

RECURRING VULNERABILITY DETECTION IN IOT FIRMWARE VIA
REACHING-DEFINITION-GUIDED FINGERPRINTINGQirong Shen¹, Yutong Ye², Lei Sun³, Sihan Shen⁴, Qifan Shen^{*5},
Muhammad Wahab Hanif^{*6}¹Foretek Smart Technology (Zhejiang) Co., Ltd. Shaoxing, Zhejiang Province, 312030, China¹shenqr@hotmail.com, ²dayle.ye@hotmail.com, ³alex.sun@iforetek.com, ⁴harry.shen@outlook.com,
^{*5}shen.qifan@hotmail.com, ^{*6}mdw2016@hotmail.comDOI: <https://doi.org/10.5281/zenodo.22159950>**Keywords**IoT security; firmware analysis;
recurring vulnerability detection;
code reuse; vulnerability
fingerprint**Article History**

Received: 28 January 2026

Accepted: 13 March 2026

Published: 27 March 2026

Copyright @Author**Corresponding Author: ***
Qifan Shen**Abstract**

Existing recurring-vulnerability detection schemes based on vulnerability fingerprints commonly suffer from low fingerprint-extraction efficiency and limited robustness to code mutation. To address these problems, this paper proposes ReVuln, an automated method for detecting recurring vulnerabilities in IoT firmware. ReVuln first uses a large language model (LLM) agent to extract high-level vulnerability features from vulnerability reports. It then derives path constraints by combining reaching definition analysis (RDA) with constrained symbolic execution, and analyzes the neighboring nodes of the function call graph, to build a semantic-structural, two-dimensional vulnerability fingerprint. Finally, based on this fingerprint, a multi-stage matching strategy detects recurring vulnerabilities in target firmware. Experimental results show that ReVuln achieves an average fingerprint matching ratio of 92.59% on mutated samples. For vulnerability detection, ReVuln attains a precision of 50.7% and a recall of 69.2%; its precision is comparable to FirmRec's 51.2% and well above SaTC's 19.0%, while its recall exceeds FirmRec's 59.0% and SaTC's 29.4%. Compared with FirmRec, ReVuln reduces fingerprint-extraction time by 24%. These results show that ReVuln improves fingerprint-extraction efficiency, mutation robustness, and detection recall while maintaining detection precision.

1. Introduction

The rapid proliferation of connected devices has made the Internet of Things (IoT) the largest computing service platform in the world, and the number of connected devices is expected to keep growing rapidly[1], [2]. Firmware, as the core low-level component of IoT devices, directly determines device operational stability and is closely tied to user interests[3], [4]. However, firmware is typically closed-source and has weak observability, and is difficult to update or patch, making it a common attack surface whose exploitation can lead to data leakage or device

failure[5], [6], [7], [8]. Moreover, code reuse in firmware development causes vulnerabilities to propagate across products and versions, giving rise to recurring vulnerabilities[9]. Compared with static and dynamic analysis, which suffer from path explosion and emulation difficulties in firmware settings, fingerprint-based recurring-vulnerability detection relies on lightweight feature matching, offering a low deployment threshold, high throughput, and strong compatibility with heterogeneous firmware; it therefore serves as a key complement to existing techniques [10], [11], [12]. However, the key

bottleneck of such methods is that vulnerability-information analysis is difficult to automate intelligence sources are fragmented and heterogeneous in format, still requiring substantial manual parsing, which constrains efficient fingerprint generation[13]. Within the overall firmware security pipeline, recurring-vulnerability identification covers more than 80% of known risks and is the technical direction most urgently needed by industry. The core workflow of recurring-vulnerability detection typically involves extracting a vulnerability fingerprint from sample firmware and matching it against target firmware; in practice, there remains a pressing need to improve fingerprint stability under mutation conditions such as compiler optimization and code refactoring, and to reduce the cost and time of fingerprint extraction.

To address these challenges, this paper proposes ReVuln, an automated recurring-vulnerability detection method for IoT firmware, consisting of three stages. First, in the vulnerability-information extraction stage, a large language model (LLM) agent automatically extracts structured vulnerability information from vulnerability reports, including the vulnerability entry point, the associated Common Weakness Enumeration (CWE) category, and binary localization details. Second, in the fingerprint-extraction stage, reaching definition analysis (RDA), implemented with the angr framework, performs control-flow analysis between the source (determined by the input function) and the sink (determined by the CWE category) to identify candidate vulnerable paths on the call graph; angr-based constrained symbolic execution[14] then simulates these candidate paths to derive the path constraints of triggerable vulnerabilities, and the call-graph neighbors of the verified vulnerable function are analyzed and recorded as structural features. The semantic and structural features are then combined to construct the vulnerability fingerprint. Finally, in the detection stage, the fingerprint guides symbolic execution over the target firmware to enable recurring-vulnerability detection.

The main contributions of this work are as follows:

(1) An automated, LLM-based method for vulnerability-information extraction. We design and implement an LLM agent that automatically extracts high-level vulnerability features from multi-source, heterogeneous, unstructured vulnerability information, addressing the reliance on manual analysis and the resulting inefficiency of traditional approaches.

(2) A semantic-structural, two-dimensional vulnerability fingerprint model, together with an automated fingerprint-extraction and detection method built upon it. The method uses RDA to pre-filter path constraints, applies constrained symbolic execution to precisely extract path constraints as semantic features, and captures function call-graph neighbors as structural context. A cascaded detection strategy that combines structural matching with symbolic execution improves fingerprint-extraction efficiency and robustness to code refactoring and compiler optimization while preserving detection precision.

(3) ReVuln, an automated recurring-vulnerability detection system for IoT firmware that combines efficient fingerprint extraction and matching. On a test set spanning firmware from multiple mainstream vendors, ReVuln is comprehensively compared against several baseline methods. The results show that ReVuln's fingerprint design gives a clear advantage in mutation robustness and improved extraction efficiency, while its detection performance is comparable to state-of-the-art methods and exceeds that of traditional static analysis techniques.

2. Related Work

2.1. Automated Vulnerability Information Extraction

Generating high-quality vulnerability fingerprints depends on the accurate parsing of vulnerability information. However, existing vulnerability intelligence is generally multi-source, heterogeneous, and unstructured, and converting it into machine-processable structured data remains the primary challenge in this area. In early work, Cheng et al. [15] used rule-based binary diffing to match patch differences and localize vulnerability-related code, which was effective for patch-assisted vulnerability localization; however, the reliance

on fixed matching rules limits the ability to handle the diversity and ambiguity of natural-language descriptions, reducing recall and precision under textual variation and yielding limited overall automation. Subsequently, Yitagesu et al. [16] applied natural language processing (NLP)-based information extraction to automatically identify and extract key elements from vulnerability descriptions, supporting structured vulnerability knowledge and fingerprint generation; this improved automation and coverage relative to purely rule-based methods, but the approach lacks robust understanding of firmware-security-domain terminology and context, depends heavily on large volumes of high-quality labeled training data, and performs poorly at cross-document information integration and complex logical reasoning, making it difficult to construct complete exploit chains.

