Automating Firmware Vulnerability Triage via High-Level Representations and Similarity Digests

Daniel Huici
University of Zaragoza
Zaragoza, Spain
dhuici@unizar.es

Narges Yousefnezhad
Binare Oy
Jyväskylä, Finland
narges.yousefnezhad@binare.io

Ricardo J. Rodríguez
University of Zaragoza
Zaragoza, Spain
ricardo@unizar.es

Andrei Costin
University of Jyväskylä
Jyväskylä, Finland
ancostin@jyu.fi

***Abstract*—Tracking N-day vulnerabilities in fragmented firmware ecosystems is an open challenge, often hampered by the disconnect between abstract CVE descriptions and the binary code actually distributed in production and connected devices. In this paper, we present a generic CVE-based framework for correlating vulnerable files in heterogeneous firmware images using similarity digests. Our approach leverages APOTHEOSIS, an open-source approximate nearest neighbor search system, to scale similarity queries across massive collections of artifacts. To bridge the semantic gap between vulnerability reports and binary reality, we introduce an automated process that lifts confirmed vulnerable implementations to high-level intermediate representations and generates function-level search signatures. We demonstrate the effectiveness of this system as a rapid triage tool using the OpenWrt ecosystem as a case study. In the event of a new CVE disclosure, our approach allows analysts to consult the pre-created APOTHEOSIS index to immediately generate a prioritized list of affected firmware versions, significantly accelerating impact assessment without being dependent on reliable nor accurate vendor/CVE metadata or source code.**

I. Introduction

Software vulnerabilities continue to pose a critical risk to modern computing systems, particularly in ecosystems characterized by extensive code reuse and long software lifecycles [1]. Open-source software (OSS) components are widely reused across operating systems, firmware images, and embedded platforms; however, this ubiquity comes with limited visibility into how vulnerabilities propagate across versions and derivatives [2], [3], [4]. While the Common Vulnerabilities and Exposures (CVE) framework provides a standardized catalog of known vulnerabilities, CVE records typically rely on very high-level textual and metadata descriptions that lack machine-actionable and/or code-level representations. This creates a huge and challenging semantic gap that prevents the fast and quite importantly, *reliable yet narrow*, identification and isolation of vulnerable binaries or components in the wild based on CVE data alone [5].

This problem is intensified in the IoT and embedded space, where devices frequently run aging firmware and receive infrequent or no updates [6]. Vendors often customize upstream distributions without consistently propagating security fixes, effectively breaking the link between version numbers and security status. Consequently, publicly known vulnerabilities often persist as N-day or 1-day issues long after fixes are available upstream [7], [8]. This challenge is particularly acute in distributions such as `OpenWrt`, where thousands of binaries are built across versions, architectures, and configurations. In such fragmented firmware environments, traditional version-based scanning fails, allowing vulnerabilities to persist due to incomplete patch adoption or unnoticed reuse of vulnerable components [9].

To address these limitations, binary code similarity analysis has emerged as a practical and scalable technique [10]. By comparing compiled code rather than relying on source-level artifacts or unreliable version strings, similarity techniques can detect reused (or related) components even when source code is unavailable or significantly altered by compiler optimizations.

However, while the underlying similarity algorithms are well established, their application to effective N-day triage at the scale of an entire software distribution remains an open challenge. In this paper, we reframe this task not as a simple metadata verification step, but as a systematic code correlation problem: *determining whether a specific vulnerable logical pattern manifests itself in a massive, heterogeneous corpus*.

Contributions. We present an operational CVE-driven, similarity-based approach to find and triage `OpenWrt` firmware images potentially affected by specific vulnerabilities. While we practically demonstrate this on `OpenWrt` releases, the methodology is generic and designed to be applicable to general-purpose Linux-based firmware ecosystems or other distributions. In particular, our contribution is three-fold:

- We leverage APOTHEOSIS [11], [12], an open-source approximate nearest-neighbor (ANN) search system, to perform efficient similarity queries across massive collections of firmware artifacts. Instead of relying on exact matches, we compare the code actually shipped in firmware using function-level similarity digests to locate vulnerable code within heterogeneous images.
- We introduce a pipeline that automates the *code lifting* of confirmed vulnerable implementations from upstream binary sources to intermediate code representations. The

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.23027
www.ndss-symposium.org

pipeline then generates function-level digests from these representations, which serve as search signatures, effectively bridging the gap between abstract (and mostly text-only) CVE/bug descriptions and practical realities of binary code.
- We demonstrate how this system serves as a rapid and practical triage tool. Upon the disclosure of a new CVE (or internal 0-day discovery), an analyst can query an index (`OpenWrt` in our case) to immediately generate an ordered list of potentially affected firmware versions. This allows for a faster determination of the impact and creates the necessary context for a specific and deeper manual analysis.

Paper Organization. The remainder of this paper is organized as follows. Section II describes the most important concepts necessary to understand our approach to binary similarity and indexing. Section III details our experimental methodology, ranging from the selection of `OpenWrt` versions and CVEs to the construction and querying of the APOTHEOSIS index. Section IV presents an evaluation using eight CVEs in `OpenWrt` versions from 18.06.0 to 24.10.4, including hierarchical correlation results, recovery curves, and architecture-specific performance metrics. Section V assesses the operational feasibility of the workflow and addresses current limitations. Section VI reviews related work. Finally, Section VII concludes the paper and outlines future work.

II. Background & Terminology

In this section, we introduce the fundamental concepts required to understand our approach, define the scope of the problem, and describe the `OpenWrt` ecosystem and analysis corpus. Finally, we define the main technical components: function-level similarity summaries, the intermediate representation used to derive them, and APOTHEOSIS, the ANN system used in this work.

A. Scope of the Problem

This work focuses primarily on N-day vulnerabilities: publicly known security flaws (assigned a CVE) that are typically accompanied by a patch or mitigation in the original project. Despite the availability of patches, these vulnerabilities frequently persist in deployed systems due to delayed updates, unmaintained code forks, incomplete or inaccurate CVE/CPE metadata, as well as silent rollbacks where version metadata remains unchanged.

While our present assessment and evaluation focuses on N-day discovery, the methodology is independent of the vulnerability's disclosure status. The same approach applies to zero-day vulnerability classification: once a previously unknown weakness (such as zero-days) in a specific function is identified, our approach can quickly correlate such functions across a massive corpus to assess the full impact even before a patch is designed.

Let us remark that in this particular work we treat vulnerability assessment as a code correlation problem, rather than a metadata verification task. The primary goal is not to verify (declarative) version chains, but to determine if and where a specific vulnerable code pattern physically resides within a heterogeneous landscape of firmware and derivative versions.

B. The `OpenWrt` Ecosystem

`OpenWrt` is an open-source operating system based on Linux, designed for embedded devices and network equipment. It is distributed as firmware images that bundle the kernel, build system, package management system, and user-space packages into a cohesive unit.

We selected `OpenWrt` as our primary case study because it exemplifies the supply chain fragmentation inherent in the embedded development environment. The ecosystem extends far beyond its official releases, serving as the upstream foundation for a wide range of commercial IoT vendors and community-driven forks. Consequently, vulnerabilities identified at this root of the supply chain root do not remain isolated but propagate downstream into vendor-locked firmware and custom device derivatives. This amplification effect means that a single unpatched flaw in the main `OpenWrt` distribution can silently compromise thousands of derivative device models where traditional patch management is often nonexistent.

In this fragmented environment, reliably identifying the exact base version is often impossible. Firmware images often lack reliable version chains, and despite GPL license requirements, build recipes are rarely published (if at all) in a fully reproducible format. This lack of supply-chain provenance often undermines source-level auditing and renders version-based analysis ineffective, necessitating a purely binary approach to correlate code between official versions and later targets.

