CyCon 2024: Over the Horizon
16th International Conference on Cyber Conflict
C. Kwan, L. Lindström, D. Giovannelli, K. Podiņš, D. Štrucl (Eds.)
2024 © NATO CCDCOE Publications, Tallinn

ERS0: Enhancing Military Cybersecurity with AI-Driven SBOM for Firmware Vulnerability Detection and Asset Management

Max Beninger
Research Assistant
School of Computing
Queen's University
Kingston, ON, Canada
max.beninger@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: Firmware vulnerability detection and asset management through a software bill of material (SBOM) approach is integral to defensive military operations. SBOMs provide a comprehensive list of software components, enabling military organizations to identify vulnerabilities within critical systems, including those controlling various functions in military platforms, as well as in operational technologies and Internet of Things devices. This proactive approach is essential for supply chain security, ensuring that software components are sourced from trusted suppliers and have not been tampered with during production, distribution, or through updates. It is a key element of defense strategies, allowing for rapid assessment, response, and mitigation of vulnerabilities, ultimately safeguarding military capabilities and information from cyber threats.

In this paper, we propose ERS0, an SBOM system, driven by artificial intelligence (AI), for detecting firmware vulnerabilities and managing firmware assets. We harness

the power of pre-trained large-scale language models to effectively address a wide array of string patterns, extending our coverage to thousands of third-party library patterns. Furthermore, we employ AI-powered code clone search models, enabling a more granular and precise search for vulnerabilities at the binary level, reducing our dependence on string analysis only. Additionally, our AI models extract high-level behavioral functionalities in firmware, such as communication and encryption, allowing us to quantitatively define the behavioral scope of firmware. In preliminary comparative assessments against open-source alternatives, our solution has demonstrated better SBOM coverage, accuracy in vulnerability identification, and a wider array of features.

Keywords: *vulnerability detection, firmware analysis, firmware management, artificial intelligence*

1. INTRODUCTION

The implementation of a software bill of materials (SBOM) practice in military operations is becoming increasingly critical, particularly in the context of supply chain security, asset management, and vulnerability management [1], [2]. Military operations typically rely on a complex network of platforms, operational technologies, and their software components that are often sourced from a myriad of suppliers, each with varying levels of trust and transparency. The complex nature of modern supply chains in software procurement and deployment inherently assumes a high level of trust in all participating suppliers. This trust-based approach, however, exposes these supply chains, particularly open-source software (OSS), to significant risks of supply chain attacks [3]. Such attacks can occur when a malicious actor infiltrates the supply chain at any point, potentially compromising the integrity and security of the software components being distributed or the pipelines through which the components are produced and integrated. This systematic weakness is especially concerning in environments where software plays a critical role in operational functionality and security, such as in military operations.

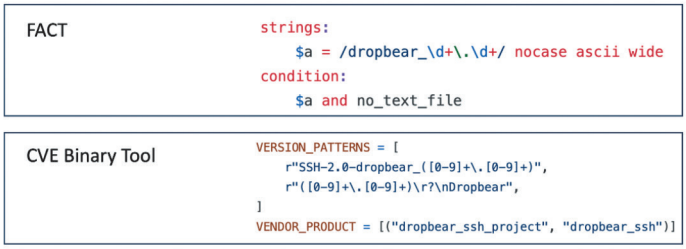
An SBOM serves as a strategic tool to mitigate these risks by introducing an element of transparency into the supply chain. By providing a comprehensive and detailed list of all software components used in a system, including their origins, versions, and dependencies, an SBOM makes it possible to scrutinize and validate each component's security and integrity. This level of transparency is crucial in identifying and addressing vulnerabilities that might otherwise go unnoticed in the complex

web of supply chain relationships. As a proactive approach, an SBOM is particularly valuable in preventing supply chain attacks, as it allows for the early detection of any anomalies or unauthorized alterations in the software components.

Obtaining a reliable SBOM is challenging, as relying on suppliers to provide this crucial information reintroduces the very trust issues SBOMs are meant to mitigate. Expecting suppliers to disclose the complete source code for all released firmware is also unrealistic, not only due to proprietary concerns but because the pipeline that transforms source code into final firmware can itself be compromised within the supply chain. Consequently, a shift-right approach is necessary, predicated on a zero-trust assumption toward suppliers. This approach seeks the development of specialized tools capable of directly extracting SBOMs from the already released or deployed firmware and software components. Such tools would independently analyze and generate a comprehensive list of components, bypassing the need to rely on supplier-provided information and ensuring a more accurate and secure assessment of the software's composition and potential vulnerabilities.

