ARTIFACT EVALUATED USENIX AVAILABLE

Bond: Constraint-Directed Fuzzing for Automated Validation of Taint Analysis Results in Linux-based IoT Firmware

Jiaqian Peng[*†‡], *Puzhuo Liu*[*§¶✉], *Kai Cheng*[†✉], *Zhaoteng Yan*[‡], *Jie Liu*[†]
Chengnian Sun[‖], *Hongsong Zhu*[†‡]
[†] *Institute of Information Engineering, Chinese Academy of Sciences*
[‡] *School of Cyber Security, University of Chinese Academy of Sciences*
[§] *Ant Group*
[¶] *Tsinghua University*
[‖] *University of Waterloo*

Abstract

Firmware vulnerabilities in IoT devices pose serious security threats, yet state-of-the-art taint analysis tools often generate large numbers of reports with limited validation. We present Bond, a directed fuzzing framework that bridges static taint analysis and dynamic vulnerability validation. Bond introduces constraint-guided input mutation by integrating three categories of constraints with six semantic types, enabling efficient exploration of paths associated with taint reports. We evaluate Bond on 19 IoT devices from 8 vendors, covering 2,776 taint reports produced by four state-of-the-art taint analyzers. Bond successfully validated 1,349 reports as real vulnerabilities, including 155 previously unknown vulnerabilities, of which 108 have been assigned CVE/PSV identifiers. On 60 known vulnerabilities, Bond achieved a 91.67% recall rate. Compared with four leading IoT fuzzers, Bond improves vulnerability validation by up to 5.5X. Ablation studies further demonstrate the effectiveness of Bond's key components and constraint extraction. These results establish Bond as a practical and effective framework for validating firmware taint analysis results.

1 Introduction

Static taint analysis[1] has become a mainstream method for detecting vulnerabilities in IoT firmware [1–10]. However, while taint analysis can efficiently flag potential flaws without executing code and with wide coverage, it also generates a large number of false positives (FPs), especially in complex embedded systems [11, 12]. With the exponential growth of IoT devices [13], now deployed across smart cities [14], industrial environments [15], and households [16], the volume and diversity of taint analysis reports have surged dramatically. This creates two pressing problems: ① analysts must spend substantial manual effort validating each report, leading to inefficiency and delays [17–20]; ② genuine vulnerabilities may be overlooked amid the noise of FPs. Automating the validation of taint analysis results thus holds both significant research interest and practical value.

Table 1: Firmware Taint Analysis Tool Output Information.

Tool	Action	Taint Source		Sink	Path		Constraint
		CTS	ITS		Function	Block	
Arbiter [21]	✗	✗	✗	✓	✓	✗	✗
Karonte [2]	✗	✓	✗	✓	✓	✓	✗
SaTC [3]	✘	✗	✓	✓	✓	✗	✗
Mango [4]	✗	✗	✓	✓	✓	✗	✗
OctopusTaint [5]	✗	✗	✓	✓	✓	✓	✗
EmTaint [6]	✗	✓	✘	✓	✓	✗	✗

`CTS`: Classical taint sources that are interface library functions directly receiving user input (e.g., `recv`, `read`).
`ITS`: Inferred intermediate taint source [10], a custom function that processes user input received via library functions and returns a part of the input to be used by other functions (e.g., `webGetVar`).

What taint analysis provides—and what is missing. As shown in Table 1, existing taint engines typically output three elements: (i) taint sources, (ii) sinks, and (iii) call traces or basic block paths between them. These outputs are useful but insufficient for proof-of-concept (PoC) construction. To validate a vulnerability, analysts must also (details in § 2):

- Identify entry points and parameters. Firmware exposes numerous hidden entry points through proprietary parsing and dispatching logic. Analysts must locate the correct interface and recover its parameter set, which is often spread across multiple functions.
- Extract path and field constraints. Vulnerabilities are often triggered only under specific conditions (e.g., a buffer overflow occurs only when `ipmode=static` and an overly long IP address is provided). Without precise constraints, execution diverges from vulnerable paths.

Limitations of existing approaches. Automated validation of IoT taint reports is not well supported by current techniques: Black-box fuzzing [22, 23] struggles with firmware validation due to inefficient mutation strategies that produce large num-

*The authors contribute equally to this paper.
✉Corresponding authors.
[1]We focus on static taint analysis only as dynamic taint analysis is usually not applicable to analyzing programs running in IoT devices. For brevity, taint analysis in this paper refers to static taint analysis.

bers of invalid test cases [24], and because it cannot model field-level semantic constraints [25–27]. Directed greybox fuzzing (DGF) [28–34] relies on execution feedback, which is hard to obtain in firmware. Even with rehosting [35–40], emulation fidelity remains low, causing frequent crashes and inaccurate detection. Symbolic execution [2, 3, 6] is hindered by path explosion and memory exhaustion, particularly in complex firmware with custom functions and libraries. Even hybrid approaches cannot eliminate FPs because they must compromise by ignoring constraints or imposing timeouts.