Finally, from an experimental perspective, `OpenWrt` provides an ideal reference database. Its open-source nature, with accessible commit histories, changelog, and patch metadata, allows us to compile and verify accurate vulnerability states. This creates a traceable testing ground where the accuracy of our similarity-based recovery can be rigorously measured against known source-level data before being applied to black-box scenarios.

C. Similarity Digest Algorithms

Similarity Digest Algorithms (SDAs) are approximate matching techniques [13] used to quantify similarity between digital artifacts. These algorithms operate at the byte-level (i.e., processing the input as a raw stream without any syntactic or semantic interpretation) to generate a compact intermediate representation known as a digest or fingerprint. Unlike cryptographic hash functions, which are designed to exhibit the avalanche effect [14] (where a single-bit change results in a drastically different output), SDAs possess a smoothness property. They are designed so that small perturbations to the input result in (ideally) proportionally small changes in the digest distance.

While several families of SDAs exist, ranging from feature sequence hash to byte sequence existence hash [15], in this paper we use Trend Micro's Locality Sensitive Hashing (TLSH) [16]. TLSH generates a fixed-length digest based on

a processed sliding window of the input stream. The digests are compared using a specialized distance metric, where a low score indicates greater similarity. We selected TLSH for two main reasons. First, its fixed-size representation (typically 70 characters) allows for uniform storage and efficient integration into the metric space, unlike variable-length digests (e.g., `ssdeep`), which are difficult to index efficiently. Second, previous evaluations indicate that TLSH offers a superior balance between detection accuracy and search speed compared to alternative digests such as `sdhash` or `Nilsimsa`, making it particularly well-suited for large-scale nearest neighbor search tasks [17].

D. High-Level Intermediate Representation

To perform effective similarity analysis, raw binary code must be normalized. Applying similarity hashes directly to raw bytes is often unreliable; previous work has shown that approximate matching has high rates of false positives and false negatives due to architectural differences such as instruction encoding, changes in register allocation, and compiler optimizations [18], [19], [20].

To address this, we rely on intermediate representations to elevate the code to a stable semantic level. Specifically, we adopt High-Level Intermediate Language (HLIL) produced by the `Binary Ninja` decompiler [21]. HLIL abstracts low-level architectural details (such as CPU-specific registers or stack pointer arithmetic) and reconstructs the program's control flow and variable logic into a tree structure similar to the source code. By generating summaries from this normalized HLIL stream instead of raw assembly, our approach becomes resilient to minor syntactic variations and instruction reordering, allowing SDA to capture the function's intrinsic logic.

E. APOTHEOSIS: An Approximate Similarity Search System

Our workflow requires querying specific vulnerability signatures across a massive and continuously growing dataset of firmware functions. A naïve linear scan ($O(N)$ complexity) is computationally prohibitive at the scale of entire operating system ecosystems. Therefore, we model vulnerability correlation as an Approximate Nearest Neighbor (ANN) search problem.

To efficiently execute these queries, we use APOTHEOSIS [11], [12], a specialized similarity search system designed for high-dimensional summary data. APOTHEOSIS is designed as a fully modular framework, allowing complete configuration of both the underlying SDA and the associated metadata schema. Furthermore, it functions as an approximate search system, meaning it avoids the prohibitive cost of exhaustive linear analysis by using probabilistic graph traversal to quickly retrieve the closest matches with high probability, rather than mathematical certainty. Specifically, it employs a hybrid indexing strategy to optimize both accuracy and retrieval: (i) it uses a radix tree [22] to instantly identify identical digests; and (ii) for non-identical matches, it relies on a Hierarchical Navigable Small World (HNSW) graph [23]. HNSW constructs a multi-layered graph structure where the upper layers serve as fast highways for traversing long distances in metric space, and the lower layers enable precise local navigation.

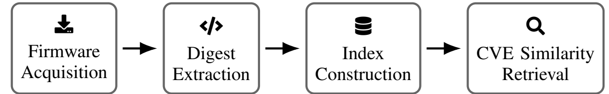


Fig. 1. Overview of our proposed analysis pipeline. The workflow encompasses everything from acquiring the firmware corpus to extracting function-level similarity digests (TLSH in this work). These summaries are then indexed using APOTHEOSIS to enable rapid, similarity-based retrieval and triage of potentially vulnerable firmware images.

This architecture allows APOTHEOSIS to perform queries with logarithmic complexity ($O(\log N)$). When a query summary (representing a vulnerable function) is submitted, the system traverses the graph to return the K nearest neighbors based on the TLSH distance. These neighbors are then mapped back to their metadata (firmware version, binary name), allowing us to instantaneously transition from a single function signature to a list of potentially affected devices.

III. Methodology

In this section, we detail the experimental framework and technical process designed to evaluate our similarity-based triage approach. We begin by introducing our proposed analysis pipeline. Next, we define the `OpenWrt` firmware corpus and the criteria used to select a representative set of N-day vulnerabilities for evaluation. We then describe the automated analysis process, which handles firmware unpacking, binary lifting to HLIL, and the generation of function-level TLSH digests. Finally, we explain the construction of the APOTHEOSIS index and the query methodology used to correlate known vulnerability signatures with the indexed ecosystem.

A. Our Approach

The operational workflow of our approach, illustrated in Fig. 1, comprises four stages: (i) the systematic acquisition and unpacking of firmware images (*Firmware Acquisition*); (ii) the extraction and TLSH-based hashing of function-level signatures from the target binaries (*Digest Extraction*); (iii) the construction of an APOTHEOSIS index using these summaries (*Index Construction*); and (iv) the triage phase, where confirmed vulnerable functions are lifted, hashed, and queried against the index to identify related artifacts across the corpus (*CVE Similarity Retrieval*).

B. `OpenWrt` Corpus Selection

Our assessment corpus covers the main series of stable `OpenWrt` releases, from version 18.06.0 to 24.10.4 (the latest stable release at the time of writing). This dataset encompasses both x86 and ARM architectures to assess the resilience of cross-architecture similarity. We excluded MIPS (despite it being a common target for `OpenWrt`) because `Binary Ninja` could not reliably convert a significant portion of

MIPS binaries to valid HLIL, thus introducing systematic noise into the resulting digests.

We selected version 18.06.0 as the lower bound because it marks the first stable release following the OpenWrt/LEDE merger, providing a consistent technical foundation for all subsequent "modern" OpenWrt releases. Furthermore, this period coincides with vulnerability disclosures in official *OpenWrt Security Advisories*, ensuring that our corpus contains the specific vulnerability intervals necessary for validation [24].

C. CVE Selection Criteria

To evaluate the system's effectiveness in real-world triage scenarios, we applied a selection filter to 23 CVEs listed in the *OpenWrt Security Advisories* [24]. We prioritized vulnerabilities that satisfied the following three criteria: (1) the vulnerability affects core system libraries or daemons (e.g., libubox, uhttpd, dnsmasq) that are present by default in almost all OpenWrt images, ensuring a sufficient sample size for detection; (2) the vulnerable components are compiled for multiple architectures (e.g., they are not restricted to a specific hardware driver), allowing us to test the robustness of HLIL-based similarity across instruction sets and hardware architectures; (3) logical errors reside within the user space of OpenWrt and custom middleware (rather than being generic Linux kernel flaws) as these vulnerabilities are usually most prone to silent rollback and supply chain fragmentation.

We applied these criteria to the full set of 23 vulnerabilities reported in OpenWrt Security Advisories during the target period. We excluded 15 candidates for three main reasons: they affected project infrastructure (e.g., server configurations or build systems) rather than distributed firmware images; they involved hardware-specific drivers not available on all architectures; or they lacked sufficient public documentation to accurately identify the vulnerable function for ground-truth verification.