Existing solutions for SBOM generation primarily depend on manual processes, where specific string-matching patterns are crafted. These patterns are designed to detect various versions of strings linked to a particular product or open-source project. An example of this can be seen in Figure 1, which displays manually created regular expressions aimed at identifying different versions of the Dropbear library. This manual process is not only labor-intensive but also demands expertise in firmware analysis, given the extensive range of pattern variants necessary for effective matching. As a result, existing state-of-the-art solutions, such as the Firmware Analysis and Comparison Tool (FACT) [4] and the Common Vulnerabilities and Exposures (CVE) Binary Tool [5], are limited in their scope, only able to identify a few hundred specific products and components. This limitation underscores the need for more advanced and automated methods in SBOM generation. Moreover, these solutions often overlook code-level patterns, which are crucial as they contain the actual vulnerabilities. The failure to capture these patterns represents a significant gap in the effectiveness and thoroughness of current SBOM generation methods. Another inadequately addressed aspect is the characterization of firmware's high-level capabilities, such as encryption, communication, and I/O operations. These capabilities are crucial for military operations, as they define the operational scope, the potential attack vectors, the presence of a possible backdoor, and the integrity of the firmware overall. Therefore, there is a pressing need for a solution that not only identifies the components of an SBOM but also comprehensively understands and delineates the high-level functionalities and potential capabilities of the firmware.

FIGURE 1: EXAMPLES OF MANUALLY CREATED RULES FOR VERSION STRING MATCHING IN TWO DIFFERENT SBOMS TOOLS



To address these challenges, we propose a novel system, ERS0, that leverages machine learning to scale up and expand the SBOM creation process. Our methodology includes two key components: (1) character-level string similarity learning for precise version string and product detection, and (2) cross-architecture code clone search for OSS library and version identification. This approach considerably broadens the scope and accuracy of SBOM analysis. Furthermore, we aim to detect high-level capabilities over firmware images using static analysis. We have developed a generative large language model (LLM) that is up-trained to create capability identification rules based on the ATT&CK behavior catalog [6]. ERS0 not only enhances the accuracy of SBOM identification but also provides a comprehensive understanding of the firmware's functionalities and potential vulnerabilities, thereby substantially contributing to the security and robustness of military operations. Our contributions can be summarized as follows:

- We propose a learning-based version string-matching approach to address the scalability of the original manual process. Our proposed system, ERS0, is capable of accommodating over 1.4 million variant packages across a diverse range of products.
- We propose an efficient SBOM generation method based on cross-architecture assembly code clone search. This technique specifically targets the previously unaddressed gap in code-level analysis.
- We develop a generative LLM designed to create rules that effectively match string and code patterns correlated with the high-level behavioral capabilities of firmware.

This paper is organized as follows. Section 2 discusses the related tools in this domain. Section 3 describes the overall workflow of the ERS0 system and elaborates on each individual component. Section 4 demonstrates the effectiveness of our system, and Section 5 presents its interface. Finally, Section 6 concludes this paper.

2. RELATED WORKS

SBOM is becoming increasingly important for software security, especially for identifying components in software packages. SBOM generation is a relatively new area of research. Tools such as FACT, the CVE Binary Tool, and EMBArk are well known in this domain, but they have some scalability drawbacks. FACT helps to break down and examine firmware, which is important for generating SBOMs for embedded systems and Internet of Things devices. FACT is good at automatically finding and analyzing parts of firmware, helping to list all software parts needed for an SBOM. However, it requires manually created rules to identify these parts, which makes it less effective when there is a variety of different products. As more products with different types of firmware come out, updating these rules by hand can lead to missing or old information in SBOMs.

The CVE Binary Tool [5] scans software to find known vulnerabilities, adding to the SBOM by identifying potential security issues in the software components. It looks at software for versions that have known problems, which helps SBOMs show possible security risks. But, like FACT, the CVE Binary Tool also depends on rules that need to be manually generated. Keeping these rules up to date is labor-intensive, especially when software changes quickly and comes in many forms. This makes it hard for the tool to keep up and cover a wide range of software components accurately.

EMBArk [7] focuses on firmware security and is important for making SBOMs for embedded systems. It does a good job of automatically analyzing firmware and giving detailed information needed for SBOMs. But EMBArk also has the same problem as the other tools: It requires manually made rules to find parts and security issues. Writing these rules takes a lot of work and does not keep pace with the fast changes in software and the many different products available. This makes it less useful for generating SBOMs for a large number of products.

In short, while these tools are helpful for producing SBOMs and analyzing software security, their need for manually made rules makes it hard to use them for a wide range of products. This is a serious problem, because software is always changing and there are so many different types of software products available. We need ways to make and update SBOMs that are faster and can handle many different products at once.

3. SYSTEM DESIGN

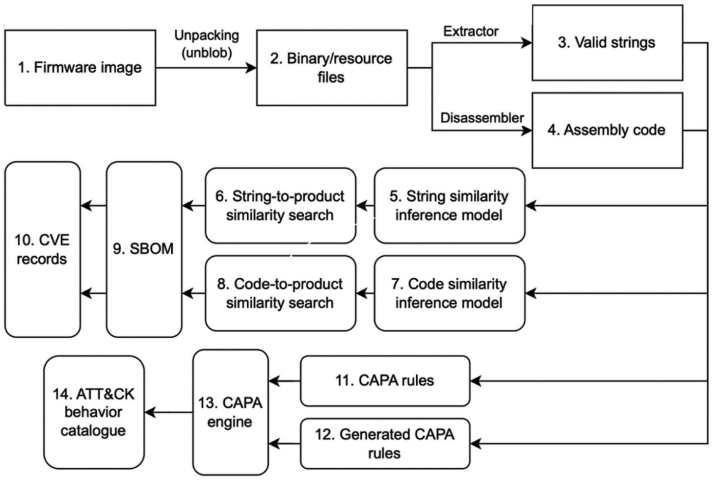
The automated SBOM generation process in ERS0 is a general workflow that starts with a firmware image and results in a thorough SBOM, complete with security vulnerability assessments and behavior analysis. This detailed process integrates the use of an open-source unpacking tool, machine learning models, and both predefined and dynamically generated rules to produce an SBOM that is informative not only in terms of component listing but also in terms of security analysis. The workflow, shown in Figure 2, is as follows:

- **Unpacking, extraction, and disassembly (Steps 1–4):** The analysis begins with a firmware image that will be scrutinized for its software contents. Utilizing unblob [8], an open-source tool that supports many image formats, the firmware is unpacked to isolate binary and resource files. After this, the extraction utility identifies valid strings from the binaries, which can be indicative of component names, versions, and other key metadata. Next, a disassembler abstraction module, compatible with both the Ghidra [9] and IDA Pro [10] disassemblers, converts the machine code of binary files into assembly code, which can be analyzed for richer insights.
- **String embedding and string-to-product search (Steps 5–6):** A machine learning model that transforms strings into an embedding space to identify their features and relationships to version strings, aiding in the identification of software components. An embedding space is a high-dimensional vector space where strings are represented as points or vectors. This representation facilitates the comparison of strings by measuring distances or angles between them, allowing for the identification of similar products based on their proximity in the space. Next, this module compares the resulting string embeddings to known product name embeddings to identify potential matches in software components.
- **Code embedding and code-to-product search (Steps 7–8):** Like the string model, this model transforms segments of assembly code into an embedding space to detect patterns and features that correspond to known software code components. The model is trained to match semantically similar assembly code across different platforms. This step involves comparing code embeddings with known product code embeddings to identify components within the firmware.
- **SBOM and CVE records matching (Steps 9–10):** All identified components, along with their versions and interrelations, are compiled into an SBOM. With the SBOM, ERS0 conducts a review of CVE records to determine if any of the identified components are associated with known vulnerabilities.

- **CAPA rules and generated CAPA rules (Steps 11–13):** ERS0 integrates the CAPA engine [11]. It is a part of the analysis system that applies predefined and dynamically generated rules to evaluate the capabilities and behaviors of software based on its code and metadata. It uses these rules to detect patterns that could signify potential security threats, vulnerabilities, or malicious activities within the software being analyzed. ERS0 uses predefined, manually created rules within the CAPA engine to abstract software capabilities from observed behaviors for threat analysis. In addition to the predefined rules, supplementary CAPA rules are generated by our language model to enhance the behavioral analysis.
- **ATT&CK behavior catalog (Step 14):** The system catalogs the identified behaviors according to the MITRE ATT&CK® framework [6], which serves as a global knowledge base of adversary tactics and techniques. More details are provided in Section 2.C.

Overall, the automated SBOM generation workflow facilitates not only the identification and documentation of software components but also the assessment of their security risks, thereby offering a robust tool for software component transparency and risk management.

FIGURE 2: OVERALL WORKFLOW OF ERS0 SYSTEM FOR FIRMWARE SBOM AND CAPABILITY ANALYSIS



A. Similarity Learning for Version String Matching

Instead of employing manual rule creation, we propose the use of a machine learning model to efficiently capture string patterns associated with various products on a large

scale. This machine-learning model is designed to convert a given input string into a numeric vector embedded within a high-dimensional space. Specifically, the model aims to ensure that the vector representation of a version string closely aligns, in terms of angular similarity, with its corresponding product name. Conversely, an invalid version string or one belonging to a different product should exhibit dissimilarity with the product name in this embedded space. By adhering to this principle, the model can effectively memorize how different variations of version strings should relate to their associated product names.

For instance, let us take the product name "pdns" and its valid version string "[lua2backend]," which corresponds to the lua2 backend version 4.7.3. Here, the model is trained to yield a similarity value of 1, since "[lua2backend]" is a valid version string for pdns. However, it should produce a similarity value of 0 when presented with the OpenSSL product name. Invalid version strings are those that may appear to conform to the standard format of major, minor, and build versions, but do not accurately represent the product's version. For example, the string "IEEE 802.11" might seem like a valid version string with "802.11" as its version number. However, it pertains to the network communication protocol and should not be treated as an SBOM entry. Therefore, it should yield a similarity value of 0 when compared to all product names. The model is trained on a dataset that follows this format, comprising over 4.3 million unique valid version string patterns. The dataset comprises pre-compiled Linux libraries, excluding Windows binaries, to test cross-platform generalizability.

The model itself operates as a character-level language model. It accepts a raw string as input, applies character-level tokenization, converts the raw string into a sequence of characters, and encodes each character into a numeric vector (embedding) using multiple layers of transformer models. Specifically, we leverage the CANINE [12] pre-trained language model as the encoder. Additional details can be found in the original paper [12]. This model has been pre-trained on an extensive text corpus, encompassing various domains such as news postings, Wikipedia articles, and programming questions/answers. Leveraging this pre-trained language model facilitates semantic matching between product names and version strings. For example, "libcrypto" is a library within the "libssl" package, which is part of the OpenSSL project. The pre-trained language model, without further fine-tuning, can already establish a high degree of similarity between "libcrypto" and both "libssl" and "openssl."