A New Approach via Constraint-directed Black-box Fuzzing. We argue that automated validation is feasible by combining taint reports with static analysis of firmware binaries to extract constraint information—including entry points, parameters, and path conditions—and then using these constraints to guide black-box fuzzing. This avoids the need for source code, dynamic instrumentation, or faithful emulation, and requires only the firmware binary, which is usually easy to obtain from vendor websites, upgrade packages, or device extractions [41].

Challenge. Despite this potential, automated validation faces two major obstacles: ① Lack of interface and parameter descriptions. Unlike software with explicit APIs or the `main` function [42, 43], firmware embeds entry points in hidden dispatch routines. Mapping taint sources to true entry points and their parameters is non-trivial. ② Lack of constraints and protocol information. Vulnerability triggering requires precise path constraints and compliance with often proprietary protocol specifications. Missing or mis-prioritized constraints easily divert execution, while non-compliant requests are rejected at the parser level, wasting fuzzing effort.

Bond. To address these challenges, we propose Bond, a constraint-directed black-box fuzzing framework for validating taint analysis results in IoT firmware. Bond consists of four key components: ① Entry point identification (§ 3.2.1). Traverses control flow graph (CFG) and call graph (CG) to locate dispatch structures (e.g., nested conditionals, function pointer tables, registration functions), extracts associated keywords, and cross-checks them with front-end observations. ② Reachable region partitioning (§ 3.2.2). For each source–sink pair, computes the reachable region between entry point and sink, enumerating relevant blocks and user-controllable parameters. ③ Constraint analysis (§ 3.3). Performs interprocedural data-flow analysis to extract constraints from branch predicates and library calls. Deviation basic blocks are used to derive mandatory constraints, partial constraints are preserved for path exploration, and unconstrained inputs are classified separately. ④ Directed black-box fuzzing (§ 3.4). Employs LLM-assisted template generation to ensure protocol compliance (capturing method, keyword position, and parameter format), and applies hierarchical mutation (mandatory → partial → non-constrained) to prioritize efficient vulnerability validation. Through this design, Bond enables scalable, automated validation of firmware taint analysis results—bridging the gap between taint reports and actionable vulnerability confirmation.

Evaluation. To rigorously validate the effectiveness of Bond, we conducted a comprehensive evaluation consisting of three parts (details in § 4): ① We purchased devices whose firmware had been used in prior state-of-the-art (SOTA) firmware taint analysis studies [3, 4, 6–8, 10, 44–47] (19 devices from 8 vendors) and automated the validation of taint analysis results. Bond successfully confirmed 1,349 out of 2,776 taint reports generated by SaTC, Mango, OctopusTaint, and EmTaint as real vulnerabilities. Importantly, this included 155 previously unknown vulnerabilities, all disclosed to vendors, among which 108 have already been assigned CVE/PSV identifiers. ② To further ensure reliability, we curated a dataset of 60 known vulnerabilities. Bond achieved a recall rate of 91.67% and an average vulnerability triggering time of 5m53s. Compared with four SOTA IoT fuzzing tools (GreenHouse [39], FIRM-AFL [36], SNIPUZZ [22], BooFuzz [48]), Bond consistently outperformed them, achieving up to 5.5X improvement in vulnerability discovery over the best-performing baseline. ③ We conducted ablation studies on Bond's three core components and its constraint extraction mechanism. The results clearly demonstrate the individual contributions of each component as well as the overall necessity of combining them for optimal performance.

Contribution. We make the following major contributions.

- We present Bond, the first directed fuzzing framework tailored for validating taint analysis reports in the context of IoT firmware, enabling effective black-box testing on real-world IoT devices.
- We design a constraint-directed fuzzing approach that combines three constraint categories with six semantic types to define the permissible mutation space and determine parameter mutation priorities, thereby enabling targeted and efficient path exploration.
- We perform a comprehensive evaluation of Bond on 19 real-world IoT devices. Results demonstrate that Bond surpasses mainstream IoT fuzzers in bug validation, successfully uncovering 155 0-day vulnerabilities. Following responsible disclosure, 108 CVE/PSV IDs have been assigned.

2 Motivation

Fig. 1 illustrates the motivation and challenges of validating taint analysis results in IoT firmware, using a real-world vulnerability from a D-Link router (simplified for clarity).

Known Information. Before validating an alert, we have the following: ① As shown in Table 1, the **taint report** provides three types of information: the source, the sink, and the call traces. ② Using the call traces, we can locate the relevant function **code snippet** by reverse-engineering the firmware binary. In Fig. 1, the identified bug involves a user-controlled `IP` field (Listing 1.10) that propagates to a `strcpy` call (Listing 2.15), leading to a buffer overflow. The bug-triggering