2.2. Fingerprint-based Recurring Vulnerability Detection

Existing IoT firmware vulnerability detection methods fall into two categories: exploratory general-purpose detection, which includes dynamic analysis, static analysis, and deep-learning-based detection; and targeted recurring-vulnerability detection, which includes code-clone-based and fingerprint-based methods. Among general-purpose methods, dynamic analysis approaches such as those of Zhou et al. [17], Bley et al. [18], and Venkataramani et al. [19] trigger vulnerabilities by executing firmware in an emulated environment; their effectiveness is limited by dependence on external peripherals, which makes emulation difficult to reproduce and constrains applicability. Static analysis approaches such as those of Sun et al. [20], Tu et al. [21], and Li et al. [22] use symbolic execution and reaching definition analysis to examine the static structure of vulnerabilities, but cannot exploit known vulnerability information to perform targeted search. Deep-learning-based approaches, such as those of Wen [23] and Al-Zhao et al. [24], model binary code as sequences or program graphs and learn semantic representations end-to-end using convolutional neural networks (CNNs), graph neural networks (GNNs), or Transformers, achieving a degree of cross-compiler, cross-optimization, and cross-

instruction-set generalization; their main limitations are a lack of interpretable, reproducible evidence chains for model decisions, high sensitivity to the distribution and quality of training data, and an inability to explicitly incorporate known vulnerability knowledge to support stable, targeted retrieval and recurring-vulnerability detection.

Recurring-vulnerability detection has evolved from code-clone-based to fingerprint-based techniques. Early methods relied mainly on code-clone detection: Germano et al. [25] and Yang et al. [26], for example, identified potential recurring vulnerabilities by retrieving repeated code blocks, but the coarse granularity of such features fails to capture vulnerability-specific semantics and behavior, leading to matching errors and missed detections once a vulnerability has been patched, lightly rewritten, or structurally altered. To overcome these limitations, Huang et al. [27] proposed FirmRec, which builds vulnerability fingerprints by capturing key features of the exploit chain; however, FirmRec still suffers from insufficient mutation robustness and low efficiency. Mutation robustness, the stability and generalization of a fingerprint under semantically equivalent code mutations, is a key indicator of the effectiveness of recurring-vulnerability detection methods. FirmRec's fingerprint matching is highly sensitive to binary-level variation introduced by compiler differences, optimization settings, and minor code changes: instruction sequences and key function-call relationships readily deviate from their original patterns after inlining or code reorganization, reducing fingerprint stability and causing drift or mismatch, which in turn limits mutation robustness. In addition, FirmRec's fingerprint extraction relies heavily on symbolic execution, which must maintain complex constraints while exploring many paths; as program size and branching increase, computational cost rises sharply and path explosion becomes likely, resulting in long detection times, high resource consumption, and difficulty scaling to large firmware corpora.

3. Vulnerability Definition

Figure 1 shows a representative example of a recurring vulnerability. Figure 1(a) shows CVE-

2020-10214, a vulnerability found in the D-Link DIR-825 router [28], [29]. The vulnerability resides in the httpd binary: the function sub_414134 retrieves the user-supplied CGI parameter ntp_server via get_cgi, and this input, without adequate validation or sanitization, is concatenated directly into an operating-system command string by sprintf and subsequently executed by system, constituting a typical command-injection vulnerability. This path constraint is the necessary condition for the injection: an attacker can craft a malicious ntp_server value to inject an additional shell command into the ntpclient invocation. Such injected commands execute with the privileges of the web-server process and can lead to sensitive-information disclosure, data corruption, denial of service, or full compromise of the device. Figure 1(b) shows a recurring instance of CVE-2020-10214, in which an

attacker injects an additional shell command of the same form into the ntpclient invocation, producing the same impact as the original CVE-2020-10214.

Although the two code snippets differ in their surrounding functionality, they share the same vulnerability features and path constraint, showing that a vulnerability instance can be abstracted into a reusable fingerprint. The features shown in Figure 1(a) can therefore serve as an effective fingerprint for identifying the essentially identical flaw in Figure 1(b). Detecting such recurring instances, however, remains a weakness of current automated tools: although security analysts can readily recognize this kind of cross-function vulnerable pattern from experience, existing automated tools still struggle to look past superficial implementation differences, making effective recurring-vulnerability detection a significant challenge.

```

1  int __fastcall sub_414134(char *a1, int a2) {
2      ...
3      v4 = (char *)get_cgi("ntp_server");
4      if (!v4)
5          v4 = "";
6      sprintf((char *)v7, "ntpclient -h %s -s -i 5 -c 1", v4);
7      printf("ntp_sync.cgi: cmd=%s\n", (const char *)v7);
8      system((const char *)v7);
9      ...
10 }
```

(a) CVE-2020-10214 vulnerability code

```

1  int NTP_Server(char *a1, int a2) {
2      ...
3      ntp = (char *)get_cgi("ntp_server");
4      v6 = ntp;
5      if (!v6)
6          return 0;
7      sprintf((char *)v2, "ntpclient -h %s -s -i 5 -g 0 -t", ntp);
8      system((const char *)v2);
9      ...
10 }
```

(b) CVE-2020-10214 repetitive vulnerability code

Figure 1. Example of a repetitive vulnerability

4.

5. ReVuln Method Design

As shown in Figure 2, ReVuln consists of a vulnerability-information extraction stage, a vulnerability-fingerprint extraction stage, and a vulnerability-detection stage. The vulnerability-information extraction stage converts multi-

source, heterogeneous vulnerability information into standardized data. It first automatically aggregates unstructured text such as official CVE advisories, technical articles, and proof-of-concept (PoC) scripts; an LLM agent then consolidates this raw intelligence and, through

prompt engineering, distills the verbose natural-language descriptions into structured information, producing key fields, including the vulnerability entry point, CWE category, and binary localization, that provide standardized input for the subsequent stages.

The fingerprint-extraction stage generates a vulnerability fingerprint based on the semantic-structural, two-dimensional fingerprint model. It maps the extracted input field to the source and, through a predefined mapping, converts the CWE field into the sink, completing the data flow from intelligence to fingerprint. To

avoid path explosion, this stage first uses reaching definition analysis (RDA) to identify candidate paths connecting the vulnerability entry point to the CWE-mapped dangerous function, narrowing the analysis scope; it then applies constrained symbolic execution to the candidate paths to extract path constraints as the vulnerability semantic feature; next, it analyzes the neighbors of the vulnerable function to capture the vulnerability's structural context in the call graph; finally, the two feature types are combined to form the target vulnerability fingerprint.

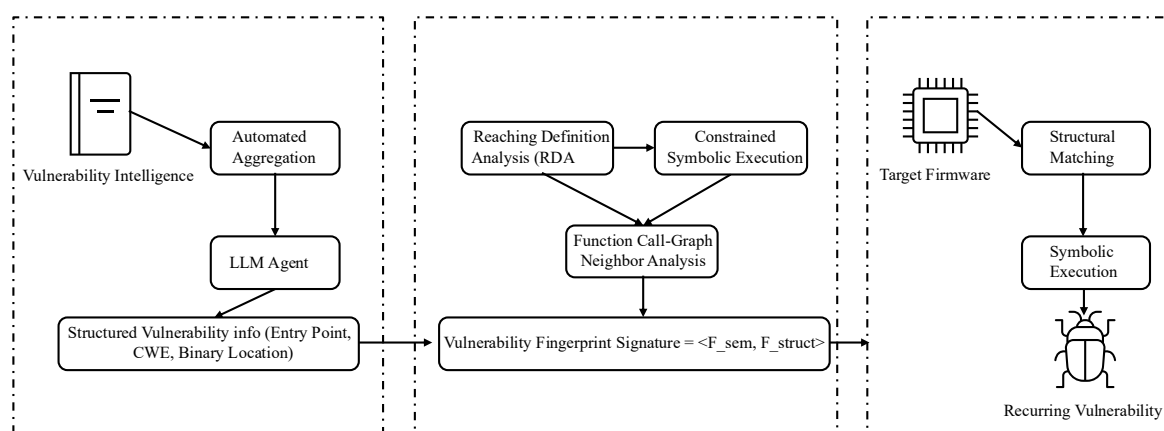


Figure 2. ReVuln architecture

The detection stage uses the constructed fingerprint to automatically detect recurring vulnerabilities in target firmware, following a coarse-to-fine strategy that combines preliminary screening with precise verification. It first performs rapid matching using the fingerprint's structural context over the target firmware's function call graph to efficiently screen highly similar candidate functions; it then drives a symbolic execution engine, combined with taint analysis, using the fingerprint's semantic feature to perform in-depth verification on the candidates. A recurring vulnerability is confirmed once an execution flow satisfying the path constraint is found. Each stage is described in detail below.