As shown in Figure 3, the encoder model is up-trained following a Siamese architecture. A Siamese network is a type of neural network architecture used for learning similarity or dissimilarity between pairs of data points [13]. In our case, the goal is to measure the similarity between product names and version strings. Let us denote the input product name as P and the input version string as V. These inputs are

converted into numeric vectors using the CANINE pre-trained language model. Let $f(P)$ and $f(V)$ be the embeddings (numeric vectors) of P and V, respectively:

$$f(P) = \text{CANINE}(P)$$
$$f(V) = \text{CANINE}(V)$$

The cosine similarity between the embeddings $f(P)$ and $f(V)$ is calculated as:

$$\text{cosine\_similarity}(f(P), f(V)) = \frac{f(P) \cdot f(V)}{\| f(P) \| \cdot \| f(V) \|}$$

Now, the cosine loss function can be defined as follows:

$$\text{Cosine\_Loss}(f(P), f(V), \text{label}) = \begin{cases} 1 - \text{cosine\_similarity}(f(P), f(V)), & \text{if label} = 1 \\ \text{cosine\_similarity}(f(P), f(V)), & \text{if label} = 0 \end{cases}$$

We want to minimize the cosine similarity between valid pairs (product name and its corresponding version string) and maximize the cosine similarity between invalid pairs (product name and a version string of a different product). In practice, a label of 1 is assigned to denote a valid pair of strings, while a label of 0 is used for an invalid pair of strings.

During the deployment stage, as shown in Figure 4, the first step involves encoding all existing product names into vector representations. These vectors serve as a reference database against which incoming strings can be compared. The encoding is carried out by the above trained encoder, which transforms raw string data into a high-dimensional space where semantically similar terms are placed closer together.

FIGURE 3: A SIMILARITY-BASED MACHINE LEARNING MODEL FOR THE VERSION STRING-TO-PRODUCT NAME-MATCHING PROBLEM

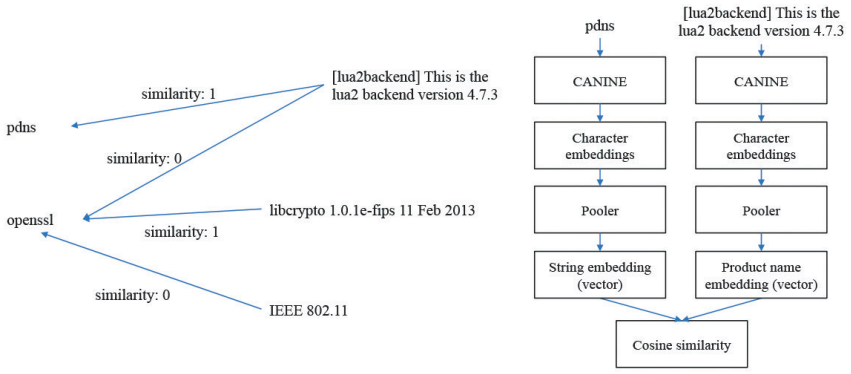
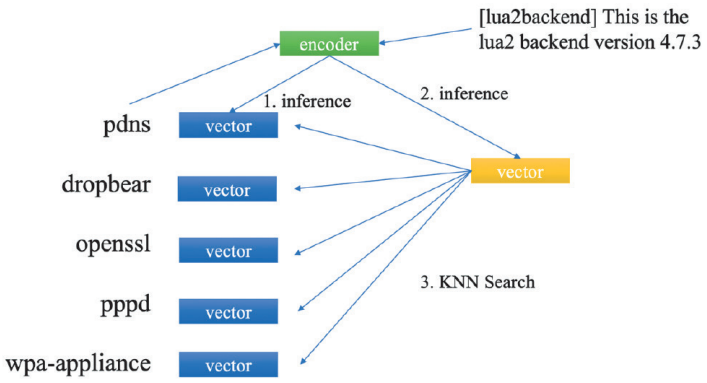


FIGURE 4: MATCHING AND SEARCHING PROCESS DURING THE DEPLOYMENT STAGE



When a new incoming string, which could be a potential version string, is received, the same encoder processes it to generate its vector representation. This vector is then compared against the pre-encoded vectors of product names, using a cosine similarity measure. The process searches for the vector among the pre-encoded product names that are most similar to the vector of the incoming string, limited by a predefined threshold. If a vector that exceeds this threshold is found, the incoming string is considered to be a match for the corresponding product name. This method allows for robust string matching, accommodating variations and minor discrepancies that often occur in real-world data.