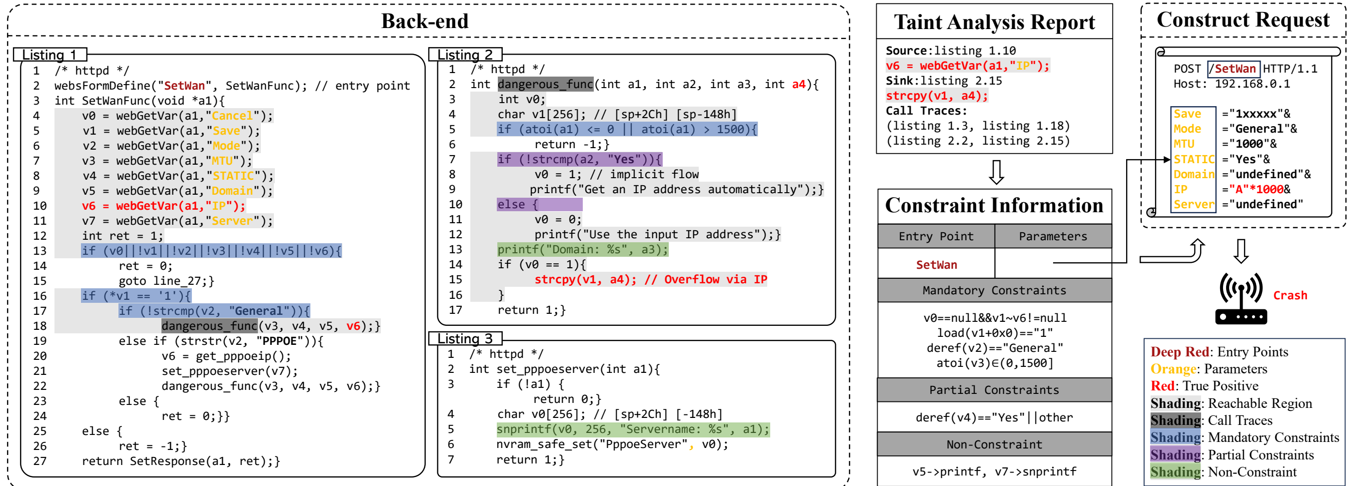


Figure 1: A vulnerability warning produced by EmTaint [6]. The example illustrates how a crafted input—guided by entry point inference and constraint modeling—can trigger a buffer overflow.

trace follows the sequence Listing 1.3 → 1.18 → 2.2 → 2.15. Crucially, if the execution path deviates (e.g., Listing 1.3 → 1.22), the vulnerable function `dangerous_func` is still reached, but the overflow at Listing 2.15 will not be triggered because the `v6` is sanitized by `get_pppoeip`. This highlights the importance of accurately validating traces rather than merely reaching sinks.

PoC Construction. To validate such a vulnerability, analysts must manually construct a PoC: ① Entry point identification. The starting point of the trace, `SetWanFunc`, is not directly accessible to users. Instead, it is invoked through a keyword-based routing mechanism, where the keyword `SetWan` is mapped to the handler via registration (`websFormDefine`, Listing 1.2). Only when `SetWan` is included in a POST request can the execution reach `SetWanFunc`. ② Parameter setting under constraints. The request must also satisfy multiple parameter constraints to follow the correct execution path. For instance, `Save`, `Mode`, `MTU`, etc., must be provided with valid values, `Cancel` must remain empty; otherwise, execution diverges and the vulnerability is not triggered.

Constructing such PoCs is time-consuming and labor-intensive. Since IoT firmware typically lacks both source code and dynamic execution visibility, traditional methods become ineffective. The scale of the problem is daunting: for the firmware in Fig. 1, EmTaint reports 169 alerts, and validating each requires 5+ hours of manual effort by an experienced analyst [18–20]. Existing techniques (e.g., directed greybox fuzzing for traditional software and kernels) rely heavily on source code access and runtime information, both of which are often unavailable in IoT firmware.

Bond. To bridge this gap, we propose Bond, a constraint-directed black-box fuzzing approach. Bond leverages taint analysis reports and reverse-engineered binary code to automatically infer and extract key constraints, which are then applied to guide fuzzing-based test case generation. This enables automated PoC construction and scalable validation of taint analysis results. However, realizing this vision requires overcoming two core challenges, which we detail below.

Challenge 1: Inference of entry points and corresponding parameters. Unlike traditional applications with explicit APIs, firmware hides entry points and parameter specifications inside proprietary parsing and dispatching logic. These routines often intertwine dispatch and data-processing code, making automated inference highly non-trivial. Accurate entry point identification is critical: it defines the true processing boundary and determines whether the correct set of parameters can be recovered—both prerequisites for effective static analysis and fuzzing.

Solution: Bond leverages the observation that entry point resolution logic typically appears at program boundaries in the form of nested conditionals, function pointer tables, or registered functions. Starting from source points marked in taint reports, Bond performs a boundary-aware backward traversal that alternates between the CFG and CG to recover candidate entry points. It then applies pattern matching to detect characteristic dispatch structures and extract their associated keywords, which are cross-checked with front-end observations to confirm true entry points. For each verified entry point, Bond conducts reachability analysis to delineate the reachable region and extract user-controllable parameters. For example, in Fig. 1, when backward traversal from Listing 1.10 misses nested branches, Bond resolves `SetWanFunc` through its registration in `websFormDefine`, matches the `SetWan` keyword, and confirms the correct entry point.