5.1. Vulnerability Information Extraction Stage

The vulnerability-information extraction stage is the first stage of ReVuln and a prerequisite for recurring-vulnerability detection. Its goal is to extract key feature information from multi-

source, heterogeneous, unstructured vulnerability descriptions and convert it into structured data, through two steps: automated aggregation and semantic analysis.

5.1.1. Automated Aggregation

The automated-aggregation module addresses the fragmentation and format heterogeneity of vulnerability data sources. It first anchors on authoritative vulnerability databases to extract structured baseline facts, including the vulnerability description, CVSS score, and CWE classification; it then performs targeted crawling and content normalization on key technical resources found in reference links; it further collects supplementary analytical corpora through a multi-strategy retrieval mechanism; and finally, it cleans and standardizes the collected data, producing normalized vulnerability information as high-quality input for subsequent automated analysis.

5.1.2. Semantic Analysis

The semantic-analysis module is driven by an LLM agent that performs a complete pipeline of semantic understanding, structured extraction, verification, and output normalization over the aggregated multi-source data. By constructing a unified evidentiary context and imposing strict schema constraints, the LLM agent precisely locates key meta-information within complex natural-language descriptions, focusing on high-level vulnerability features such as the vulnerability entry point, CWE type, firmware metadata, affected version range, and binary localization information. This module leverages the semantic-understanding capability of the LLM to convert fragmented natural-language descriptions into structured, high-level vulnerability features, providing high-quality input for subsequent fingerprint extraction. To ensure output quality, the module performs completeness checks and default-value backfilling, along with firmware-identifier normalization and schema alignment. A rule-based correction mechanism and consistency check finally convert the unstructured semantic understanding into a computable, structured representation, ensuring the data stability required by downstream automated tasks.

We designed a prompt template to drive the LLM agent's semantic analysis and structured extraction. The template first assigns the model the role of a cybersecurity vulnerability-analysis expert and requires it to output only a predefined JSON object, with no additional content. The template specifies a JSON schema containing fields such as poc_url, vuln_url, firmware_url, firmware_filename, official_range, groundtruth, input, and cwe, with guiding instructions to ensure consistency. To improve extraction performance in the firmware-security domain, we adopt a few-shot prompting strategy, appending two real CVE cases to the end of the prompt: a command-injection vulnerability (CVE-2020-10214) and a buffer-overflow vulnerability (CVE-2019-1663). To further mitigate LLM hallucination, we implement a multi-stage post-hoc verification mechanism: first, JSON syntax and schema validation, with re-extraction and regeneration (up to three retries) on failure; second, regular-expression and business-rule checks on field formats; third,

string and fuzzy matching between the extracted groundtruth/input.kv fields and the original source material; and finally, empty-string backfilling for missing fields, with anomalies logged for manual review.

Despite few-shot prompting and post-hoc verification, the LLM agent may still occasionally produce invalid JSON output. ReVuln handles such parsing failures through a fault-tolerant degradation mechanism: the system first attempts format repair and regeneration to recover a parseable result, constraining the LLM agent to fix formatting only, without altering semantic content. If repeated repair attempts fail to yield a valid structured result, the system falls back to a degraded mode. When sink information is missing or unstable, ReVuln fills the default sink value using a predefined CWE-to-sink mapping table based on the extracted CWE type, ensuring that the subsequent RDA and symbolic-execution pipeline can proceed. When other fields are missing or of low confidence, the system flags the extraction as low-confidence and applies a more conservative input-source enumeration and backtracking strategy in subsequent analysis.

5.2. Vulnerability Fingerprint Extraction Stage

The core idea of a vulnerability fingerprint is to abstract the key features of a known vulnerability into a general rule that is independent of the specific binary instruction-set architecture and compilation options. To enable detection of homologous vulnerabilities across different instruction-set architectures, we propose a semantic-structural, two-dimensional vulnerability fingerprint model. As shown in Figure 3, a fingerprint is represented as the tuple Signature = $\langle F_{sem}, F_{struct} \rangle$. The semantic feature F_{sem} takes the form of a path constraint composed of a set of constraint expressions that precisely describe the logical conditions (e.g., value assignments, formatting, length limits) that tainted data must satisfy while flowing from the source to the sink to trigger the vulnerability. The structural context F_{struct} is centered on the host function containing the sink and records its first-order neighbor nodes in the call graph, characterizing the code structure surrounding the vulnerability.

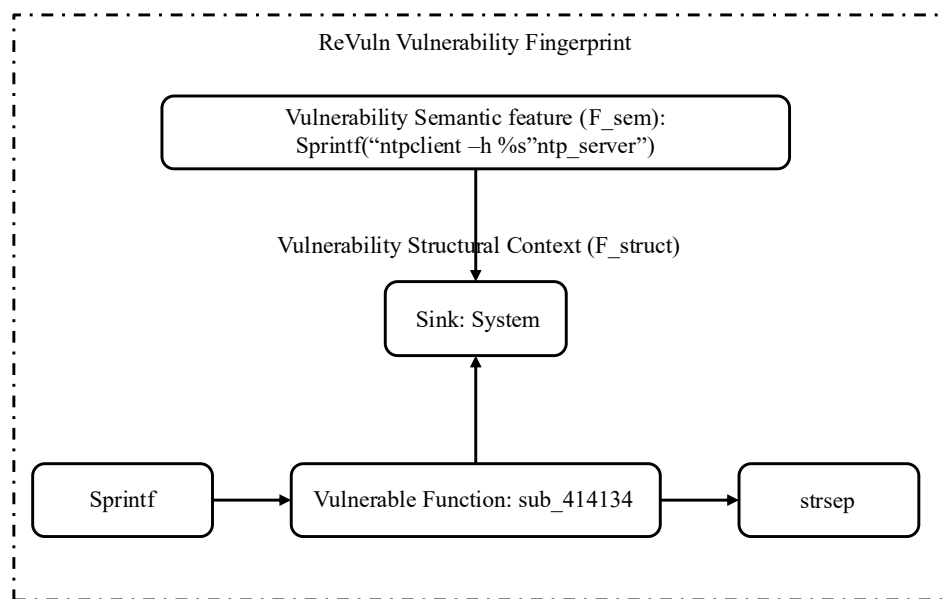


Figure 3. CVE-2020-10214 vulnerability fingerprint model

Fingerprint extraction is a prerequisite for detection; its goal is to automatically construct a vulnerability fingerprint from a known sample firmware image. This process comprises three steps: first, RDA identifies candidate paths connecting the vulnerability entry point to the dangerous function, avoiding path explosion in subsequent symbolic execution; second, semantic extraction applies constrained symbolic execution to the candidate paths to extract the constraint expressions that describe the vulnerability-triggering logic as the semantic feature; and third, the call relationships of the vulnerable function are analyzed and its neighbor nodes are extracted to generate the structural context.

5.2.1. Path Pre-filtering Based on Reaching Definition Analysis

Obtaining a complete path constraint requires tracing the execution path in the sample firmware from the sink back to the source. The prior method FirmRec, however, relies solely on symbolic execution to explore all paths from potential input points to program endpoints; this exhaustive traversal is prone to path explosion, resulting in an oversized analysis scope and low efficiency. To address this, ReVuln introduces an RDA-based path pre-filtering mechanism that rapidly prunes irrelevant paths and concentrates computational resources on verifying high-value paths.