Based on these criteria, we selected eight vulnerabilities for our assessment, which are detailed in TABLE I. This selection covers a wide range of security flaws, including critical stack buffer overflows, access control bypasses, and integrity violations, affecting core system components such as uhttpd, pppd, and dnsmasq. For each entry, the table provides the CVE identifier, the affected software package, the specific vulnerable function analyzed, the vulnerability type, and the associated CVSS severity score to illustrate the potential impact on the ecosystem.

D. Automated Firmware Analysis Pipeline

To ensure reproducibility and scalability, we developed an automated pipeline that orchestrates the entire analysis lifecycle. The process begins with the systematic retrieval of firmware images for all target version series. Each image undergoes recursive extraction to handle nested compression layers and expose the root filesystem.

From each extracted filesystem, the pipeline identifies the target binaries associated with the selected CVEs and sends them to Binary Ninja for static, non-GUI analysis. For each function within a binary, our analysis pipeline: (i) lifts the machine code to HLIL representation; (ii) serializes the HLIL structure into a normalized text stream; (iii) calculates a TLSH digest from this stream; and (iv) associates the digest with contextual metadata, including firmware version, architecture, binary name, and function address. The end result is a consolidated JSON dataset containing function-level digests for the entire corpus, structured for immediate indexing.

To promote transparency and facilitate further research on firmware ecosystems, we have released all components of this work as open artifacts. The complete reproduction package is available online and comprises the complete collection of analyzed firmware images, the automated scripts for downloading, extraction, and binary analysis used to build the APOTHEOSIS index, the set of manually extracted vulnerable function digests, and the full analysis output[1].

E. Index Construction and CVE Similarity Retrieval

Using the consolidated dataset, we built the APOTHEOSIS index. The system ingests the JSON records, instantiating nodes that encapsulate both the TLSH digest and its associated metadata.

To optimize retrieval, as discussed in Section II-E, APOTHEOSIS builds a hybrid data structure comprising a radix tree for exact deduplication and a HNSW graph for approximate search. For the evaluation presented in this work, we configured the HNSW index with neighbor connectivity parameters $M = 32$ and $M_0 = 64$ (for the base layer), and a search traversal depth of $ef_{search} = 64$. These parameters were empirically selected to maximize retrieval recall while minimizing computational overhead, resulting in consistent sub-second query latencies that facilitate interactive, real-time triage.

To evaluate the system, we manually ground-truthed the query signatures. This design reflects our intended operational workflow, which assumes that the analyst has a confirmed vulnerable reference function (e.g., from a patch diff or upstream source) to initiate the search. For each of the selected CVEs, we identified the specific vulnerable function (and its patched counterpart, when applicable) in the original firmware image, and extracted their HLIL-based TLSH digests. These ground-truth signatures were then queried to the APOTHEOSIS index. Our system returns the K nearest neighbors ordered by TLSH distance, thus generating a prioritized list of firmware images likely to contain the vulnerability.

IV. Results

In this section, we evaluate the effectiveness of our approach for correlating N-day vulnerabilities across the fragmented OpenWrt ecosystem. As described in Section III-B, our analysis corpus spans the entire lineage of major stable OpenWrt releases, from version 18.06.0 to the latest 24.10.4. This collection comprises a total of 98 distinct firmware images, representing a longitudinal cross-section of the distribution's

[1] See https://zenodo.org/records/18672121

TABLE I
DETAILS OF THE SELECTED CVES USED FOR EVALUATION.

CVE ID	Component (CPE)	Vulnerability Type	CWE	CVSSv3x	Vulnerable Function	Impact
CVE-2019-5102	libustream-ssl	Cert Validation Bypass	CWE-295	5.9	ustream_ssl_check_conn	MitM
CVE-2019-19945	uhttpd	Stack Buffer Overflow	CWE-125, CWE-218	7.5	client_parse_header	DoS / RCE
CVE-2020-7982	opkg	SHA-256 Integrity Bypass	CWE-345, CWE-754	8.1	checksum_hex2bin	Package Spoofing
CVE-2020-8597	pppd	Stack Buffer Overflow	CWE-120	9.8	eap_request	RCE
CVE-2020-28951	libuci	Heap Buffer Overflow	CWE-416	9.8	uci_import	DoS
CVE-2020-25684	dnsmasq	DNS Cache Poisoning	CWE-358	3.7	reply_query	DNS Spoofing
CVE-2020-25686	dnsmasq	DNS Cache Poisoning	CWE-290, CWE-358	3.7	get_new_frec	DNS Spoofing
CVE-2025-62526	ubusd	Heap Buffer Overflow	CWE-122	7.8	ubusd_alloc_event_pattern	RCE

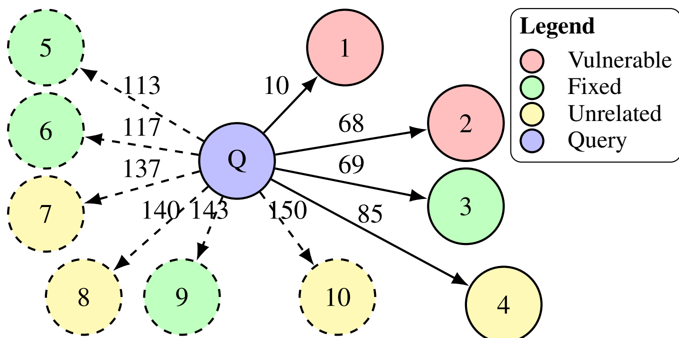


Fig. 2. APOTHEOSIS nearest neighbor visualization for a query function affected by CVE-2020-8597 (pppd). The system successfully groups other vulnerable versions (Nodes 1 and 2) and the patched variant (Node 3) at a close distance, while unrelated functions (Node 4) appear farther away. For completeness, the figure also shows the remaining neighbors recovered during our experiments (indicated by dashed nodes).

evolution on x86 and ARM architectures. From this dataset, we extracted and analyzed the binaries associated with our target CVEs, resulting in a total of 134,328 analyzed functions, which were decomposed into 19,360 unique TLSH digests. These unique digests form the population of the APOTHEOSIS index used in our experiments.

To demonstrate the querying logic, in Fig. 2 we present a practical example with a pppd function affected by CVE-2020-8597. We queried the APOTHEOSIS index using the digest extracted from the vulnerable version of OpenWrt 18.06.0 and analyzed the ranking of the retrieved neighbors (smaller values indicate a closer match).

Fig. 2 visualizes the local neighbors of the query (Node Q) in the TLSH distance space, where smaller distances indicate greater similarity. As expected, the two closest neighbors (Nodes 1 and 2, with distances of 10 and 68, respectively) correspond to vulnerable instances of the same function in different versions of OpenWrt. This confirms that the digest remains stable between versions 18.06 and 19.0. The remaining neighbors are either patched variants that retain substantial structural similarity to the vulnerable code (e.g., Node 3, distance 69), or semantically unrelated functions that appear only at larger distances (e.g., Node 4, distance 85 and above). This separation aligns with the expected outcome for this query (two vulnerable results followed by patched or unrelated results), suggesting that simple distance-based filtering can effectively prioritize vulnerable candidates while pushing fixed or unrelated functions further down the ranking.

In what follows, we first detail the experimental environment and hardware specifications used in this paper. Next, we analyze the system's recovery effectiveness using two metrics: ranked correlation, which assesses the semantic accuracy of returned neighbors, and RecallK, which measures the completeness of vulnerability recovery. Finally, we examine the robustness of HLIL-based signatures across different instruction sets to assess cross-architecture correlation limits.