Challenge 2: Meeting Parameter Constraints and Protocol Compliance. Even after recovering entry points, validat-

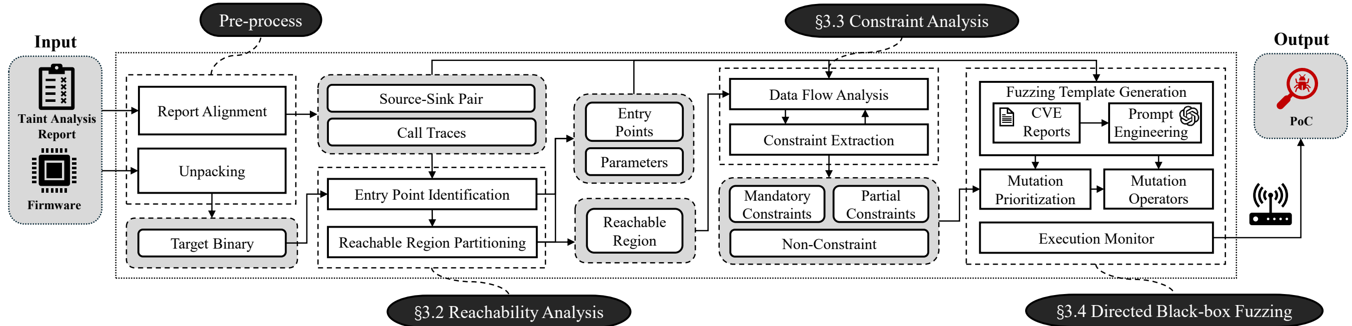


Figure 2: Workflow of Bond.

ing a specific source–sink report requires precise handling of parameters. The difficulties are threefold. *Constraint extraction:* A single parameter may be constrained at multiple locations; only a subset is relevant for vulnerability reachability. Inaccurate extraction can immediately divert execution. *Heterogeneous semantics:* Parameters can be governed by diverse constraint types, such as string equality, numeric ranges, or offset-based byte constraints, requiring semantic-aware modeling. *Mutation priorities and protocol compliance:* Efficient validation requires prioritizing promising parameters for mutation while avoiding parser-level rejections caused by vendor-specific protocol extensions.

Solution: Bond performs interprocedural data-flow analysis over the reachable region to track constraints imposed by branch predicates and library functions. It classifies constraints into: Mandatory constraints (deviation branches that must be satisfied, blue region in Fig. 1), Partial constraints (conditions guiding alternative paths, purple region in Fig. 1), Non-constraint (parameters not influencing predicates, green region in Fig. 1). Mutation priorities are then assigned accordingly: mandatory-constrained parameters are fuzzed first, followed by partial constraints, while unconstrained inputs are deferred. To maintain protocol compliance, Bond integrates LLM-assisted template generation that learns from CVE reports, vendor advisories, and prior PoCs, ensuring that generated requests respect protocol structures (capturing method, keyword position, and parameter format). For example, in Fig. 1, the entry point keyword appears in the standard URL path position, where the parameter fields follow the common key–value format.

Accurate recovery of entry points, parameter constraints, and protocol specifications is indispensable for validating taint analysis reports in IoT firmware. Unlike conventional software testing, IoT validation is performed in black-box settings on physical devices with limited throughput (about 1 request/sec [49, 50]) and no runtime introspection. Each invalid input incurs high opportunity cost, either being rejected outright or diverting execution into irrelevant states. Thus, errors in entry point recovery prevent reaching vulnerable code; imprecise constraint handling blocks vulnerability paths; and missing protocol specifications lead to parser-level rejection. Collectively, these issues make large-scale validation extremely resource-intensive—emphasizing the necessity of Bond's automated, constraint-directed approach.

3 Design of Bond

3.1 Overview

The overall workflow of Bond is illustrated in Fig. 2. Bond takes IoT firmware and taint analysis reports as input. It first aligns reports by standardizing outputs from different tools into a unified format containing (source, sink, call traces). Then, it extracts the file system, identifies file types, disassembles binaries, and collects auxiliary information such as CFG, CG, and function pointer tables from the `.data` section. After preprocessing, Bond proceeds to automatically validate taint reports through three key modules.

Reachability Analysis. This module identifies the target-relevant entry point, enumerates its associated user-controlled parameters, and determines the region reachable from the entry point to the sink. Starting from the sink in the taint analysis result, Bond performs an interprocedural backward traversal over the CFG and CG to locate parsing logic (nested comparisons or function dispatch) and uses pattern matching to collect candidate entry points. These are cross-checked with front-end keywords to confirm the actual entry point. Bond then computes the entry-to-sink reachable region through context-sensitive traversal and inspects additional callsites of taint source functions to recover parameter keywords, which are bound to their corresponding entry points for subsequent constraint analysis.

Constraint Analysis. For each target under validation, this module analyzes the parameters via lightweight interprocedural data-flow tracking, monitors both branch predicates and constraint-related library function calls (e.g., `strcmp`) and extracts input-dependent constraints. Constraints are classified into three categories based on whether the input affects any path condition and on the program location of the constraint: mandatory constraints (negated predicates that would exit

the reachable region), partial constraints (branch conditions within the reachable region), and non-constraint (parameters that never participate in conditional checks). They are further characterized into six semantic categories: string matching, numeric ranges, byte-level checks, null-value requirements, network formats, and heuristic semantic constraints. Together, these three constraint categories and six semantic types define the permissible mutation space for parameters and guide the prioritization of their mutations in the directed fuzzing phase.