Specifically, this stage first uses the extracted vulnerability information to fix the analysis anchors, marking external input interfaces as the source and dangerous functions obtained through CWE-type mapping (e.g., system, strcpy) as the sink. ReVuln uses the clinic module of the angr analysis framework to lift the code to the Abstract Intermediate Language (AIL). Compared with the conventional VEX intermediate representation, AIL more accurately recovers function calls and stack-variable operations, which is particularly useful for understanding complex control flow. ReVuln then performs RDA on the AIL representation, starting from the sink function's call site and tracing, along the control flow graph (CFG), the data definitions that can affect the sink function's key arguments, checking whether they are logically reachable from the input variable at the source. Through this process, ReVuln quickly filters out the large number of infeasible paths that cannot trigger vulnerability owing to unreachable control flow or broken data dependencies, yielding a candidate path-constraint set of controlled size. This set excludes the great majority of irrelevant paths, providing a precise input scope for the subsequent constrained symbolic execution and improving the overall efficiency of fingerprint extraction. To further mitigate path explosion, ReVuln applies three targeted strategies.

Indirect-call resolution. In firmware binaries, the construction of dangerous-function arguments often crosses PLT/GOT trampolines and function pointers, obscuring the call target and substantially enlarging the symbolic-execution state space. ReVuln locally identifies common trampoline patterns and recovers the actual callee entry point, converting unresolvable indirect jumps into resolvable direct calls wherever possible. For indirect calls whose target still cannot be recovered, ReVuln conservatively marks them as uncertain call sites and restricts the branching they introduce, focusing the analysis on the data dependencies necessary for constructing the dangerous sink's arguments and narrowing the scope of subsequent constrained symbolic execution.

Loop unrolling. In firmware binaries, input-related variables are often accumulated, concatenated, or validated incrementally within loop bodies; unrolling loops statement-by-statement causes RDA and subsequent symbolic execution to generate large numbers of redundant states along back edges, triggering path explosion. ReVuln therefore performs limited, small-step loop unrolling to cover common vulnerability-triggering patterns, by default unrolling only a bounded number of iterations to avoid state explosion. A loop is unrolled further only if its loop variable, or an in-loop definition, can reach a key argument of the dangerous sink or affect a key branch condition leading to the sink, and carries taint originating from the source; loops unrelated to the sink's arguments are instead summarized rather than unrolled, suppressing path explosion from irrelevant iterations and narrowing the analysis scope.

Recursion-depth limiting. In firmware binaries, logic such as configuration parsing, directory traversal, and tree processing is often implemented recursively, obscuring call targets. Unbounded expansion of recursive calls would rapidly inflate the call stack and cause the path space to grow exponentially, undermining the scalability of candidate-path screening and symbolic-execution verification. ReVuln therefore sets a default recursion-unrolling depth threshold, expanding recursive calls normally within this threshold to preserve key

data-dependency chains; beyond the threshold, recursive calls collapsed into function summaries to avoid further expanding the call tree and to suppress path explosion. For the small number of candidates that cannot be determined as triggerable within shallow unrolling, ReVuln applies a progressive deepening strategy that relaxes the depth as needed and caches the results of recursive calls with identical symbolic input signatures to avoid re-expanding the same subtree, balancing recall against performance while keeping the analysis scope as narrow as possible.

5.2.2. Vulnerability Semantic Feature Extraction

The vulnerability semantic feature describes, through constraint expressions, the logical conditions that tainted data must satisfy from its introduction to the point where it triggers the vulnerability. Unlike a simple sequence of assembly instructions, this expression forms the path constraint connecting the source, the point in the code where external tainted data is introduced, and the sink, the dangerous function that ultimately triggers vulnerability. The constraint expression recursively records each key transformation the source data undergoes as it propagates, capturing the fact that “for the argument to trigger the vulnerability at the sink, its value must be produced by a specific sequence of operations on the source data.” ReVuln first uses the angr framework to lift the binary code to an intermediate representation and models the input data at the source as a symbolic variable; the symbolic execution engine then simulates execution along the candidate path, continuously tracking the state of the symbolic variable. During this process, the engine disregards specific register allocations and irrelevant control-flow jumps, focusing instead on collecting the path constraints along the control flow; when execution reaches the sink function, the system consolidates the data dependencies to generate the constraint expression. Two additional strategies address common sources of instability in this stage.

Pointer-aliasing handling. In binary symbolic execution, some memory-access addresses are determined by input-dependent symbolic

expressions, making it difficult for the analyzer to precisely determine whether different accesses refer to the same memory object; this can introduce conservative but potentially spurious aliasing relationships, causing the data-dependency chain to fluctuate with address-concretization choices and reducing the stability and reproducibility of semantic constraint extraction. ReVuln applies consistent concretization to the symbolic variables within symbolic address expressions, so that the same symbolic address maps, as far as possible, to a stable set of candidate addresses across the analysis, reducing spurious aliasing caused by random concretization. When tainted data is detected being used as an address, ReVuln avoids imposing strong assumptions on the dereferenced content and instead introduces a new abstract memory object representing the uncertainty of the dereference, explicitly preserving its association with the tainted address in the data-dependency graph to avoid propagation loss from incorrectly merged aliases. This strategy remains conservative while suppressing path explosion caused by spurious-alias-induced state forking, constraining memory uncertainty to the necessary objects and narrowing the analysis scope, thereby improving the stability and reproducibility of semantic constraint extraction.

External-function modeling. In firmware binaries, much of the key data flow crosses standard-library and system interfaces whose true semantics are difficult to reproduce precisely under purely static symbolic execution; directly inlining the real implementation is costly and prone to introducing additional state forking, reducing the stability and reproducibility of path-constraint extraction. ReVuln therefore uses function summaries to model external calls during constrained symbolic execution: common library functions are given semantic summaries that explicitly describe their preconditions, key memory read/write effects, and taint-propagation rules, directly updating the symbolic state rather than expanding the function body, thereby avoiding path explosion; external calls with unknown semantics or uncertain behavior are given a conservative over-approximate summary to

preserve coverage while limiting the irrelevant branch expansion introduced by uncertain return values, narrowing the analysis scope. Using CVE-2020-10214 in Figure 1(a) as an example, ReVuln identifies the call to `get_cgi` as the taint source, retrieving the configuration value `ntp_server`, and identifies the call to `system` as the dangerous sink. Path analysis by the symbolic execution engine then shows that the source data is formatted and embedded into the template string `"ntpcient -h %s"` by `sprintf`. Based on this, the system extracts the constraint on the sink argument, producing the equality relation `Arg_system = sprintf("ntpcient -h %s", "ntp_server")`. This expression is the semantic fingerprint of the vulnerability, indicating that the sink argument `Arg_system` must equal the result of applying this specific formatting to the source data.

5.2.3. Vulnerability Structural Context Feature Construction

To rapidly converge the analysis scope when scanning target firmware, ReVuln builds a structural-context feature around the located host function, based on the function call graph. To handle localization in stripped binaries, the algorithm starts from a standard-library sink function, still identifiable in stripped binaries, and locates its direct caller as the host function. Once the host function is located, ReVuln extracts its first-order neighbor nodes in the function call graph, forming the vulnerability structural context that characterizes the code structure surrounding the vulnerability. To reduce noise from generic library calls, the neighbor nodes are further filtered: user-defined function nodes are prioritized, while only library functions that carry anchor significance and remain stably identifiable in the binary are retained; other unstable nodes, such as generic library wrapper functions, are filtered out. Because call relationships between functions are more stable across the source-to-binary compilation process than intra-function control flow, and are less affected by code refactoring, this structural feature strengthens the fingerprint's robustness to mutation.