B. Code Clone Search for Library Matching

This module functions in a way that is similar to the string-matching module, but it has a different data target. Its primary goal is to compare the assembly code of a given binary executable with the code of known product binaries. It utilizes a specifically trained model known as Pluvio [14] to identify similarities in assembly code across various computer architectures. This model goes beyond the traditional approach of matching version strings to product names. Instead, Pluvio is designed to detect assembly code that is similar in function and purpose across different types of platforms and against variations of the compilation toolchain. For example, a training sample involves the task of comparing two versions of a checksum algorithm CRC32 [15], one compiled for an ARM processor and the other for an AMD64 processor. The model's training on such examples enables it to recognize functions that have similar purposes or behaviors, even though they might be compiled in different ways.In the operational workflow, ERS0 first disassembles the binary executable. This task is carried out using a selected backend disassembler, as explained in the overall

workflow (referenced in Figure 2). The disassembler's job is to convert the machine code into assembly code, thereby revealing richer semantic information about the code's meaning and structure. Subsequently, the disassembler performs a control flow analysis. This analysis is crucial, as it segments the continuous stream of assembly code into individual and distinct assembly functions. Each binary executable is decomposed into a comprehensive list of assembly functions, making it easier to analyze and compare. When a new binary executable is extracted from a firmware, ERS0 compares it against a vast repository of known assembly functions from a wide range of products, extracted from open-source Linux and Windows packages. This repository is extensive and includes a diverse array of functions, providing a robust basis for comparison. By comparing the new executable to this repository, ERS0 can effectively identify any similarities or matches.

In the code matching algorithm (Algorithm 1), the process begins with Step 1, where the target binary is inputted as the subject of the search and analysis. Step 2 involves extracting all the assembly functions from the target binary, which requires disassembling the binary to understand its low-level code structure. In Step 3, each function *f* extracted from the target binary is iterated through for individual analysis. Step 4 uses the Pluvio model to search for clones of each function *f* in a comprehensive repository; these clones are variants of the function found in different binaries, and the search focuses on matches that meet a specified similarity threshold. Step 5 involves counting the origin of every clone identified in the previous step, thereby increasing a counter corresponding to its source binary or library; this step is crucial for tracking which binaries or libraries have functions most similar to those in the target binary. Step 6 involves selecting the top 10 binaries or libraries with the highest clone count, representing the binaries/libraries whose functions most frequently matched with those in the target binary. Finally, Step 7 refines this process by searching each function *f* from the target binary again, but only within the top 10 previously identified binaries/libraries, to ensure an accurate clone count. In Step 8, the algorithm checks the clone count threshold and picks the highest matched source binary or library, thus concluding the binary analysis and identification process.

Algorithm 1: Binary assembly function matching algorithm

```
Input: Target binary executable query Q, matching threshold θ, top k parameter k,
match ratio threshold ρ
Output: Matched source binary/library                    ▷ Step 1: Input target binary
procedure MatchAssemblyFunctions(Q, θ, k, ρ)
    F ← RetrieveAssemblyFunctions(Q)   ▷ Step 2: Retrieve assembly functions
    Counter ← InitializeCounter                          ▷ Initialize counter variable
    for each function f ∈ F do                                              ▷ Step 3
        C_f ← SearchClonesAgainstCompleteRepository(f, θ, k)                ▷ Step 4
        for each clone c ∈ C_f do                                           ▷ Step 5
            IncreaseCounter(Counter, origin(c))
        end for
    end for
    B ← GetTopBinariesWithHighestCount(Counter, k)                          ▷ Step 6
    for each function f ∈ F do                                              ▷ Step 7
        for each binary b ∈ B do
            SearchFunctionInLibraries(f, b)
        end for
    end for
    S ← CheckCloneCountThreshold(Counter, ρ)                                ▷ Step 8
    return PickHighestMatchedSource(S)
end procedure
```

C. Generative AI for Behavior Rule Generation

The final phase of the ERS0 project focuses on the development of behavior-matching rules. The objective is to leverage a large pre-trained natural language model, such as ChatGPT or Llama2, for generating YARA rules. YARA rules are essentially patterns or sets of conditions used to identify and classify binary executables [16]. Originally, YARA rules were used to classify and label malware, but their applications have since been extended beyond malware analysis into the domain of general binary analysis for threat hunting. CAPA is an open-source project leveraging YARA rules to identify high-level capabilities in binary executables. Figure 5 shows an example YARA rule. It tries to identify the "ws2_32. select" application programming interface (API) and label the binary with the capability to get socket status. The CAPA project and its YARA rules are crucial for identifying specific behaviors cataloged in ATT&CK, a comprehensive database of techniques employed by malicious entities. The example rule also defines the corresponding ATT&CK technique label T1016. T1016 in the MITRE ATT&CK framework refers to "System Network Configuration Discovery." This technique is part of the discovery tactic, where attackers seek to gather information about your network and systems, which can then be used to guide further actions, such as lateral movement through the network or understanding what defenses are in place.

FIGURE 5: EXAMPLE YARA RULE FROM THE CAPA OPEN-SOURCE PROJECT TO FIND CAPABILITIES PRESENTED IN A BINARY EXECUTABLE

```
rule:
  meta:
    name: get socket status
    namespace: communication/socket
    authors:
      - michael.hunhoff@mandiant.com
    scopes:
      static: function
      dynamic: call
    att&ck:
      - Discovery::System Network Configuration Discovery [T1016]
    mbc:
      - Communication::Socket Communication::Get Socket Status [C0001.012]
    examples:
      - 6A352C3E55E8AE5ED39DC1BE7FB964B1:0x1000C1F0
  features:
    - and:
      - api: ws2_32.select
```

However, the rule creation process relies on manual effort, requiring a security analyst with a strong background in understanding ATT&CK techniques, as well as strong familiarity with both low- and high-level programming APIs and libraries across many platforms. This poses a significant challenge, as the rule generation process must be scaled up to cover all ATT&CK techniques. We employ Retrieval Augmented Generation (RAG) prompt engineering to facilitate the rule creation process.