A. Experimental Settings

All experiments were performed on a workstation running Debian 12, equipped with an Intel Core i7-10700 processor and 64 GiB of RAM. The analysis workflow used Binary Ninja 5.2.8722 for HLIL lifting and Python 3.9 for automation. For the indexing backend, we used APOTHEOSIS configured with the HNSW parameters defined in Section III-E. The total processing time for the entire OpenWrt target-corpus (downloading, extraction, lifting, and indexing) was approximately $1\,\mathrm{h}$, with an average query latency of $0.15\,\mathrm{ms}$ per function.

B. Ranked Correlation Analysis

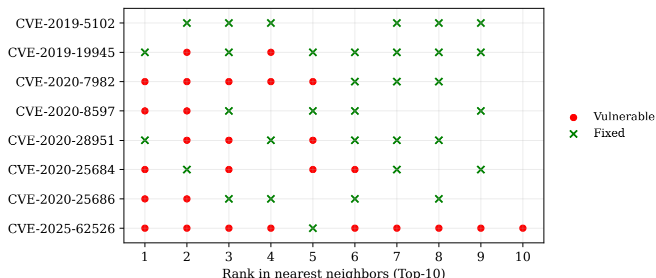


Fig. 3. Hit-rank raster (Top-10) for same-architecture retrieval (x86 queries vs. x86 targets). Each row corresponds to a specific CVE query, showing its first 10 retrieved neighbors: red circles indicate confirmed vulnerable instances, green crosses denote corrected (patched) instances, and empty spaces represent matches with completely unrelated functions.

Fig. 3 visualizes the retrieval performance for each CVE query. To establish a clear performance baseline independent of instruction set variations, we focused solely on retrieval from the same architecture (specifically, x86 to x86). The plot is structured as a raster scan where the x-axis represents the rank k (1–10) returned by APOTHEOSIS.

The results demonstrate a clear semantic clustering capability. For most queries, vulnerable instances (red circles) are concentrated in the uppermost ranks ($k = 1 \ldots 6$). This confirms that the digest distance effectively prioritizes functionally identical or very similar code. Patched instances (green crosses), which share the majority of their logic with the vulnerable version but contain the patch, are typically interspersed in the lower ranks ($k = 4 \ldots 10$). This separation is significant for the analyst's triage, as it indicates that the system not only finds *identical functions* but also traverses a fluid semantic range. The presence of the patched version in the immediate vicinity allows the analyst not only to identify vulnerable targets but also to potentially verify the presence of a patch by inspecting immediate neighbors.

Two extreme cases in Fig. 3 require explanation. For CVE-2019-5102, the corresponding row shows no vulnerable matches. While our dataset covers the range of affected versions (18.06.0 to 18.06.4), the vulnerable library (`libustream-ssl`) was not included in the default `OpenWrt` images until version 21.02.0. Consequently, the vulnerable component is completely absent from our corpus, resulting in zero ground-truth instances. We retain this case as a negative control to validate that the system correctly distinguishes between patched and missing code, demonstrating its robustness against the dynamic addition or removal of dependencies throughout the ecosystem lifecycle. Conversely, the CVE-2025-62526 row is densely populated with vulnerable matches because the flaw affects `ubusd`, a core daemon present in all versions. Since the fix was only applied in the latest version of the service (24.10.4), almost all the neighbors retrieved from the historical corpus are vulnerable instances.

C. Recall Performance

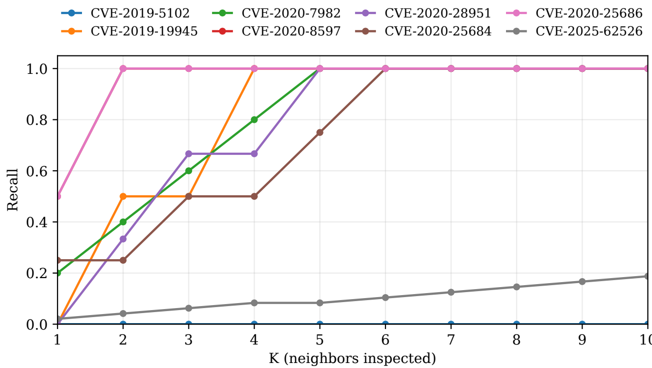


Fig. 4. Recall@K curves. The fraction of the total known vulnerable instances recovered within the Top-K candidates.

Fig. 4 shows the Recall@K metric, defined as the fraction of the total set of ground-truth instances T recovered within the first K results. Formally, $\text{Recall@}K = H(K)/T$, where $H(K)$ is the number of vulnerable matches in the first K results.

For most CVEs, the recovery curve quickly approaches 1.0 within a small neighborhood (i.e., generally within $k \leq 6$). This implies that inspecting a short, minimal list is sufficient to recover the entire lineage of vulnerable artifacts. Nevertheless, for widespread vulnerabilities such as CVE-2025-62526, the curve remains linear and low, even for larger k values. This is a mathematical inevitability rather than a recovery failure: the total set of vulnerable instances in the corpus is large ($T = 48$), meaning that even a "perfect" recovery of 10 vulnerable neighbors results in a maximum possible recovery of only $10/48$ ($\approx 35.71\%$). In triage scenarios, this behavior is acceptable since the goal is often to find at least one high-confidence starting point for investigation rather than exhaustively enumerating every file in a single query. In subsequent analyst's optimizations, the already triaged/verified versions/functions could be removed or ignored from the query result-set associated with specific over-represented CVE (e.g., CVE-2025-62526), so that additional triage points could be added into the neighboring range and further inspected.

D. Architecture-Specific Robustness

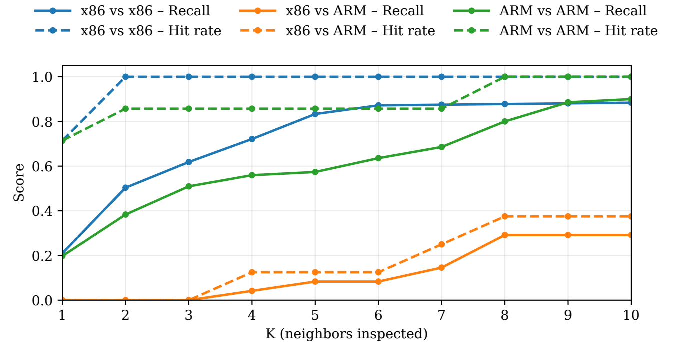


Fig. 5. Comparison of recall and hit rates when the query and target share the same ISA (x86/x86, ARM/ARM) versus cross-ISA queries.

To evaluate the stability of HLIL-based signatures across different instruction sets, we compared retrieval effectiveness between matching architectures (same architecture) and mixed architectures (cross-architectures). For this analysis, we excluded MIPS targets due to limitations observed in the current `Binary Ninja` lifter, where a significant percentage of MIPS functions failed to be lifted to valid HLIL, generating systematic noise that would distort the comparison and make cross-version similarity results hard to interpret fairly.

Fig. 5 highlights two main findings regarding the stability of the architecture. First, queries within the same Instruction Set Architecture (ISA) exhibit consistently high recall, with x86 recall slightly higher than ARM recall. Our hypothesis is that this is due to the nature of the instruction sets: x86 (CISC) instructions are typically mapped more directly to complex, high-level operations; whereas ARM (RISC) sequences can introduce greater variability in data loading depending on compiler optimizations. Second, recall between architectures (e.g., x86 queries versus ARM targets) exhibits a marked decrease in performance. This suggests that while HLIL effectively abstracts registers and op-code mnemonics,

the underlying structural shape of the control flow graph is still influenced by architecture-specific idioms (e.g., calling conventions, conditional execution patterns) that survive the lifting process.