Directed Black-box Fuzzing. This module combines LLM-assisted template generation with constraint-guided mutation to drive execution toward vulnerability-relevant paths in IoT web interfaces. For a given target device, the LLM produces standardized HTTP templates by extracting four structural elements—HTTP method, entry point keyword location, entry point keyword prefix, and parameter format—from publicly available PoCs and advisories of the target and related models. Mutation is performed in two phases with a hierarchical priority: parameters with mandatory constraints are tested first, followed by those with partial constraints, and finally unconstrained parameters. The first phase focuses on constrained parameters to quickly trigger shallow vulnerabilities, while the second phase expands the scope to all parameters to explore deeper execution paths.

3.2 Reachability Analysis

The reachability analysis module identifies entry points leading to sinks reported by the taint analysis tool. For each entry point, it extracts both the user-controlled parameters and the reachable basic blocks along the path to the sink.

3.2.1 Entry Point Identification

Entry points in IoT devices are often not directly exposed but are instead embedded within dedicated control parsing logic [51]. This logic interprets control fields in incoming requests (e.g., URL keywords) and dispatches execution to the corresponding handler. Based on the analysis of a large set of IoT firmware samples, we classify such parsing logic into two primary patterns: conditional-branch dispatching and function-pointer dispatching. The former implements its parsing logic within a single function, determining the target handler by matching one control field through nested branches (e.g., chained `if-else` or `switch` cases). In contrast, the latter resolves the keyword via a mapping mechanism that operates at the handler function level, implemented either through statically defined function pointer tables or through dynamically constructed mappings established by registration functions during system initialization.

Algorithm 1 shows the design of entry point identification. Bond begins with the source–sink pair from the taint analysis result and performs an interprocedural backward traversal over the target binary. The traversal alternates between the CFG and the CG, applying pattern matching to locate potential entry points. Specifically, beginning at the basic block containing the callsite of the taint source function that may trigger the vulnerability (line 2), Bond backtracks along CFG predecessors. If it encounters a conditional-branch basic block whose decision on one control field is part of a nested multi-level comparison structure (e.g., chained `strcmp`, `memcmp`), all subsequent paths associated with this nested branch structure are recorded as potential entry points (lines 11 to 13). If no match is found, the traversal switches to the CG and follows caller relationships. If a call target is resolved via a function pointer table or through a registration function, Bond also includes other functions in the same table and those registered via the same mechanism as candidates (lines 18 to 21). Otherwise, traversal returns to the caller's callsite in the CFG and repeats until entry points are identified (lines 24 to 27).

As part of this process, Bond extracts keywords associated with nested branches, function pointer tables, or registration functions identified above. These keywords are used to verify candidate entry points by checking whether they appear in a keyword set derived from front-end analysis. The front-end analysis collects keywords from control-related fields in web pages or scripts (line 1). A successful match confirms the entry point; otherwise, backtracking continues until a match is found or the analysis boundary is reached. These keywords serve as critical inputs for the subsequent directed fuzzing.

Algorithm 1: Entry Point Identification

```
Input:  R: Taint report with (src, sink)
        FFs: Front-end files
Output: EPs: Entry points
        EPKs: Entry point keywords
1  FKs ← FrontendAnalysis(FFs)
2  Worklist ← getBlock(sink)
3  VisitedF, VisitedB, EPs, EPKs ← ∅
4  while Worklist not empty do
5      block ← Pop(Worklist)
6      if block ∈ VisitedB then
7          continue
8      VisitedB ← VisitedB ∪ { block }
9      f ← getFunc (block)
10     while hasPred (f, block) do
11         block ← Pred (f, block)
12         if isNestedCompare (f, block) then
13             EPs, EPKs ← extractNestedCompare (f, block)
14             if KeywordMatching (EPKs, FKs) then
15                 return EPs, EPKs
16     if f ∉ VisitedF then
17         VisitedF ← VisitedF ∪ { f }
18         ref_locs ← getRefLocation (f)
19         foreach ref_loc ∈ ref_locs do
20             if isFuncDispatch (ref_loc) then
21                 EPs, EPKs ← extractFuncDispatch (ref_loc)
22                 if KeywordMatching (EPKs, FKs) then
23                     return EPs, EPKs
24         callsites ← getCallerSites (f)
25         foreach callsite ∈ callsites do
26             if callsite ∉ VisitedB then
27                 Worklist.Push(callsite)
```

3.2.2 Reachable Region Partitioning

Bond computes the reachable region for each source–sink pair identified in the taint analysis report to guide the subsequent constraint analysis. In this paper, the reachable region refers to the set of basic blocks that are control-flow reachable from the entry point to the sink location. Bond first checks the reachability between the entry point and the sink in the CG. Only if they are reachable does Bond proceed to compute the reachable region through an interprocedural traversal over both CFG and CG to enumerate basic blocks on paths to the sink. This traversal is recursive: for each function call inside the reachable region—including both custom and library functions—Bond continues to explore all functions that the call may invoke, expanding the reachable region until no new reachable basic blocks can be added. Note that taint analysis reports usually contain only source–sink pairs and their call traces, whereas constructing the reachable region also requires the call chain from the entry point to the source. For this part, Bond conservatively selects the shortest call chain to guide the reachable region computation [28].