ReVuln constructs the fingerprint following Algorithm 1. The algorithm first builds the CFG of the input binary and extracts the function call

graph CG, identifying all call sites that reference the target sink function SinkFunc (lines 1–3). For each sink call, the system performs RDA on the AIL representation derived from the CFG, filtering candidate path constraints by tracing control-flow dependencies that can be traced back to the input source FirmEntry (lines 4–12). It then verifies the candidate path set via constrained symbolic execution, simulating the

path state (lines 13–14). If the execution state successfully triggers the vulnerability-detection logic (line 15), the algorithm extracts the vulnerability semantics along the path (line 16), locates the vulnerable function and its neighbors using CG to build the structural context (lines 17–19), and finally returns the resulting fingerprint Signature (lines 20–22).

Algorithm 1. Vulnerability fingerprint generation

```

Input: Binary file (BinaryFile), sink function (SinkFunc),
       source function (FirmEntry), vulnerability type (VulnType)
Output: Vulnerability fingerprint Signature

1. CFG ← GenerateCFG(BinaryFile)
2. CG ← BuildCallGraph(CFG)
3. SinkCallSites ← FindReferences(CFG, SinkFunc)
4. FOR func_pred IN SinkCallSites DO
5.   AIL_Graph ← LiftToAIL(func_pred, CFG)
6.   Defs ← RDA(AIL_Graph, SinkFunc.args)
7.   FOR definition IN Defs DO
8.     IF ReachesSource(definition, FirmEntry) THEN
9.       Paths.add(ExtractControlFlowPath(definition))
10.    END IF
11.  END FOR
12. END FOR
13. FOR Path IN Paths DO
14.   ExecState ← ConstrainedSymExec(FirmEntry, Path)
15.   IF IsTriggered(VulnType, ExecState) THEN
16.     SemFeat ← ExtractSemantic(ExecState.constraints)
17.     Anchor ← GetEnclosingFunction(Path)
18.     Neighbors ← GetNeighbors(CG, Anchor)
19.     StructFeat ← BuildContext(Anchor, Neighbors)
20.     Signature.add(SemFeat, StructFeat)
21.   END IF
22. END FOR

```

5.3. Vulnerability Detection Stage

The detection stage uses the generated vulnerability fingerprint to quickly and accurately determine whether a recurring vulnerability is present in target IoT firmware. Because target firmware typically lacks a symbol table and may have been compiled differently from the sample firmware, ReVuln avoids source- or instruction-level text comparison and instead lifts the target binary, via angr, to the same AIL representation as the fingerprint for matching. ReVuln adopts a coarse-to-fine, layered detection strategy consisting of two core stages: structural matching and symbolic execution.

ReVuln first performs coarse-grained localization and screening based on structural context. Since the target firmware lacks symbol information, ReVuln uses the fingerprint's library-function sink as an index, reverse-locating, in the function call graph, all higher-level functions that call this sink as initial candidates. It then extracts the first-order neighbors of these candidate functions and compares them with the fingerprint's structural context using the Dice coefficient. The Dice threshold for structural-context matching was set to 0.85, based on preliminary experiments spanning a range of values. As shown in Table 1, as the threshold increases from 0.70 to 0.95, precision

risks from 46.2% to 64.6%, while recall falls from 75.1% to 52.9%. A threshold that is too low weakens the screening effect and substantially enlarges the candidate set, increasing the cost of subsequent symbolic execution, whereas a threshold that is too high becomes overly sensitive to minor refactoring or inlining, increasing mismatches and reducing

recall. Within this range, a Dice threshold of 0.85 achieves the most robust trade-off between precision and recall and yields the highest F1-score. This process exploits topological features preserved under code reuse to eliminate false-positive candidates, narrowing down to a high-confidence candidate set and ensuring the matching process is quantitatively tractable.

Table 1. Precision, recall, and F1-score of structural-context matching under different Dice thresholds (%)

Dice threshold	Precision	Recall	F1-score
0.7	46.2	75.1	57.2
0.75	48.2	71.4	57.5
0.8	49.8	70.3	58.3
0.85	50.7	69.2	58.6
0.9	54.3	61.6	57.7
0.95	64.6	52.9	58.2

ReVuln then performs semantic-feature-based verification on the screened candidate functions. To improve efficiency, ReVuln initializes the symbolic state from the fingerprint's path-constraint entry point and uses a symbolic execution engine combined with taint analysis to track data flow. During exploration, the engine performs fingerprint-guided matching along candidate execution paths, verifying: (1) control-flow matching, whether a feasible execution path exists along which controlled data flows from a source to the dangerous operation specified as the sink; (2) path-constraint matching, whether the path constraint at the sink satisfies the vulnerability-triggering condition defined in the fingerprint; and (3) vulnerability-type matching, whether execution reaching the sink triggers a condition consistent with the vulnerability type described by the fingerprint. Only when a single execution path simultaneously satisfies all three verification conditions does ReVuln report a matching vulnerability instance in the target firmware.

5.4. System Implementation

We implemented ReVuln, a prototype system for automated IoT firmware vulnerability detection based on the semantic-structural, two-dimensional fingerprint model, in Python 3 (v3.11). For vulnerability-information extraction, an LLM agent is built on top of a large-language-

model API, driven by the prompt template, few-shot prompting strategy, and multi-stage post-hoc verification mechanism described above to perform semantic understanding and structured extraction, with regular expressions used for output cleaning and validation. For fingerprint extraction and detection, the control flow graph and function call graph of the binary program are constructed using the angr framework, and the NetworkX library is used to analyze inter-function call topology to build structural vulnerability context. The AIL intermediate representation and RDA are used for backward control-flow tracing to obtain candidate path constraints, and angr's concolic execution engine is used for symbolic-execution-based path-constraint screening and recurring-vulnerability detection.

6. Experimental Methodology and Results

6.1. Experimental Setup

The evaluation uses 73 firmware images from seven well-known IoT vendors (Netgear, Tenda, D-Link, Linksys, TP-Link, TOTOLINK, and Cisco), spanning the two mainstream architectures ARM32 (27 images) and MIPS32 (46 images); firmware samples were collected from public vulnerability databases and vendor websites. All experiments were run on a Linux workstation equipped with an Intel Xeon E5-2686 v4 processor and 48 GB of RAM.

6.2. Results Analysis

6.2.1. Anti-Mutation Capability of Vulnerability Fingerprints

To quantitatively assess the mutation robustness of ReVuln's fingerprint design, we compare it with FirmRec. FirmRec's fingerprint relies heavily on Essential Function Calls and identifies vulnerabilities by matching function prototypes; this call-specific representation can be fragile under code mutation. In contrast, by combining semantic features with structural context, ReVuln uses constraint expressions and local graph features to effectively filter out interference from instruction obfuscation and minor control-flow adjustments, yielding stronger mutation robustness.