Despite the cross-architecture penalty, the high recall within architecture families confirms that automated pipeline effectively solves the "seed"-vulnerability correlation problem for the most common use case – tracking the spread of vulnerabilities between updates of the same lineage of devices.

Taken together, these results provide a comprehensive demonstration of our approach's capabilities. By automating the analysis of the entire `OpenWrt` lineage, we have shown that the system can scale to real-world ecosystems and reliably recover vulnerability lifecycles and impact surfaces with minimal human intervention. To facilitate research and reproducibility, the complete automation suite and generated datasets are open source, as detailed in Section III-D.

E. Qualitative Analysis: Understanding False Positives

To characterize the operational limits of our approach, we performed a qualitative failure analysis by manually inspecting a subset of false positives: instances where the APOTHEOSIS index retrieved non-vulnerable functions with high confidence (i.e., low TLSH similarity score).

Our analysis reveals that, because TLSH relies on digest comparison, it primarily captures syntactic rather than semantic similarity. Consequently, it is prone to repetitive collision, where distinct functions merge due to shared linear sequences, such as identical error-handling macros, argument validation logic, or standard structural idioms. In these scenarios, the statistical signal of shared instruction sequences overwhelms the unique logic of the target function, causing its digests to converge within the retrieval threshold.

In fact, we observed that this phenomenon is particularly pronounced in two scenarios: (i) in structural idioms, where small utility functions share a rigid control flow structure (e.g., a simple switch-case wrapper) are often assigned to similar hashes despite performing different operations; and (ii) in the patch proximity. Large functions where the vulnerability fix is syntactically minor (e.g., a single conditional check added to a complex function). In such cases, the patched and unpatched versions remain so syntactically similar that the hash function fails to separate them, returning the patched version as the best match for the vulnerable query.

We observed this latter behavior in our analysis of CVE-2020-28951. As illustrated in Figure 6, the system correctly identified the lineage of the function `uci_import` (corresponding to `OpenWrt` from 23.05.0 to 23.05.6), but struggled to distinguish the vulnerable function from structurally similar candidates.

Figure 6(b) highlights the root cause: while the linearized code representation used by TLSH fails to capture the subtle divergence in execution paths, a graph-based representation makes the difference immediately apparent. This observation validates our workflow proposal: fast syntax-based retrieval (via APOTHEOSIS) effectively filters the search space. However, it must be combined with accurate graph-theory-based verification (e.g., using `BinDiff` [25] or similar binary diffing tools) to disambiguate the final candidates.

Exploring this synergy represents a promising avenue for future research. While graph-based verification is computationally prohibitive at the scale of an entire OS distribution, our approach reduces the search space by orders of magnitude, making it operationally feasible to implement high-precision semantic analysis as a post-processing step for automated N-day confirmation.

V. Discussion

In this section, we reflect on the operational implications of our findings. First, we examine the feasibility of the proposed workflow for real-world security operations, focusing on scalability and vendor metadata independence. Then, we address the inherent limitations of the approach, including challenges related to firmware extraction, cross-architecture stability, and the probabilistic nature of similarity-based recovery.

A. Practicality of Our Approach

Our results demonstrate that the proposed workflow is practical for large-scale N-day classification. By reducing the complex vulnerability correlation problem to efficient K-nearest neighbor searches, our approach avoids the prohibitive computational cost of exhaustive pairwise comparisons. This efficiency allows analysts to query massive historical datasets, such as the entire `OpenWrt` lineage, in sub-second time intervals.

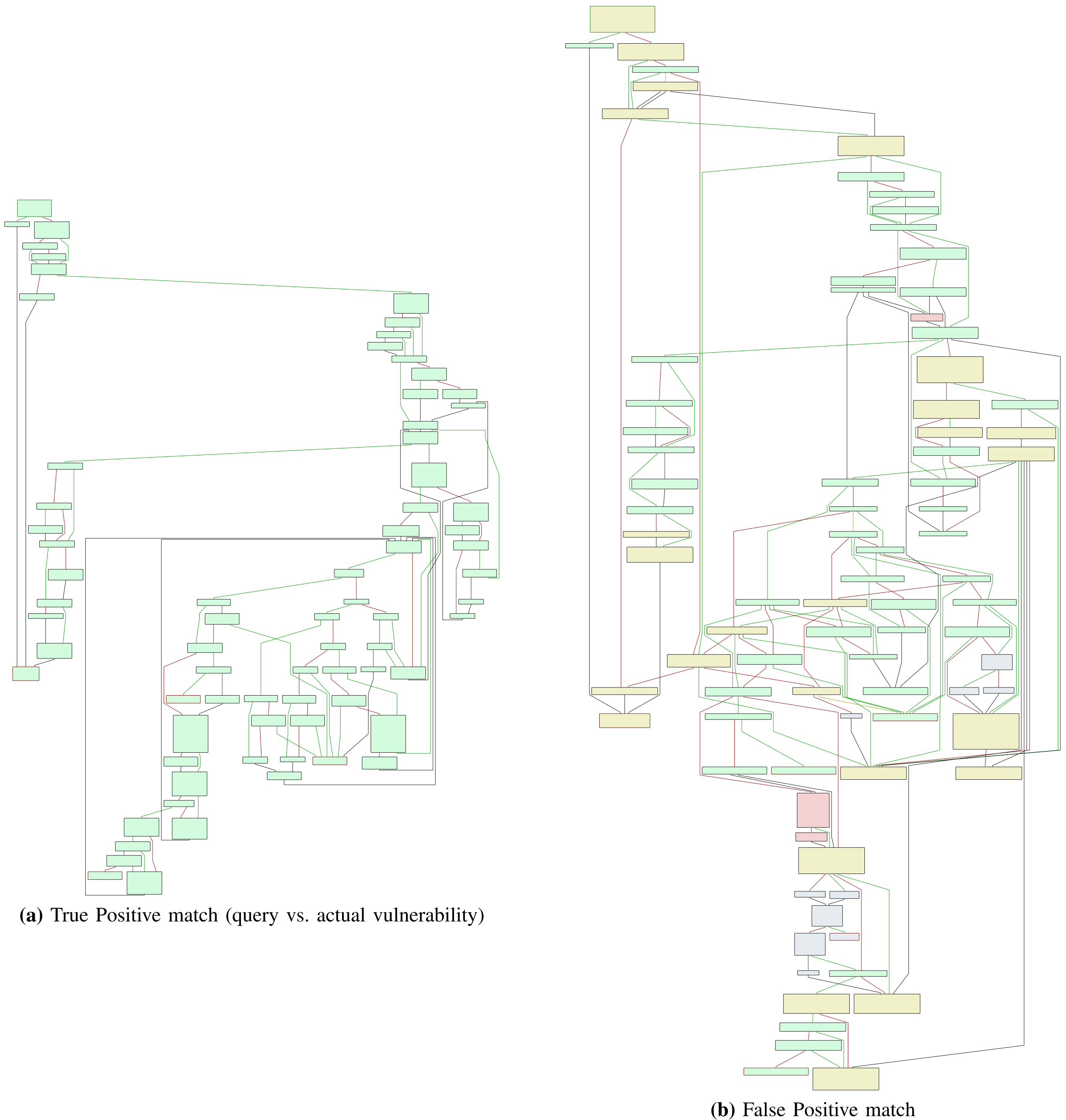
Essentially, the pipeline operates directly on distributed binaries, eliminating reliance on vendor metadata, version chains, or unreliable software bill of material, which are often missing or inaccurate in integrated ecosystems. Because each digest is indexed with complete context (firmware version, architecture, binary path), a positive match immediately identifies not only a similar function but also the specific firmware image and component requiring attention.