During traversal, Bond also identifies other callsites of the taint source function within the reachable region, treating them as potential sources of user-controllable parameters that may affect constraints or vulnerability triggering. This design is motivated by the observation that such parameters—most notably the report's source and other optional inputs—are often introduced through repeated calls to the same taint source function [10]. For each identified callsite, Bond extracts its string argument as the keyword of the parameter. If multiple entry points are reachable to the same sink, the analysis is repeated for each so that subsequent directed fuzzing can handle them separately. Finally, all extracted parameters are bound to their corresponding entry points, while those outside the reachable region are excluded from analysis, as they typically cannot influence vulnerability triggering.

3.3 Constraint Analysis

The constraint analysis module identifies input-dependent constraints in the reachable region, determining the types and value ranges of user-controllable parameters relevant to reaching the target sinks.

3.3.1 Data Flow Analysis

For each confirmed entry point, Bond marks all user-controllable parameters within the reachable region as taint sources and performs lightweight interprocedural data-flow analysis. Unlike conventional taint analysis, which traces propagation to fixed sensitive sinks (e.g., `strcpy`), our analysis focuses solely on the influence of these sources on control-flow decisions, in order to identify path constraints that may affect reaching the target sink within the reachable region.

Algorithm 2: Data Flow Analysis.

Input: RR: Reachable region
Params: User-controllable parameters
Output: MCs: Mandatory constraints
PCs: Partial constraints
NCs: Non-constrained parameters

1 MCs, PCs, NCs $\leftarrow \emptyset$
2 **foreach** $p \in$ Params **do**
3   `SeedTaint` (p)
4   **foreach** *basic block* $b \in$ RR **do**
5   **foreach** *function call* f *in* b **do**
6   **if** `isConstraintFunc` (f) ***and*** `hasTaintedParam` (f, p) **then**
7   `applyTaint` (`ret` (f), p)
8   **if** `TaintedBy` (`BranchPredicate` (b), p) **then**
            // e.g., `load(`p` + offset)` or `ret` (f)
9   **if** $\exists e \in$ `Succ`$(b) : e \notin$ RR **then**
                // deviation basic block
10   MCs$[p] \leftarrow$ MCs$[p] \cup \{\bigwedge_{e \in \texttt{Succ_out}(b)} \neg\phi_e(p)\}$
11   **else**
12   PCs$[p] \leftarrow$ PCs$[p] \cup \{\phi_e(p) \mid e \in \texttt{Succ}(b)\}$
13   **if** $p \notin$ MCs $\cup$ PCs **then**
14   NCs $\leftarrow$ NCs $\cup \{p\}$
15 **return** MCs, PCs, NCs

Algorithm 2 outlines our data-flow analysis. During data-flow tracking, Bond first performs standard taint propagation (lines 3 to 4), monitoring tainted values after load operations to determine whether they participate in branch comparisons or other control-flow decisions (line 8), thereby establishing a mapping between input bytes and branch predicates. In addition, Bond leverages the fact that Linux-based IoT firmware frequently invokes libc APIs [7], many of which inherently embed constraint semantics (e.g., `strcmp`, `strtol`). When user input flows through such functions (line 6), Bond assigns a new taint label to their return values and continues propagation (line 7); when these tainted values appear in branch predicates, the corresponding conditions are recorded as potential path constraints (see § 3.3.2 for details).

Bond builds on the concept of deviation basic blocks introduced in WindRanger [30] by extending it to a global analysis of all relevant control-flow decisions. A deviation basic block is one that can reach at least one target site while at least one of its successors cannot, representing key decision points where execution may either proceed toward or deviate from the target. When such blocks are detected during data-flow analysis (line 9), Bond collects all branch predicates that would divert execution outside the reachable region RR—even when they reside in different basic blocks—and negates them (line 10). The negated predicates are then combined into mandatory constraints MCs that must hold throughout fuzzing, ensuring execution remains within the reachable region and avoiding unnecessary exploration. If a tainted branch predicate's successors are entirely within reachable region, Bond records each alternative path condition as a partial constraint PCs; these often mutually exclusive pairs are later used to guide coverage-based exploration within the reachable re-

gion, reducing scheduling bias and increasing the likelihood of uncovering complex vulnerabilities. Finally, if a parameter never influences any branch predicate (line 14), it is recorded as non-constrained NCs.

The complete constraint set, consisting of mandatory constraints, partial constraints, and non-constrained parameters, together with their associated entry points and parameters, is then passed to the mutation-prioritization module in the directed testing phase to guide input generation and scheduling.