To evaluate this quantitatively, we built a test set covering four representative CVE vulnerability types. Using the Gemini-3-pro model, we designed four targeted prompts to generate, from the target source code, semantically equivalent but syntactically different mutated

samples covering variable renaming, function refactoring, control-flow transformation, and benign code insertion. All samples were first successfully compiled into binaries for the corresponding architecture. To ensure that mutated samples preserved the original vulnerability semantics, we performed semantic-equivalence verification: static data-flow analysis to confirm consistency of the key path and taint propagation, followed by PoC reproduction in an isolated environment to confirm exploitability. In total, 162 mutated samples were generated from 20 CVEs, all passing the verification procedure above. Using the function-refactoring mutation of CVE-2020-10214 shown in Figure 4 as an example, FirmRec's matching ratio dropped sharply because the `get_cgi` call was replaced by a wrapper function, whereas ReVuln, relying on its constraint expression, successfully saw through the wrapper and correctly identified the core tainted path.

```

1 char* get_ntp_server_address() {
2     char *server_address = (char *)get_cgi("ntp_server");
3     if (!server_address) {
4         server_address = "";
5     }
6     return server_address;
7 }
8 int __fastcall sub_414134(char *input_arg, int another_arg) {
9     char *server = get_ntp_server_address();
10    char cmd_buffer[256];
11    sprintf(cmd_buffer, "ntpcclient -h %s -s -i 5 -c 1", server);
12    system(cmd_buffer);
13    return 0;
14 }

```

Figure 4. CVE-2020-10214 mutated code produced by function refactoring

Using the same 162 mutated samples across four mutation types (variable renaming, function refactoring, control-flow transformation, and benign code insertion) and four vulnerability types (improper input validation, stack overflow, command injection, and buffer overflow/out-of-bounds access), all of which passed compilation, static data-flow consistency checks, and PoC

reproduction, we extracted fingerprints from the original and mutated samples using ReVuln and FirmRec and determined matches by Jaccard similarity. Table 2 reports fingerprint matching ratios by mutation method, and Table 3 by vulnerability type. In both breakdowns, ReVuln consistently outperforms FirmRec, indicating stronger generalization and stability.

Table 2. Fingerprint matching accuracy across different mutation methods (%)

Mutation method	ReVuln	FirmRec
Variable renaming	93.4	89.3
Function refactoring	91.2	85.8
Control-flow transformation	89.3	84.5
Benign code insertion	96.4	93.5

Table 3. Fingerprint matching accuracy across different vulnerability types (%)

Vulnerability type	ReVuln	FirmRec
Improper input validation	93.6	89.1
Stack overflow	91.8	87.2
Command injection	94.2	90.5
Buffer overflow (out-of-bounds)	90.7	86.2
Average	92.59	88.27

Fingerprints were extracted from the original code and each mutated variant using both ReVuln and FirmRec, and their similarity was computed via the Jaccard similarity in Eq. (1). A match is declared when the similarity is no less than a threshold τ . To avoid subjectivity in choosing τ , we performed a sensitivity scan over $\tau \in [0.50, 0.95]$ in steps of 0.05, computing the true-match rate $\text{TPR}(\tau)$ on the positive sample set and the false-match rate $\text{FPR}(\tau)$ on the negative sample set.

$$J_s(A, B) = \frac{|A||B|}{|A| + |B| - |A \cap B|} \quad (1)$$

As shown in Table 4, $\text{FPR}(\tau)$ rises sharply for $\tau < 0.75$, while $\text{TPR}(\tau)$ drops noticeably for $\tau > 0.85$, an effect that is especially pronounced under function-refactoring and control-flow-transformation mutations. Overall, $\tau = 0.8$ falls in the region of least overlap between the positive- and negative-sample similarity distributions and provides the best trade-off between $\text{TPR}(\tau)$ and $\text{FPR}(\tau)$; we therefore adopt $\tau = 0.8$ as the default matching threshold.

Table 4. Sensitivity scan across Jaccard thresholds (%)

Jaccard threshold	TPR(τ)	FPR(τ)
0.7	96.3	8.7
0.75	94.1	4.2
0.8	92.6	2.1
0.85	88.4	0.8
0.9	82.7	0.3

6.2.2. Vulnerability Fingerprint Extraction Efficiency

We next evaluate ReVuln's fingerprint-extraction efficiency against FirmRec. FirmRec performs symbolic execution starting from potential attack input points, making it highly prone to path explosion: when analyzing firmware with complex control flow, the number of paths grows exponentially, incurring substantial computational cost. ReVuln instead first uses reachability analysis to rapidly screen the candidate path set connecting input points to dangerous functions, substantially narrowing the scope of symbolic execution, and then applies constrained symbolic execution only to this reduced set to extract path constraints, while concurrently capturing the local call-graph context of the vulnerable function.

Using CVE-2020-10214 in Figure 1 as an example, FirmRec must forward-explore every

possible branch within sub_414134 starting from the get_cgi input point, whereas ReVuln starts directly from the call site of the sink function system and uses RDA to quickly determine that its command argument originates from the formatted output of sprintf, whose input in turn depends on the return value of get_cgi. This core control-flow path is located quickly, and unrelated logic elsewhere in the function is disregarded entirely, substantially improving analysis efficiency.

We extracted fingerprints for 20 CVEs and recorded the number of potential paths, the number of triggerable paths, and the elapsed time. Potential paths refers to the total number of candidate paths identified by reachability analysis; triggerable paths refers to the number of paths, among these, that constrained symbolic execution verifies as successfully triggering the vulnerability-detection logic; and elapsed time

refers to total CPU seconds. As shown in Table 5, at comparable detection accuracy, ReVuln reduces the number of potential paths by 89%

and triggerable paths by 61% relative to FirmRec, and reduces fingerprint-extraction time by 24%.

Table 5. Vulnerability fingerprint extraction efficiency

Method	Potential paths	Triggerable paths	Time (CPU s)
ReVuln	205	51	6,235
FirmRec	1,835	132	8,196

6.2.3. Vulnerability Detection Accuracy

In vulnerability detection, overall performance is typically quantified using precision and recall. Precision reflects the proportion of true vulnerabilities among all flagged candidates, higher values indicate fewer false positives, while recall reflects the method's coverage of true vulnerabilities, higher values indicate fewer false negatives (missed detections). ReVuln adopts FirmRec's criteria for determining ground-truth vulnerabilities: for every detected candidate, an analyst manually reviews it against the PoC obtained during the vulnerability-information extraction stage, and a candidate is labeled a true vulnerability only if it satisfies all three of the following criteria: (1) reachability, control-flow analysis confirms that the function path containing the vulnerability is logically reachable

from the firmware's entry point; (2) controllability, the control-flow source that triggers the vulnerability must originate from a network interface or another attacker-controllable input; and (3) constraint satisfaction, all security checks or filtering logic present along the path can be effectively bypassed or satisfied under a specific attack input.

As shown in Table 6, reachability-based screening reduced the large pool of input entries to 41,753 valid entries, from which ReVuln reported 1,837 potential vulnerability points; manual review confirmed 931 of these as true vulnerabilities, demonstrating ReVuln's ability to localize vulnerabilities effectively at firmware scale.

Table 6. ReVuln vulnerability detection overview

Firmware vendor (count)	Architecture	All input entries	Valid input entries	Detected candidates	True vulnerabilities
Tenda (11)	ARM32 (4), MIPS32 (7)	45,126	4,312	319	184
Netgear (41)	ARM32 (17), MIPS32 (24)	289,456	29,265	1,226	589
Linksys (5)	ARM32 (4), MIPS32 (1)	9,726	812	40	21
TP-Link (2)	MIPS32 (2)	7,265	1,023	24	16
D-Link (3)	ARM32 (2), MIPS32 (1)	11,264	623	32	12
TOTOLINK (5)	MIPS32 (5)	11,032	1,203	56	26
Cisco (6)	MIPS32 (6)	50,236	4,515	140	83
Total (73)	ARM32 (27), MIPS32 (46)	424,105	41,753	1,837	931

To objectively assess detection performance, we compare ReVuln against SaTC and FirmRec, with results shown in Table 7.