This capability translates into a significant operational advantage for impact assessment. Upon disclosure of a new CVE, an analyst only needs to lift and hash the confirmed vulnerable function (or its patch). Querying the index with this single signature instantly generates a prioritized list of potentially affected artifacts, transforming a weeks-long manual audit into a rapid, data-driven triage process.

B. Limitations

While the results confirm that HLIL similarity digest correlation effectively facilitates triage at scale, several technical limitations affect its broader application.

First, the approach assumes that firmware images can be successfully decompressed and the target binaries located. In practice, vendor firmware often employs non-standard, custom packaging, encryption, or compression schemes that impede the effectiveness of generic extraction and processing tools [26]. Furthermore, complex vulnerabilities may not be encapsulated by a single function; if a flaw spans multiple



(a) True Positive match (query vs. actual vulnerability)

(b) False Positive match

Fig. 6. Control Flow Graph comparison for the `uci_import` vulnerable function (CVE-2020-28951). (a) A true positive match, showing the structural identity between the query and a vulnerable instance in a different version. (b) A false positive where TLSH assigned a high similarity score due to shared linear instructions sequences, despite significant topological divergence in the graph structure.

routines or depends on specific build configurations, a single-function signature may not capture the necessary context, necessitating multi-signature query strategies. Additionally, production binaries are frequently stripped of symbol tables, making the initial identification of target functions for querying a non-trivial challenge that may require heuristic signature matching, or non-scalable and tedious manual reverse engineering prior to hashing.

Second, the system's reliability is limited by the quality of the underlying HLIL. As observed in our MIPS experiments, if the lifting process is incomplete or unstable for a specific ISA, the resulting digests become noisy, degrading the accuracy of

the retrieval. Furthermore, variations in compiler optimization levels (e.g., -O0 vs. -O3) or aggressive function inlining can significantly alter the control flow graph, potentially causing a vulnerable function to merge with its caller or become structurally distorted beyond the similarity threshold. This dependency also explains the performance gap in cross-architecture scenarios: despite the abstraction provided by HLIL, architecture-specific idioms (such as calling conventions and register usage patterns) often persist in the graph structure, reducing the effectiveness of cross-ISA matching.

Finally, it is necessary to interpret the result correctly: a similarity score is a probabilistic indicator, not a formal proof of vulnerability. Like all fuzzy hash schemes, TLSH is susceptible to both false positives (matching similarity) and false negatives (divergence due to over-optimization or obfuscation). Since TLSH is not designed to be robust against adversaries, deliberate manipulation could further evade detection. Therefore, retrieved matches should be considered high-priority candidates for validation (requiring manual verification or specific dynamic analysis), rather than definitive judgments of vulnerability.

VI. Related Work

Our research focuses on the intersection of similarity hashing, large-scale binary retrieval, binary diffing/comparison, and automated vulnerability correlation/classification. In this section, we review the evolution of TLSH as a similarity metric, existing binary diffing tools and systems for scalable binary search, and previous methodologies for N-day vulnerability discovery.

A. TLSH-Based Similarity

TLSH has become a standard similarity digest for binary and malware analysis thanks to its design focused on resilience to small input permutations and computational efficiency [16]. Benchmarking against other similarity hashing schemes consistently demonstrates TLSH's favorable balance between robustness and speed [17]. In the specific context of firmware, He et al. [27] validated TLSH's effectiveness on heterogeneous binaries, while Tan et al. [28] extended this validation to cross-architecture scenarios, highlighting its portability across different ISAs.

However, reliance on similarity summaries carries risks. Recent studies by Fuchs et al. [29], [30], [31] and Sándor [32], among others, have demonstrated adversarial attacks targeting TLSH, including techniques to generate collisions or conceal malicious payloads from digest-based scanners. While these findings raise serious concerns for adversarial detection (e.g., active malware blocking), they are less restrictive for our use case: vulnerability triage. We use TLSH not as a security primitive to resist active evasion, but as a probabilistic search mechanism to locate the unintentional recurrence of non-adversarial vulnerabilities in cooperative software ecosystems.

B. Scalable Similarity Digest Search

While TLSH provides the underlying metric, querying large corpora requires the support of specialized systems. Our work is based on APOTHEOSIS [11], an extensible framework specifically designed for high-performance similarity digest search (see Section II-E for more details). This framework enables split-second queries across millions of preprocessed binaries, serving as the architectural backbone of our experiments.

Complementary research has focused on optimizing the underlying search data structures. `NetSHa` [33] proposes network acceleration to scale Local Sensitive Hash (LSH) queries in distributed environments, while `KEENHash` [34] explores function-aware embeddings that combine structural and semantic features to improve resilience against compiler optimizations. While these systems represent an advance in performance and scalability, they remain largely application-independent. In contrast, our work integrates these retrieval capabilities into a vulnerability-centric workflow, driven by CVE data and signatures of confirmed vulnerable functions.

C. Binary Diffing and Operational Triage

Binary diffing focuses on identifying syntactic and semantic differences between two specific binary files. Originally developed to facilitate reverse engineering, patch analysis, and malware lineage tracing, comparison techniques have become essential for vulnerability verification and license compliance [35].

Industry-standard frameworks such as `BinDiff` [25] and `Diaphora` [36] primarily use graph theory heuristics. By calculating structural isomorphism between call graphs and control flow graphs, these tools help analysts isolate security-relevant modifications in patches or updates [37], [38]. `Diaphora` further extends this paradigm by integrating with IDA Pro to support interactive, heuristic-based differentiation workflows [39].

To overcome the limitations of purely structural comparison, which can fail with aggressive compilation optimizations, recent research has explored semantic and behavior-based techniques. Ming et al. [40] proposed `BinSim`, which uses system call tracing and symbolic execution to compare logic across obfuscations. Similarly, hybrid approaches have been proposed to combine structural, semantic, and statistical features, addressing cases where graph topology alone is insufficient [41].

Learning-based methods such as `Gemini` [42], `SAFE` [43], and `Asm2Vec` [44] further improve semantic matching across architectures, but typically incur significant computational overhead during embedding generation and inference.

Learning-based methods have evolved significantly to address semantic gaps. Early approaches such as `Asm2Vec` [44] improved resilience against compiler optimizations, while graph-based systems like `Gemini` [42] and self-attentive models like `SAFE` [43] extended these capabilities to cross-architecture scenarios. More recent Deep Learning (DL) frameworks, such as `VulSeeker` [45] and `CEBin` [46], leverage semantic learning and Transformer-based models to achieve state-of-the-art accuracy. `CEBin`, for instance, combines embedding-based retrieval with precise comparison

to maximize robustness across different instruction sets, but remains subject to model and embedding drift as architectures, compilers, and real-world code bases evolve over time.

However, this accuracy introduces significant operational limitations. DL models typically require computationally intensive training phases, often relying on high-end GPU clusters [46], along with exhaustive fine-tuning on labeled datasets. In contrast, our approach prioritizes operational readiness. By employing a deterministic, untrained hash scheme, we avoid both the cold-start problem inherent in learning-based pipelines and the ongoing retraining burden imposed by drift. This design enables immediate indexing of massive, heterogeneous firmware lineages on standard hardware, thereby eliminating the latency associated with model training and data curation.

Despite their accuracy, binary diffing techniques fundamentally operate in a pairwise verification system: they compare one target binary against a known reference. The reliance on expensive graph matching, symbolic execution, or neural embeddings makes them ill-suited for large-scale discovery tasks, such as correlating a single vulnerable function across thousands of firmware images. Applying these methods at repository scale requires a linear scan of the dataset with a high per-comparison cost, rendering N-day triage operationally infeasible.