RAG combines language models with a retrieval system to generate text using external information [17]. After starting a prompt conversation, it first retrieves relevant information about the query to enrich the conversation context and then proceeds to the subsequent tasks provided in the prompt conversation. ERS0 uses the Atomic Red Team project [18] on GitHub as the external knowledge base. It contains both the technical description of each ATT&CK technique and several implementations of example attacks. This RAG design allows AI to provide up-to-date and domain-specific information dynamically. For each ATT&CK technique, ERS0 starts a prompt conversation with eight steps to create a YARA rule:

- **Step 1: Data provision.** In this step, the entire ATT&CK catalog, including technique descriptions and attack examples, is inputted into the language model from the Atomic Red Team project. This comprehensive data ensures that the model has a deep understanding of various attack techniques. This understanding is critical for accurate and effective rule generation.

- **Step 2: Technique identification.** This step involves requesting the model to provide a detailed description for a given ATT&CK technique ID. By understanding the specifics of the given technique, the model can tailor the generated rule to precisely match the behavior associated with that technique.
- **Step 3: Similarity analysis.** Here, the model is tasked with finding and listing the top five techniques most similar to a given ID based on their descriptions. This analysis helps enrich the context of the technique being targeted for rule generation and improves the accuracy of the rule being generated.
- **Step 4: System call identification.** This step instructs the model to identify typical Windows and Linux system calls that are related to the technique under consideration. Recognizing and incorporating relevant system calls is essential for pinpointing specific behaviors that need to be addressed by the rule.
- **Step 5: Advanced command analysis.** Advanced analysis involves having the model identify less common and undocumented Windows kernel APIs [19], if applicable, linked to the technique. These obscure APIs are often exploited in sophisticated attacks and including them in the rule ensures that such tactics are not overlooked.
- **Step 6: Cross-platform analysis.** Extending the analysis to include identifying relevant MacOS, Java, and Python APIs for the technique ensures that the generated rule is comprehensive and effective across different operating systems. This step broadens the scope of rule applicability.
- **Step 7: Constant value identification.** In this step, the model is asked to identify all potential constant values, such as hexadecimal numbers, floating-point numbers, decimal numbers, or string values, used by each of the APIs identified above. Constants often serve as key indicators in malicious operations, making their identification crucial for rule accuracy.
- **Step 8: YARA rule generation.** Finally, based on the information gathered in previous steps, the model is instructed to generate a YARA rule specifically tailored to detect the behavior associated with the targeted technique. This rule serves as a practical tool for identifying the presence of a specific malicious technique within a system.

By automating these steps, ERS0 overcomes the challenges of manual rule creation, enabling the generation of comprehensive and effective detection rules at a much larger scale. This method is not only efficient but also ensures that the rules are up-to-date and relevant across various systems.

4. EXPERIMENTAL EVALUATION OF SBOM MATCHING

We developed a benchmark dataset for SBOM generation, essential for firmware analysis to evaluate a tool's coverage of libraries and potential vulnerabilities. We assessed the performance of three tools: FACT, EMBArk, and the CVE Binary Tool, each in conjunction with ERS0. Our dataset was compiled by aggregating all packages from two sources: the Debian package repository for Linux pre-compiled packages and the Conan open-source package repository for MS Windows dynamic-link libraries (DLLs). This resulted in a total of 395,933 packages and 1,583,732 binary executables, each annotated with product and version information.

For our evaluation, we tested each tool individually along with ERS0. This approach was chosen because ERS0 is designed to be compatible with all the packages in our dataset, whereas each of the other tools has varying levels of support for different packages. We specifically assessed a tool on a package only if it had a predefined rule that corresponds to that package.

The results depicted in Table I provide a comparative analysis of three tools—FACT, EMBArk, and the CVE Binary Tool—in their performance of matching binaries to applicable packages, alongside the performance of ERS0. We assess tools based on their accuracy in matching binary executables, such as libcrypto.dll, to their correct vendor (e.g., OpenSSL) and version (e.g., 1.1.1). A correct match scores 1.

TABLE I: EXPERIMENTATION RESULTS

	FACT	EMBArk	CVE Binary Tool
Total number of applicable packages	42,838	78,854	102,571
Total number of applicable binaries	54,394	218,433	305,558
Number of binaries matched by the tool	129	1,075	46,495
Number of binaries matched by ERS0 and match percentage of Code-To-Product method	45,690 (14.3%)	163,824 (24.5%)	269,432 (16.1%)

FACT had a total of 42,838 applicable packages with 54,394 binaries. Of these, FACT matched 129 binaries, while ERS0 identified significantly more, with 45,690 matches. EMBArk, with 78,854 applicable packages and 218,433 binaries, matched 1,075 binaries. ERS0 again showed a higher matching count with 163,824 binaries. The CVE Binary Tool had the highest number of applicable packages at 102,571 and the highest number of applicable binaries at 305,558, with the tool itself matching 46,495

binaries. In this case as well, ERS0 demonstrated a more extensive matching capability, identifying 269,432 binaries. The effectiveness of code-to-product matching depends on the disassembler's performance and availability, but this limitation is mitigated by the string-to-product matching feature.