Table 7. Vulnerability detection accuracy: ReVuln vs. SaTC vs. FirmRec

Firmware vendor (count)	ReVuln				SaTC				FirmRec			
	True vul n.	Detected	Precision (%)	Recall (%)	True vul n.	Detected	Precision (%)	Recall (%)	True vul n.	Detected	Precision (%)	Recall (%)
Tenda (11)	184	319	57.1	64.3	60	562	10.7	21	251	424	59.2	87.8
Netgear (41)	589	1,226	48.1	70	296	1,260	23.5	35.2	406	869	35.2	48.2
Linksys (5)	21	40	52.1	56.8	2	5	40	5.4	16	37	43.3	43.2
TP-Link (2)	16	24	66.7	51.6	4	33	12.1	12.9	12	18	63.2	38.7
D-Link (3)	12	32	37.5	38.7	13	25	52	41.9	9	28	32.1	32.1
TOTOLINK (5)	26	56	46.4	100	0	54	-	-	24	52	45.9	45.9
Cisco (6)	83	140	59.3	89.4	21	142	14.7	14.7	76	122	62.3	62.3
Total (73)	931	1,837	50.7	69.2	396	2,081	19	29.4	794	1,550	51.2	59

Overall, ReVuln exceeds the static-analysis method SaTC on both precision and recall and shows better overall detection performance than FirmRec. Specifically, ReVuln achieves an overall precision of 50.7%, comparable to FirmRec's 51.2% and substantially higher than SaTC's 19.0%; its recall of 69.2% exceeds FirmRec's 59.0% and SaTC's 29.4%. ReVuln ultimately identified 931 true vulnerabilities, 137 more than FirmRec. ReVuln's recall advantage stems mainly from its stronger mutation robustness, which allows it to better cover vulnerabilities that have undergone function refactoring, wrapping, or local logic adjustments; FirmRec relies more heavily on specific call sequences and tends to miss vulnerabilities when the call chain changes owing to refactoring. On precision, ReVuln is mainly affected by the difficulty of modeling custom sanitization logic in firmware: conservative taint propagation can misclassify data flows that have already been sanitized or safely truncated as exploitable paths. SaTC, in contrast, relies only on shallow keyword matching without semantic or data-flow constraints, resulting in weaker overall performance.

6.2.4. Effectiveness of the Vulnerability Structural Context

To verify the contribution of the structural-context feature, we built an ablated variant, ReVuln-NoStruct, that removes this feature from the fingerprint, and compared it against the full ReVuln model; results are shown in Table 8. Introducing structural context reduces the number of candidate functions from 3,066 to 1,540, indicating that a large number of spurious candidates, functions that call the dangerous function but do not match topologically, are effectively filtered out. Correspondingly, average detection time drops from 2,265 s to 1,359 s, showing that structural context narrows the symbolic execution space and mitigates path explosion during detection. On precision, ReVuln shows only a slight 1.6-percentage-point decrease relative to ReVuln-NoStruct, suggesting that a small number of true vulnerabilities whose structure has changed substantially may be filtered out by the structural matching step. ReVuln-NoStruct retains only the semantic feature of the fingerprint, removing the structural context; as a result, it cannot perform call-graph-based structural screening during detection and must instead apply constrained symbolic execution directly to

every potential sink function. Overall, the structural-context feature substantially improves detection-stage efficiency at essentially no cost to

detection accuracy, making it a key component for scalable vulnerability detection.

Table 8. Effect of the vulnerability structural context

Method	Precision (%)	Average time (CPU s)	Candidate functions
ReVuln-NoStruct	52.3	2,265	3,066
ReVuln	50.7	1,359	1,540

6.2.5. Sensitivity to Different LLM Agents for Vulnerability Information Extraction

To assess the sensitivity of ReVuln's LLM agent in the vulnerability-information extraction stage, we compared extraction performance across several mainstream large language models. This stage extracts key fields, the vulnerability entry point, CWE type, and target binary, among others, from unstructured text such as CVE advisories, technical articles, and PoCs, converting them into standardized information; since these fields drive the subsequent source/sink mapping, extraction quality directly affects fingerprint completeness and detection precision. We evaluated four mainstream LLMs: the closed-source models ChatGPT-5.2, Claude Opus 4.6, and Gemini-3-pro, and the open-source model Llama-4-Scout, on the same 20-CVE, multi-source heterogeneous test set used earlier, with all models using the same prompt template, few-shot prompting strategy, and multi-stage post-hoc verification mechanism. Evaluation metrics include: (1) JSON validity rate, the proportion of model outputs that

successfully pass JSON syntax parsing, reflecting output-format stability; (2) average F1 over key fields, precision, recall, and F1 are computed for each core field (input, cwe, groundtruth, official_range, etc.) against manually annotated ground truth, and the average F1 across these fields is reported, where F1 is the harmonic mean of precision (P) and recall (R), as given in Eq. (2); (3) overall extraction F1, the aggregate F1 across all fields; and (4) average time, the average CPU time per CVE, measuring extraction efficiency.

$$F_1 = 2 \times \frac{2PR}{P + R} \quad (2)$$

Results are shown in Table 9. All four models achieve a JSON validity rate above 86% and an average key-field F1 above 85%, reaching as high as 93.6%, confirming the effectiveness of the prompt-engineering, few-shot prompting, and multi-stage post-hoc verification design. Gemini-3-pro, the model primarily used in this work, offers the best speed while maintaining a JSON validity rate of 92.7%.

Table 9. Vulnerability-information extraction performance across different LLMs

LLM	JSON validity (%)	Avg. key-field F1 (%)	Overall F1 (%)	Avg. time (s)
ChatGPT-5.2	95.1	92.8	92.5	14.2
Gemini-3-pro	92.7	91.9	91.7	11.3
Claude Opus 4.6	91.5	93.6	93.4	16.8
Llama-4-Scout	86.5	85.2	85.1	21.3

7. Conclusion

This paper presented ReVuln, an automated method for detecting recurring vulnerabilities in IoT firmware. By combining LLM-based extraction of key information from vulnerability reports with reachability analysis, constrained symbolic execution, and function-call-graph

structural-context modeling, ReVuln constructs vulnerability fingerprints with both structural and semantic constraints, enabling accurate identification and matching homologous vulnerabilities across firmware. Experimental results show that ReVuln performs well in terms of fingerprint-extraction efficiency and mutation

robustness, with overall detection performance exceeding that of traditional static analysis methods.

The method nonetheless has limitations: coverage of the CVE signature library and sample set is limited, and generalization requires further validation; moreover, purely static analysis has inherent limitations in the presence of code obfuscation, encryption, dynamic loading, and virtual calls. Future work will expand the signature library and the set of real-world firmware samples and explore combining dynamic instrumentation and de-obfuscation preprocessing with the current static-analysis pipeline.