In contrast, our approach reframes vulnerability correlation as an approximate nearest-neighbor search problem, prioritizing query latency and scalability over pairwise optimality. By leveraging HNSW indexing, we reduce search complexity to approximately $O(\log N)$, enabling sub-second identification of candidate vulnerable artifacts across massive firmware corpora. Rather than replacing deep semantic or diffing-based analyses, our system acts as a filtering layer that makes such techniques deployable in practice by narrowing the search space to a small, high-confidence candidate set.

D. Vulnerability Discovery and Code Reuse

Automated N-day vulnerability discovery presents a distinct challenge from general bug hunting, as it focuses on identifying the persistence of known flaws due to code reuse or incomplete patching. Approaches such as `V0Finder` [47], `V1SCAN` [9], and `TIVER` [48] employ code classification and clustering to track vulnerabilities in reused C/C++ components. Similarly, Xu et al. [49] use patch semantics to detect unpatched third-party libraries. However, these methods often rely on the availability of source code, patch differences, or rich semantic metadata. These resources, though, are rarely available in the opaque firmware images of embedded devices.

Despite these advances, vulnerability analysis remains prone to errors. As Dann et al. [50] point out, current open-source analysis tools face well-documented challenges, such as high false positive rates, limited coverage of reused components, and low robustness when binaries are modified, over-optimized, or symbols are removed. These limitations are particularly acute in firmware ecosystems, where aggressive compiler optimizations and symbol removal are standard compilation practices.

In the purely binary realm, early systems such as `BinArm` [51] demonstrated scalable detection tailored to specific device classes, while Costin et al. [26] performed the first large-scale analysis of firmware reuse. More recent work by Riom and Bartel [52] on Android libraries and the `DarkWrt` project on `OpenWrt` [53] confirm that long-term code reuse is a key factor in vulnerability persistence. `DarkWrt` specifically highlights how unintended or vulnerable functionality can survive across different firmware versions, evading package-level analysis, thus underscoring the need for function-level granularity.

Our approach addresses the limitations of source-dependent methods and the scalability challenges of binary analysis. By anchoring discovery to TLSH digests of confirmed vulnerable features, we avoid the need for source code or symbols. Furthermore, as Wang et al. [54] point out, digest-based approaches naturally align with privacy preservation requirements, enabling vulnerability correlation without the need to disclose proprietary raw binary artifacts.

VII. Conclusions and Future Work

In this paper, we introduced an analysis pipeline that allows for automating N-day vulnerability classification in fragmented firmware ecosystems. By extracting firmware binaries, lifting to HLIL and indexing TLSH digests at the function level, we enable rapid correlation of vulnerable code across massive firmware datasets without relying on vendor metadata or unreliable source code. This approach effectively bridges the gap between manual reverse engineering and large-scale automated scanning, reducing the complexity of vulnerability tracking to efficient nearest-neighbor searches in a high-dimensional similarity space.

Our evaluation in the `OpenWrt` ecosystem demonstrated that this method achieves high retrieval accuracy for the "same architecture" queries, with the vast majority of known vulnerable instances being retrieved within the top 10 ranked candidates. Although the system exhibits expected limitations in cross-architecture retrieval and maintains sensitivity to retrieval fidelity, it successfully identifies both vulnerable functions and their patched counterparts in real-world scenarios. To encourage reproducibility and further research in binary analysis, we have open-sourced the complete extraction process and the generated firmware datasets.

Future work will focus on improving the resilience of similarity scores to compilation artifacts. We plan to investigate graph-based embedding techniques that directly encode the topological properties of the control flow graph, aiming to improve cross-architecture recovery and robustness against optimization variance. Furthermore, we intend to extend the scope of the analysis beyond compiled code to include interpreted artifacts, such as `Lua` and common shell scripts in embedded web interfaces, to provide a more comprehensive view of the firmware's attack surface.

Acknowledgments

The authors thank: (in alphabetic order) Tuomo Lahtinen, Ahsan Saleem, Hannu Turtiainen for their valuable feedback on early iterations of this work; and the Faculty of Information Technology at the University of Jyväskylä for hosting the research visit of Daniel Huici during 2025.

This research was supported in part by grant PID2023-151467OA-I00 (CRAPER), funded by MICIU/AEI/10.13039/501100011033 and by ERDF/EU, by grant TED2021-131115A-I00 (MIMFA), funded by MICIU/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR, by grant *Proyecto Estratégico Ciberseguridad EINA UNIZAR*, funded by the Spanish National Cybersecurity Institute (INCIBE) and the European Union NextGenerationEU/PRTR, by grant *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-23R), funded by the University, Industry and Innovation Department of the Aragonese Government. (Part of this work was) Funded by the European Union (Grant Agreement Nr. 101120962, RESCALE Project). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the Health and Digital Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

The authors wish to acknowledge CSC – IT Center for Science, Finland, for computational resources. We used Google's Gemini to correct grammatical errors and improve the clarity and coherence of our paper.

References

[1] C. Okafor, T. R. Schorlemmer, S. Torres-Arias, and J. C. Davis, "SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED'22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 15–24.

[2] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[3] K. Blind, M. Böhm, P. Grzegorzewska, A. Katz, S. Muto, S. Pätsch, and T. Schubert, "The impact of Open Source Software and Hardware on technological independence, competitiveness and innovation in the EU economy," Fraunhofer ISI & OpenForum Europe, techreport LC-01346583, SMART 2019/0011, May 2021.

[4] T. Yano and H. Kuzuno, "Visualization Method for Open Source Software Risk Related to Vulnerability and Developmental Status Considering Dependencies," *Journal of Information Processing*, vol. 32, pp. 767–778, 2024.

[5] B. Barchuk and K. Volkov, "Limitations of modern vulnerability scanners and CVE Systems," *World Journal of Advanced Engineering Technology and Sciences*, vol. 12, no. 2, pp. 973–989, 2024.

[6] D. Wang, M. Jiang, R. Chang, Y. Zhou, H. Wang, B. Hou, L. Wu, and X. Luo, "An Empirical Study on the Insecurity of End-of-Life (EoL) IoT Devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 04, pp. 3501–3514, Jul. 2024.

[7] A. Costin, A. Zarras, and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, X. Chen, X. Wang, and X. Huang, Eds. ACM, 2016, pp. 437–448.

[8] P. Deng, L. Zhang, Y. Meng, Z. Yang, Y. Zhang, and M. Yang, "CHAINFUZZ: exploiting upstream vulnerabilities in open-source supply chains," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[9] S. Woo, E. Choi, H. Lee, and H. Oh, "V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 6541–6556.

[10] I. U. Haq and J. Caballero, "A Survey of Binary Code Similarity," *ACM Comput. Surv.*, vol. 54, no. 3, Apr. 2021.

[11] D. Huici, R. J. Rodríguez, and E. Mena, "An extensible and scalable system for hash lookup and approximate similarity search with similarity digest algorithms," *Forensic Science International: Digital Investigation*, vol. 53, p. 301930, 2025.

[12] ——, "APOTHEOSIS: An efficient approximate similarity search system," *SoftwareX*, vol. 29, p. 102016, 2025.

[13] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White, "Approximate Matching: Definition and Terminology," National Institute of Standards and Technology, techreport NIST Special Publication 800-168, May 2014.

[14] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2nd ed. Chapman and Hall/CRC, 2014.

[15] M. Martín-Pérez, R. J. Rodríguez, and F. Breitinger, "Bringing order to approximate matching: Classification and attacks on similarity digest algorithms," *Forensic Science International: Digital Investigation*, vol. 36, p. 301120, Apr. 2021.