3.3.2 Constraint Extraction

During data-flow analysis, we collect conditions under which parameters affect control-flow decisions within the reachable region and group them by underlying semantics. To ensure these groups are representative and practical for Linux-based IoT firmware, we conducted an empirical study of 200 firmware-related CVEs reported on the NVD [52] over the past two years. For each vulnerability, we examined the triggering conditions and distilled recurring patterns into six representative semantic types, each with targeted handling strategies to extract, model, and satisfy them during fuzzing:

- **String-matching constraints:** These constraints originate from string operations, including exact matches (e.g., `strcmp`, `strcasecmp`) and partial matches (e.g., `strchr`, `index`). Exact-match strings are directly incorporated into the parameter dictionary and periodically selected during fuzzing to maintain diversity. For partial matches, we construct lightweight symbolic models of the corresponding processing functions and employ the Z3 SMT solver [53] to solve the derived constraints, obtaining multiple satisfying values, which are subsequently subjected to mutation to further expand input diversity.
- **Numeric-parsing constraints:** Functions such as `atoi` and the `strtol` family impose numeric range restrictions on parameters. We convert branch conditions into Disjunctive Normal Form (DNF), parse atomic predicates into bounded ranges with explicit exclusions, and proportionally allocate candidate values across these ranges. For unbounded cases, finite representative values are selected.
- **Byte-level constraints:** Comparisons of specific byte offsets or bit fields in the input against fixed constants (e.g., `input[3] == 0x61`) are recorded by mapping each constrained offset to its required value. During fuzzing, constrained bytes are preserved, and mutations are applied only to other positions.
- **Null-value constraints:** Such constraints specify that a given parameter should be absent, for instance when two parameters correspond to mutually exclusive operations (e.g., `add` and `edit`), permitting only one to occur in the request. They often originate from comparing a dereferenced parameter to the `\0` terminator. During fuzzing, we enforce them by removing the corresponding parameter keywords from the generated requests.
- **Network-format constraints:** Such constraints enforce network-related data formats (e.g., IPv4 validated by the `inet_aton` family). During fuzzing, we satisfy them by supplying syntactically valid address values.
- **Heuristic semantic constraints:** For certain constraints that are also difficult to capture with taint alone, such as separator stripping, case folding, or hexadecimal parsing where byte-level normalization obscures the intended semantics, we apply heuristic keyword matching based on semantic cues in parameter names to complement static analysis. For instance, parameters with names containing `mac` are mapped to their expected format in a knowledge base and supplied with expert-defined dictionary values during fuzzing to bypass format checks. A concrete example is provided in Appendix A, illustrating how the heuristic constraints satisfy the complex checks in `checkmacaddr`.

3.4 Directed Black-box Fuzzing

The directed black-box fuzzing module targets IoT device web interfaces, leveraging entry points, parameters, and constraints to prioritize mutations on fields affecting path reachability and efficiently explore vulnerability-triggering paths.

3.4.1 Fuzzing Template Generation

In generating HTTP request templates for fuzzing IoT devices, incorrect protocol structures can cause requests to be discarded early, blocking deeper execution. Our study of the HTTP RFC specification [54] and real IoT web traffic shows that the most variable and validity-critical elements are concentrated in four locations: HTTP method; entry point keyword location (e.g., appearing as a prefix in the URL path, or in the body alongside other parameters marked by specific keywords such as `action=setwan`); entry point keyword prefix (fixed strings such as `/goform/` or `/cgi-bin/`); and parameter format (commonly &-separated key=value pairs, or formats like JSON and XML). For other elements, including common headers and origin or referer fields, a general template can be applied consistently across devices.

Inspired by the capability of LLMs to infer protocol structures from diverse information sources [55], we adopt an LLM-assisted approach to generate fuzzing templates. Our prompts instruct the LLM to autonomously search for publicly available PoCs and HTTP request examples for the target device—for example, from NVD, VulDB, GitHub repositories, or security research blogs. If device-specific examples are unavailable, the retrieval process expands to sibling devices that share the same codebase, inferred through CVE or security-advisory co-occurrence. For instance, when a CVE affects both Netgear R8500 and CBR40, and no PoC is available for R8500, the system directly uses the CBR40 PoC as a substitute. Based on the collected PoC examples, the LLM automatically parses HTTP requests and identifies key vari-

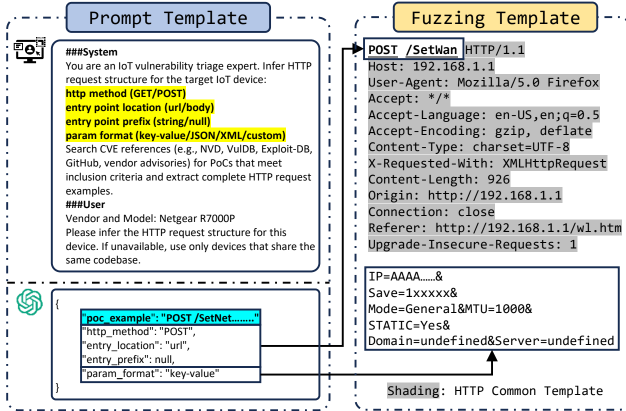


Figure 3: LLM-assisted fuzz template generation.

able slots, including the HTTP method, entry point keyword location, entry point keyword prefix, and parameter format, and then generates a structured HTTP fuzzing template with annotated components. This procedure is lightweight: each device requires only a single template generation, which is reused throughout the entire directed fuzzing workflow, where only the variable slots are replaced with device-specific entry keywords or parameter key–value pairs.