REFERENCES

- [1] J. Shehu Yalli, M. Hilmi Hasan, and A. Abubakar Badawi, "Internet of Things (IoT): Origins, Embedded Technologies, Smart Applications, and Its Growth in the Last Decade," *IEEE Access*, vol. 12, pp. 91357–91382, 2024, doi: 10.1109/ACCESS.2024.3418995.
- [2] A. Lavanya, J. Divya and R. Zahira "Introduction to the Internet of Things - IoT for Smart Grid - Wiley Online Library." Accessed: Aug. 27, 2026. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781394279401.ch1>
- [3] D. R. Gomes, F. A. A. Lins, and M. Vieira, "A Comprehensive Empirical Study of Security Vulnerabilities in IoT Gateway Software," *IEEE Internet Things J.*, vol. 13, no. 14, pp. 30303–30324, Jul. 2026, doi: 10.1109/JIOT.2026.3703749.
- [4] F. Serepas, L. Papias and V. Marinakis "Lightweight Embedded IoT Gateway for Smart Homes Based on an ESP32 Microcontroller." Accessed: Aug. 27, 2026. [Online]. Available: <https://www.mdpi.com/2073-431X/14/9/391>
- [5] M. Steinbach, J. Großschädl, and P. B. Rønne, "Hard-to-Find Bugs in Public-Key Cryptographic Software: Classification and Test Methodologies," in *Security, Privacy, and Applied Cryptography Engineering*, C. Karfa, N. Asadi, and A. Chattopadhyay, Eds., Cham: Springer Nature Switzerland, 2026, pp. 336–357. doi: 10.1007/978-3-032-16342-4_19.
- [6] G. Wu, H. Liang, L. Yin, Q. Yi, C. Wang, and J. Zhang, "Detecting Vulnerabilities in Embedded Systems via Taint Analysis: A Survey," *ACM Comput. Surv.*, Aug. 2026, doi: 10.1145/3841632.
- [7] S. Duttagupta, S. Wyns and N. Antonijevic "One Tap to Hijack Them All: A Security Analysis of the Google Fast Pair Protocol | IEEE Conference Publication | IEEE Xplore." Accessed: Aug. 27, 2026. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/11573524>
- [8] M. Khan, M. Mushtaq, R. Pacalet, and L. Apvrille, "Toward Secure RISC-V Microarchitecture: Vulnerability Classes and Defenses," *IEEE Access*, vol. 14, pp. 51504–51519, 2026, doi: 10.1109/ACCESS.2026.3679984.
- [9] D. R. Gomes, F. A. A. Lins, and M. Vieira, "A Comprehensive Empirical Study of Security Vulnerabilities in IoT Gateway Software," *IEEE Internet Things J.*, vol. 13, no. 14, pp. 30303–30324, Jul. 2026, doi: 10.1109/JIOT.2026.3703749.
- [10] M. Toptsis, N. Karkanis and A. Giannakoulas "A Review of Embedded Software Architectures for Multi-Sensor Wearable Devices: Sensor Fusion Techniques and Future Research Directions." Accessed: Aug. 27, 2026. [Online]. Available: <https://www.mdpi.com/2079-9292/15/2/295>

- [11] W. Xu and K. Li, "A Review of Adversarial Example Detection in IoT Sensor Networks: Methods, Evaluation, and Edge Deployment Constraints." Accessed: Aug. 27, 2026. [Online]. Available: <https://www.mdpi.com/1424-8220/26/16/5044>
- [12] B. Lima and R. Pinto, "Current Challenges and Future Perspectives in Testing IoT Systems: A Comprehensive Review," *IEEE Sens. Rev.*, vol. 3, pp. 22–47, 2026, doi: 10.1109/SR.2025.3628264.
- [13] P. Radanliev, K. Atefi, O. Santos, and C. Maple, "Integrating agentic risk signalling in trusted research environments: Automating VEX with Agent2Agent protocols and model context protocol (MCP) in SACRO and TRevolution pipelines," *Comput. Stand. Interfaces*, vol. 96, p. 104079, Mar. 2026, doi: 10.1016/j.csi.2025.104079.
- [14] S. Kaniewski, F. Schmidt, M. Enzweiler, M. Menth, and T. Heer, "A Systematic Literature Review on Detecting Software Vulnerabilities with Large Language Models," *ACM Trans. Softw. Eng. Methodol.*, May 2026, doi: 10.1145/3815425.
- [15] Y. Cheng et al., "'Say What You Mean': Natural Language Access Control With Large Language Models for Internet of Things," *IEEE Trans. Inf. Forensics Secur.*, vol. 21, pp. 6482–6497, 2026, doi: 10.1109/TIFS.2026.3712224.
- [16] S. Yitagesu et al., "Systematic Literature Review on Software Security Vulnerability Information Extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 4, p. 95:1-95:52, Mar. 2026, doi: 10.1145/3745026.
- [17] W. Zhou, S. Shen, and P. Liu, "IoT Firmware Emulation and Its Security Application in Fuzzing: A Critical Revisit," *Future Internet*, vol. 17, no. 1, p. 19, Jan. 2025, doi: 10.3390/fi17010019.
- [18] M. Bley, T. Scharnowski, S. Wörner, M. Schloegel, and T. Holz, "Protocol-Aware Firmware Rehosting for Effective Fuzzing of Embedded Network Stacks," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '25. New York, NY, USA: Association for Computing Machinery, Nov. 2025, pp. 4484–4498. doi: 10.1145/3719027.3765125.
- [19] G. Venkataramani et al., "Firmware as the Next Frontier for Computing Security: Challenges and Opportunities," *IEEE Secur. Priv.*, pp. 2–12, 2026, doi: 10.1109/MSEC.2026.3709650.
- [20] H. Sun and Z. Ren, "A Binary Program Vulnerability Mining Method Based on Heuristic Hybrid Symbolic Execution," in *Proceedings of the 2025 4th International Conference on Intelligent Systems, Communications and Computer Networks*, in ISCCN '25. New York, NY, USA: Association for Computing Machinery, Jun. 2025, pp. 139–143. doi: 10.1145/3732945.3732965.
- [21] H. Tu, L. Jiang, and M. Böhme, "Vital: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 9, p. 288:1-288:26, Aug. 2026, doi: 10.1145/3786795.
- [22] L. Li, R. Ren, J. Dong, B. Zhang, and X. Kang, "A Systematic Literature Review of Software Vulnerability Mining Approaches Based on Symbolic Execution," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 35, no. 05, pp. 609–656, May 2025, doi: 10.1142/S0218194025300027.
- [23] B. Wan, S. Wang, Z. Wei, J. Huang, and C. Hu, "Binary Code Similarity Detection via LLM-Based Source Code Conversion," *IEEE Internet Things J.*, vol. 12, no. 24, pp. 51842–51853, Dec. 2025, doi: 10.1109/JIOT.2025.3579231.

- [24] Z. Zhao, T. Zhang, Q. Mao, Q. Zhao, X. Luo, and X. Fan, "BinEncoder: Binary code similarity detection across compilation configurations via microcode," *Appl. Intell.*, vol. 56, no. 11, p. 368, Jul. 2026, doi: 10.1007/s10489-026-07390-z.
- [25] L. Germano, R. Goldschmidt and J. Duarte "A Systematic Review on Detection, Repair, and Explanation of Vulnerabilities in Source Code Using Large Language Models | IEEE Journals & Magazine | IEEE Xplore." Accessed: Aug. 27, 2026. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/11237115>
- [26] Y. Yang, B. Xu, X. Gao, and H. Sun, "Context-Enhanced Vulnerability Detection Based on Large Language Models," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 8, p. 234:1-234:47, Jul. 2026, doi: 10.1145/3779222.
- [27] J. Huang, M. Wang, and Y. Hu, "FirmCCF: Detecting Custom Cryptographic Function Vulnerabilities Through Query-Driven Approaches," *IEEE Internet Things J.*, vol. 13, no. 2, pp. 2988–2999, Jan. 2026, doi: 10.1109/JIOT.2025.3631834.
- [28] H. J. Tay *et al.*, "Responsible Disclosure is a Two-Way Street: Empirically Measuring the Responsible Disclosure Contract in the Firmware Ecosystem," in *2026 IEEE Symposium on Security and Privacy (SP)*, May 2026, pp. 1–18. doi: 10.1109/SP63933.2026.00180.
- [29] J. Huang, J. Chen, Y. Xiao, Z. Qin, and Y. Hu, "FirmPass: Identifying Broken Password Management in Linux-Based IoT Firmware Through Query-Driven Approaches," *IEEE Internet Things J.*, vol. 12, no. 14, pp. 26911–26922, Jul. 2025, doi: 10.1109/JIOT.2025.3561520.