[16] J. Oliver, C. Cheng, and Y. Chen, "TLSH – A Locality Sensitive Hash," in *Fourth Cybercrime and Trustworthy Computing Workshop (CTC)*, 2013.

[17] H. Liu, J. Hagen, M. Ali, and J. Oliver, "An Evaluation of Malware Triage Similarity Hashes," in *Proceedings of the 25th International Conference on Enterprise Information Systems, ICEIS 2023, Volume 1, Prague, Czech Republic, April 24-26, 2023*, J. Filipe, M. Smialek, A. Brodsky, and S. Hammoudi, Eds. SCITEPRESS, 2023, pp. 431–435.

[18] F. Pagani, M. Dell'Amico, and D. Balzarotti, "Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: ACM, 2018, pp. 354–365.

[19] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[20] C. Jakobs, A. Mahr, M. Lambertz, M. Rybalka, and D. Plohmann, "Bytewise approximate matching: Evaluating common scenarios for executable files," *Forensic Science International: Digital Investigation*, vol. 53, p. 301927, 2025, dFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA.

[21] Vector 35 Inc., "BNIL Guide: HLIL," [Online; https://docs.binary.ninja/dev/bnil-hlil.html], 2024, accessed on December 19, 2025.

[22] D. R. Morrison, "PATRICIA–Practical Algorithm To Retrieve Information Coded in Alphanumeric," *Journal of the ACM (JACM)*, 1968.

[23] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[24] OpenWrt Project, "OpenWrt Security Advisories," [Online; https://openwrt.org/advisory/start], 2025, accessed on December 12, 2025.

[25] Zynamics, "BinDiff," [Online; https://www.zynamics.com/bindiff.html], 2011, accessed on December 19, 2025.

[26] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-scale Analysis of the Security of Embedded Firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 95–110.

[27] D. He, X. Yu, S. Zhu, S. Chan, and M. Guizani, "Fuzzy Hashing on Firmwares Images: A Comparative Analysis," *IEEE Internet Comput.*, vol. 27, no. 2, pp. 45–50, 2023.

[28] H. Tan, "BCD: A Cross-Architecture Binary Comparison Database Experiment Using Locality Sensitive Hashing Algorithms," *arXiv preprint arXiv:2112.05492*, 2021.

[29] G. Fuchs, R. Nagy, and L. Buttyán, "Targeted Attacks Against the TLSH Similarity Digest Scheme," *IEEE Trans. Inf. Forensics Secur.*, vol. 20, pp. 8922–8935, 2025.

[30] G. Fuchs, T. Fülöp, and R. Nagy, "Concealing targeted attacks on the TLSH similarity digest scheme," in *Proceedings of the Digital Forensics Doctoral Symposium, DFDS 2025, Brno, Czech Republic, 1 April 2025*. ACM, 2025, pp. 3:1–3:7.

[31] G. Fuchs, R. Nagy, and L. Buttyán, "A Practical Attack on the TLSH Similarity Digest Scheme," in *Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES 2023, Benevento, Italy, 29 August 2023- 1 September 2023*. ACM, 2023, pp. 13:1–13:10.

[32] J. Sándor, "Improving the Robustness of Similarity-Based IoT Malware Detection Methods against Adversarial Samples," Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, mathesis, Oct. 2023.

[33] P. Zhang, H. Pan, Z. Li, P. Cui, R. Jia, P. He, Z. Zhang, G. Tyson, and G. Xie, "NetSHa: In-Network Acceleration of LSH-Based Distributed Search," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 9, pp. 2213–2229, 2022.

[34] Z. Liu, Q. Tang, S. Nie, S. Wu, L. F. Zhang, and Y. Tang, "KEENHash: Hashing Programs into Function-Aware Embeddings for Large-Scale Binary Code Similarity Analysis," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, pp. 801–824, 2025.

[35] M. Arutunian, H. Hovhannisyan, V. Vardanyan, S. Sargsyan, S. Kurmangaleev, and H. Aslanyan, "A method to evaluate binary code comparison tools," in *2021 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 2021, pp. 3–5.

[36] J. Koret, "Diaphora: A Free and Open Source Binary Diffing Tool," [Online; https://github.com/joxeankoret/diaphora], 2015, accessed on January 8, 2026.

[37] J. Oh, "Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries," in *BlackHat Technical Security Conference*, 2009.

[38] S. Ullah and H. Oh, "BinDiff$_{\mathrm{NN}}$: Learning Distributed Representation of Assembly for Robust Binary Diffing Against Semantic Differences," *IEEE Trans. Software Eng.*, vol. 48, no. 9, pp. 3442–3466, 2022.

[39] R. Cohen, R. David, R. Mori, F. Yger, and F. Rossi, "Experimental Study of Binary Diffing Resilience on Obfuscated Programs," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 22nd International Conference, DIMVA 2025, Graz, Austria, July 9-11, 2025, Proceedings, Part I*, ser. Lecture Notes in Computer Science, M. Egele, V. Moonsamy, D. Gruss, and M. Carminati, Eds., vol. 15747. Springer, 2025, pp. 223–243.

[40] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 253–270.

[41] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[42] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, 2017, pp. 363–376.

[43] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "SAFE: Self-Attentive Function Embeddings for Binary Similarity," in *Proceedings of the 16th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2019, pp. 309–329.

[44] 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 *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

[45] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE '18)*. New York, NY, USA: ACM, 2018, pp. 896–899.

[46] H. Wang, Z. Gao, C. Zhang, M. Sun, Y. Zhou, H. Qiu, and X. Xiao, "CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. New York, NY, USA: ACM, 2024, pp. 149–161.

[47] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, "V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 3041–3058.

[48] Y. Choi and S. Woo, "Tiver: Identifying Adaptive Versions of C/C++ Third-Party Open-Source Components Using a Code Clustering Technique," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2025, pp. 2458–2469.

[49] S. Xu, J. Dong, W. Cai, J. Li, A. Shaghaghi, N. Sun, and S. Ma, "Enhancing Security in Third-Party Library Reuse–Comprehensive Detection of 1-day Vulnerability through Code Patch Analysis," *arXiv preprint arXiv:2411.19648*, 2024.

[50] A. Dann, H. Plate, B. Hermann, S. E. Ponta, and E. Bodden, "Identifying Challenges for OSS Vulnerability Scanners - A Study & Test Suite," *IEEE Trans. Software Eng.*, vol. 48, no. 9, pp. 3613–3625, 2022.

[51] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "BINARM: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. Giuffrida, S. Bardin, and G. Blanc, Eds., vol. 10885. Springer, 2018, pp. 114–138.

[52] T. Riom and A. Bartel, "An In-Depth Analysis of Android's Java Class Library: its Evolution and Security Impact," in *IEEE Secure Development Conference, SecDev 2023, Atlanta, GA, USA, October 18-20, 2023*. IEEE, 2023, pp. 133–144.

[53] D. Uezono, J. Kushibiki, T. Sasaki, S. Hara, Y. Zhauniarovich, C. H. Ganán, M. van Eeten, and K. Yoshioka, "Poster: DarkWrt: Towards Building a Dataset of Potentially Unwanted Functions in IoT Devices," *Network and Distributed System Security*, 2025.

[54] H. Wang, Z. Liu, S. Wang, Y. Wang, Q. Tang, S. Nie, and S. Wu, "Are We There Yet? Filling the Gap Between Binary Similarity Analysis and Binary Software Composition Analysis," in *9th IEEE European Symposium on Security and Privacy, EuroS&P 2024, Vienna, Austria, July 8-12, 2024*. IEEE, 2024, pp. 506–523.