in all models. LLaMA 3.3, for instance, achieves an F1 score of 0.522, a marked improvement from its original performance. Qwen 2, Gemma 2, and Mistral 0.3 also show improved metrics, with

F1 scores increasing to 0.719, 0.671, and 0.749, respectively. Syntax correction addresses structural inconsistencies, enabling the models to better interpret and process data. This adjustment results in higher recall and precision, demonstrating its impact on performance.

The combination of syntax correction and quality improvement delivers the best results across all models. For example, LLaMA 3.3 reaches a recall of 0.992 and an F1 score of 0.874, while Qwen 2 achieves a recall of 0.986 and an F1 score of 0.937. Mistral 0.3 and Gemma 2 show similar improvements, with F1 scores of 0.928 and 0.888, respectively. These enhancements refine both the input and the underlying understanding of the models, leading to improved predictions and reduced errors. This approach highlights the benefits of combining structural corrections with quality refinements to achieve optimal performance.

6. CONCLUSION

The proposed method significantly improves vulnerability patch verification by combining large language models, syntax correction, and quality enhancement techniques. Experimental findings demonstrate the limitations of initial models, and the performance gains achieved through structured refinement. By automating the generation of YARA rules and focusing on syntax and matching precision, the system overcomes challenges faced by traditional version-based detection methods. This approach offers a scalable and accurate solution for verifying software vulnerabilities, addressing the demands of environments requiring high security and reliability.

7. FUTURE WORK

While our experiments show promising results in model training across a subset of tasks and datasets, new avenues of research remain open for further exploration and improvement. We plan to benchmark our model's performance on each dataset separately and address emerging common weakness enumerations (CWEs) While our study focuses on the existing architecture, many new transformer-based and other neural network variants are emerging. Further research could involve fine-tuning these novel architectures under similar conditions to benchmark performance, parameter efficiency, and speed. By pursuing these directions, we hope to deepen our understanding of model fine-tuning, leading to improved vulnerability patch verification.

REFERENCES

[1] L. J. Camp and V. Andalibi, "SBoM vulnerability assessment & corresponding requirements," (response to Notice and Request for Comments on Software Bill of Materials Elements and Considerations), National Telecommunications and Information Administration, 2021.

[2] R. Ramachandran. “Qualys Top 20 Most Exploited Vulnerabilities.” 2003. Accessed: Jan. 8, 2025. [Online]. Available: https://blog.qualys.com/vulnerabilities-threatresearch/2023/09/04/qualys-top-20-exploited-vulnerabilities

[3] E. D. Wolff, K. M. Growley, M. O. Lerner, M. B. Welling, M. G. Gruden, and J. Canter, “Navigating the SolarWinds supply chain attack,” *The Procurement Lawyer*, vol. 56, no. 2, 2021.

[4] H. Ghanbari, K. Koskinen, and Y. Wei, “From SolarWinds to Kaseya: The rise of supply chain attacks in a digital world,” *J. Inf. Technol. Teach. Cases*, Nov. 2024, doi: 10.1177/20438869241299823.

[5] P. Ladisa, H. Plate, M. Martinez, and O. Barais, “SoK: Taxonomy of attacks on open-source software supply chains,” in *Proc. IEEE Symp. Secur. Priv. (SP)*, San Francisco, CA, USA, May 2023, pp. 1509–1526.

[6] Y. Shen, X. Gao, H. Sun, and Y. Guo, “Understanding vulnerabilities in software supply chains,” *Empir. Softw. Eng.*, vol. 30, no. 20, Nov. 2024.

[7] Fraunhofer FKIE. “GitHub—fkie-cad/FACT_core: Firmware analysis and comparison tool.” Accessed: Jan. 8, 2025. [Online]. Available: https://github.com/fkie-cad/FACT_core

[8] E-M-B-A. “GitHub—e-m-b-a/embark: EMBArk—The firmware security scanning environment.” Accessed: Jan. 8, 2025. [Online]. Available: https://github.com/e-m-b-a/embark

[9] Intel. “GitHub—intel/cve-bin-tool: The CVE binary tool helps you determine if your system includes known vulnerabilities.” Accessed: Jan. 8, 2025. [Online]. Available: https://github.com/intel/cve-bin-tool

[10] M. Beninger, P. Charland, S. H. H. Ding, and B. C. M. Fung, “ERS0: Enhancing military cybersecurity with AI-driven SBOM for firmware vulnerability detection and asset management,” in *Proc. 16th Int. Conf. Cyber Conflict: Over the Horizon (CyCon)*, Tallinn, Estonia, May 2024, pp. 141–160.

[11] VirusTotal. “GitHub—VirusTotal/yara: The pattern matching Swiss knife.” Accessed: Jan. 8, 2025. [Online]. Available: https://github.com/VirusTotal/yara

[12] J. Clause, W. Li, and A. Orso, “Dytan: A generic dynamic taint analysis framework,” in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, London, U.K., July 2007, pp. 196–206.

[13] L. Li, S. H. H. Ding, Y. Tian, B. C. M. Fung, P. Charland, W. Ou, L. Song, and C. Chen, “VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution,” *ACM Trans. Privacy Secur.*, vol. 26, no. 3, art. no. 28, pp. 1–25, Apr. 2023.

[14] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in *Proc. 14th USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2020.

[15] F. Wang and Y. Shoshitaishvili, “Angr—The next generation of binary analysis,” in *Proc. IEEE Cybersecur. Dev. (SecDev)*, Cambridge, MA, USA, Sept. 2017, pp. 8–9.

[16] S. H. H. Ding, B. C. M. Fung, and P. Charland, “Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization,” in *Proc. IEEE Symp. Secur. Priv. (SP)*, San Francisco, CA, USA, May 2019, pp. 472–489.

[17] Z. Fu, S. H. H. Ding, F. Alaca, B. C. M. Fung, and P. Charland, “Pluvio: Assembly clone search for out-of-domain architectures and libraries through transfer learning and conditional variational information bottleneck,” 2023, *arXiv:2307.10631*.

[18] L. Giray, “Prompt engineering with ChatGPT: A guide for academic writers,” *Ann. Biomed. Eng.*, vol. 51, no. 12, pp. 2629–2633, Dec. 2023.

[19] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “LoRA: Low-rank adaptation of large language models,” 2021, *arXiv:2106.09685*.

[20] Z. Yao, C. Li, X. Wu, S. Youn, and Y. He, “A comprehensive study on post-training quantization for large language models,” 2023, *arXiv:2303.08302*.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017, *arXiv:1707.06347*.

[22] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn, “Direct preference optimization: Your language model is secretly a reward model,” in *Proc. 37th Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, New Orleans, LA, USA, Dec. 2023, pp. 53728–53741.

[23] A. Challande, R. David, and G. Renault, “Building a commit-level dataset of real-world vulnerabilities,” in *Proc. 12th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, Baltimore, MD, USA, Apr. 2022, pp. 101–106.

[24] A. Grattafiori et al., “The Llama 3 herd of models,” 2024, *arXiv:2407.21783*.

[25] J. Bai et al., “Qwen technical report,” 2023, *arXiv:2309.16609*.

[26] M. Riviere et al., “Gemma 2: Improving open language models at a practical size,” 2024, *arXiv:2408.00118*.

[27] A. Q. Jiang et al., “Mistral 7B,” 2023, *arXiv:2310.06825*.