These results suggest that while each tool has its own capacity to match binaries, ERS0 shows a more robust performance in identifying binary matches across all tools. The high number of matches by ERS0 could indicate its higher matching coverage capability. However, the lower match counts for the individual tools do not necessarily imply poor performance; they may be highly specialized or conservative in their matching to ensure high accuracy. It is also important to consider the context and specific use cases in which one tool might perform better than the others despite the lower numbers, such as in scenarios requiring specific package support or higher precision.

5. SYSTEM INTERFACE

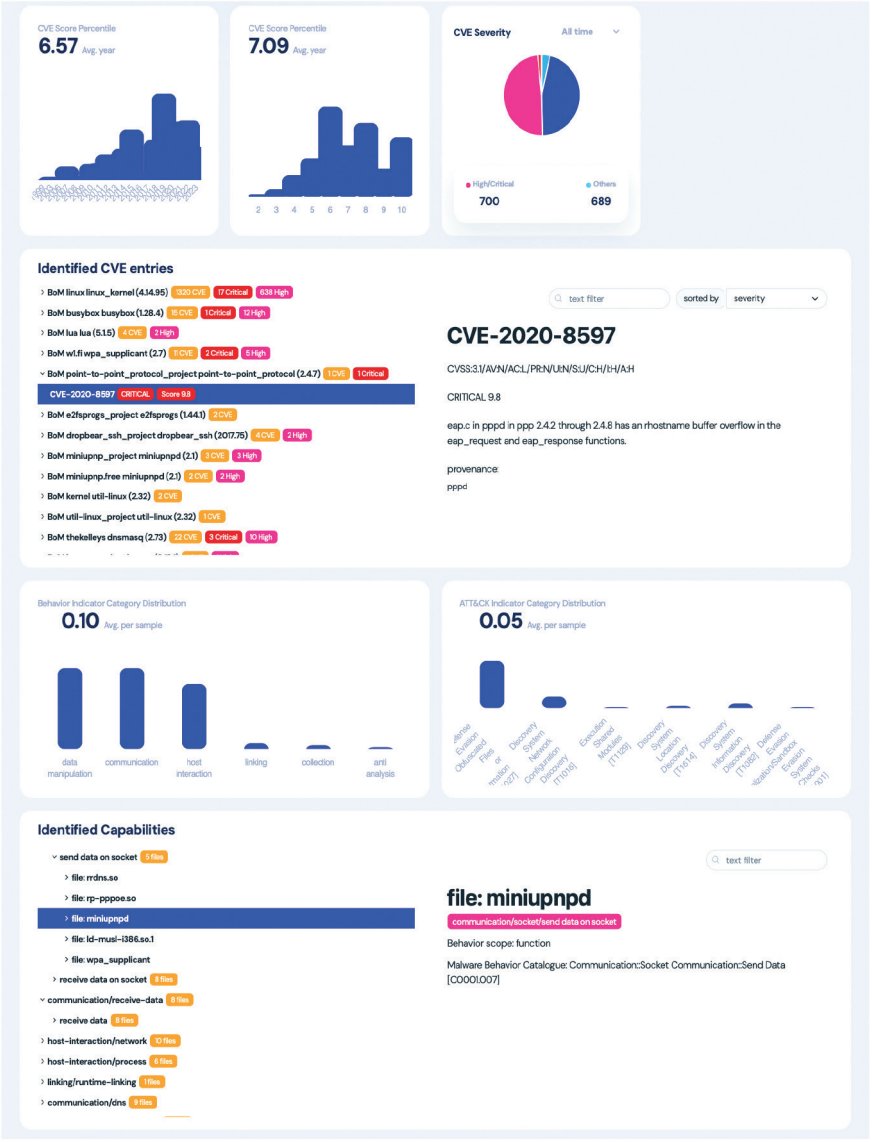
ERS0 organizes firmware images based on the concept of a repository. A repository, in this context, embodies a collection of firmware images earmarked for management and analysis. It is essential to note that each repository functions independently from the others. These repositories are exclusively owned by the current users and upcoming features will enable access sharing.

The repository dashboard, shown in Figure 6, provides a high-level view of the security analysis within a repository of firmware images. Key metrics here include the number of binaries extracted (21,835), instances of CVE vulnerabilities (4,697), and the prevalence of high-severity vulnerabilities (48% of the total). The dashboard highlights the average severity score of CVEs (6.94), an increase in average severity score (+2.45% compared to some baseline), and the criticality of vulnerabilities, with 129 identified as critical severity (3% of the total). It also offers historical context with a line graph showing the average age of CVEs spanning over five years, suggesting how quickly vulnerabilities are being identified and potentially addressed. An analysis of behavior categories indicates a focus on root-level actions, while the ATT&CK tactics charts categorize observed behaviors and their sources, which are critical for understanding attack patterns and planning defenses.

FIGURE 6: ERS0 REPOSITORY DASHBOARD

Given a firmware image in the repository, the user can open its corresponding latest SBOM analysis report, as illustrated in Figure 7. The SBOM and vulnerability report provide an in-depth analysis of the vulnerabilities present in the system's software components. The CVE score percentile graphs illustrate the distribution of vulnerability severities over time, with an average yearly score of 7.10, pointing to the presence of significant security risks. The CVE severity pie chart reflects this by showing that high/critical vulnerabilities constitute a substantial portion (50 out of 83 total instances). The report lists several specific CVE entries, such as a critical vulnerability in the "BoM point-to-point_protocol_project" and a high vulnerability in "BoM lua." These entries are categorized by severity and offer actionable intelligence for prioritizing patches and remediation efforts. The data suggests a need to continuously monitor and update software components to mitigate these identified vulnerabilities effectively.

FIGURE 7: ERS0 ANALYTIC REPORT



The capability report delves into the specific functions and potential risks associated with files within the system. It categorizes behaviors into areas such as data manipulation, communication, and host interaction, with data manipulation being the most prevalent, averaging 0.10 instances per sample. This is exemplified by 35

files having data manipulation/encoding capabilities, with 27 of these using XOR encoding—a common technique for obfuscating data to evade detection. The file named "signify" is specifically identified as employing this technique, aligning with the Malware Behavior Catalog's [20] Defense Evasion category and the ATT&CK framework's T1027 indicator for obfuscated files or information. Such insights are crucial for identifying files that may pose a security risk using sophisticated obfuscation or evasion methods.

6. CONCLUSION

ERS0 offers an effective enhancement to firmware security and asset management processes. By integrating AI and large-scale language models, it provides an improved method for constructing SBOMs that can scale to accommodate a wide variety of string patterns and library signatures. The preliminary results suggest that ERS0 may offer more thorough SBOM coverage and a more accurate identification of vulnerabilities, while also bringing a broader set of features to the table. Its performance, when compared with open-source alternatives, indicates that it has the potential to support military operations with a scalable and efficient tool for managing firmware assets and securing the supply chain. Moving forward, ERS0 could play a supportive role in the continuous improvement of cyber defenses, catering to the evolving needs of military technology and infrastructure.

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] S. Cho, E. Orye, G. Visky, and V. Prates, *Cybersecurity Considerations in Autonomous Ships*. Tallinn: CCDCOE, 2022.

[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] Fraunhofer FKIE. "fkie-cad/FACT_core: Firmware analysis and comparison tool." GitHub.com. Accessed: Mar. 12, 2024. [Online]. Available: https://github.com/fkie-cad/FACT_core

[5] Intel. "intel/cve-bin-tool: The CVE binary tool." GitHub.com. Accessed: Mar. 12, 2024. [Online]. Available: https://github.com/intel/cve-bin-tool

[6] "MITRE ATT&CK®." MITRE. Accessed: Mar. 12, 2024. [Online]. Available: https://attack.mitre.org/

[7] E-M-B-A. "e-m-b-a/embark: EMBArk—The firmware security scanning environment." GitHub.com. Accessed: Mar. 12, 2024. [Online]. Available: https://github.com/e-m-b-a/embark

[8] "unblob—extract everything!" Unblob.org. Accessed: Mar. 12, 2024. [Online]. Available: https://unblob.org/

[9] National Security Agency. "Ghidra." Ghidra-SRE.org. Accessed: Mar. 12, 2024. [Online]. Available: https://ghidra-sre.org/

[10] "Hex Rays—State-of-the-art binary code analysis solutions." Hex-Rays.com. Accessed: Mar. 12, 2024. [Online]. Available: https://hex-rays.com/ida-pro/

[11] Mandiant. "Mandiant/capa: The FLARE team's open-source tool to identify capabilities in executable files." GitHub.com. Accessed: Mar. 12, 2024. [Online]. Available: https://github.com/mandiant/capa

[12] J. H. Clark, D. Garrette, I. Turc, and J. Wieting, “CANINE: Pre-training an efficient tokenization-free encoder for language representation,” *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 73–91, 2022.
[13] D. Chicco, “Siamese neural networks: An overview,” *Artificial Neural Networks*, vol. 2190, pp. 73–94, 2021.
[14] 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*.
[15] W. W. Peterson and D. T. Brown, “Cyclic codes for error detection,” *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.
[16] VirusTotal. “YARA—The pattern matching Swiss knife for malware researchers.” GitHub.com. Accessed: Mar. 12, 2024. [Online]. Available: https://virustotal.github.io/yara/
[17] P. Lewis et al., “Retrieval-augmented generation for knowledge-intensive NLP tasks,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.
[18] Red Canary. “Atomic-red-team: Small and highly portable detection tests based on MITRE’s ATT&CK.” GitHub.com. Accessed: Mar. 12, 2024. [Online]. Available: https://github.com/redcanaryco/atomic-red-team
[19] T. Nowak. “NTAPI undocumented functions.” NTinterlnals.net. Accessed: Mar. 12, 2024. [Online]. Available: http://undocumented.ntinternals.net/
[20] MITRE. “Malware behavior catalog (version 3.0).” GitHub.com. Accessed: Apr. 8, 2024. [Online]. Available: https://github.com/MBCProject/mbc-markdown