As illustrated in Fig. 3, the LLM retrieves publicly available PoC examples (e.g., `POST /SetNet...`) from GitHub repositories linked to a specific CVE via web search. The LLM then analyzes the HTTP request and identifies all variable slots. In this example, the HTTP method is set to `POST`, the entry point keyword is located in the header without any prefix (i.e., containing only `/`), and the payload follows a common key–value format. Based on this analysis, the LLM generates a reusable fuzzing template in which these slots are explicitly annotated, and we only need to replace these slots with the entry points and parameter values required to validate a given taint analysis report. For instance, the PoC entry point `/SetNet` is replaced with `/SetWan`, and the parameter fields are filled with corresponding values (e.g., `IP=AAAA...`).

3.4.2 Mutation Prioritization

To expedite the triggering of alerts reported by taint analysis, we employ a two-phase mutation strategy with a hierarchical prioritization scheme. Our observation is that parameters governed by explicit path constraints are generally more effective for vulnerability discovery than unconstrained ones. Partial constraints are not always relevant to vulnerability activation, becoming meaningful only when the activation involves complex data dependencies or implicit flows. Accordingly, the prioritization hierarchy is defined as: parameters with mandatory constraints, followed by those with partial constraints, and finally those with no constraints.

In the first phase, non-constrained parameters are directly assigned fixed initial values to reduce the search space, while parameters with mandatory and partial constraints are mutated and exhaustively combined. Mandatory constraints usually define a small, well-delimited value domain that directly determines reachability; we therefore perform limited yet diverse sampling across this domain. Partial constraints specify a preferred subset of values within the overall admissible domain; to balance diversity and efficiency, we proportionally sample from both the preferred subset and the remaining admissible values. More specifically, Bond adopts a fixed black-box mutation strategy with gradient-based proportional sampling. If the value space is small (<10), it tests all values; for larger spaces (e.g., 1–1000), it samples without replacement at a ratio of 1:100. This design aims to quickly trigger shallow vulnerabilities that can be activated without exploring large search spaces.

If parameter combinations have been exhaustively enumerated and no vulnerabilities are triggered, the second phase is initiated. In this phase, we expand the mutation scope to include non-constrained parameters and relax the mutation range for partial constraints, while maintaining the mutation strategy for mandatory constraints. This phase targets deeper and more complex vulnerabilities whose activation depends on intricate path constraints, or on parameters that lack explicit constraints due to imprecision in the static analysis.

3.4.3 Mutation Operators

For parameters governed by path constraints, we leverage the solutions derived in § 3.3.2 to guide dictionary selection or range-based sampling. Unconstrained parameters are instead mutated from an initial seed value using havoc-style mutations. For alerts classified under the buffer overflow category, we generate overly long strings and introduce random separators to bypass potential field-specific checks, as well as insert combinations of numbers and spaces or IP addresses and spaces to bypass functions such as `atoi` or `inet_addr`. For alerts classified under the command injection category, we fill the parameter with commands such as `telnetd` or `ping`, which can quickly trigger explicitly observable responses. It is worth noting that such vulnerability-oriented mutations are never applied to parameters outside the source–sink mapping, as this could inadvertently trigger vulnerabilities in unrelated parameters, leading to special false positives.

4 Evaluation

We evaluate Bond on real-world IoT firmware and compare it against SOTA tools to address the following three research questions:

- **RQ1**: To what extent can Bond accurately validate the reports produced by taint analysis tools?
- **RQ2**: How does Bond perform in terms of effectiveness and efficiency compared to SOTA fuzzing tools in reproducing known vulnerabilities?

Table 2: Evaluated devices.

ID	Vendor	Model	Version(1-day/0-day)	Type
1	Netgear	R8500	/1.0.2.160	WiFi Router
2	Netgear	R7000P	1.3.0.8,1.3.1.64/1.3.3.154	WiFi Router
3	Netgear	XR300	1.0.3.72/1.0.3.78	WiFi Router
4	Netgear	WNDR4500	/1.0.1.46	WiFi Router
5	Netgear	EX6120	1.0.0.68/1.0.0.70	Range Extender
6	Netgear	D6400	/1.0.0.114	DSL Modem
7	Linksys	E1700	1.0.0.4.003/1.0.0.4.003	WiFi Router
8	Linksys	RE6500	1.0.010.001/1.0.013.001	Range Extender
9	D-Link	DIR816	1.10CNB03/1.10CNB05	WiFi Router
10	D-Link	DIR619L	2.06B01/2.06B01	WiFi Router
11	D-Link	DNS320L	/1.11	Network Attached Storage
12	D-Link	DCS932L	1.00/2.18.01	IP Camera
13	TOTOLink	A3002R	V1.1.1/V1.1.1	WiFi Router
14	TOTOLink	CA300-PoE	V6.2c.884/V6.2c.884	Wireless Access Point
15	Tenda	AX3	V16.03.12.10/V16.03.12.10	WiFi Router
16	Tenda	O3V2	1.0.0.12/1.0.0.12	Outdoor Wireless Bridge
17	TP-Link	TL-WR841N	V11_160325/V11_160325	WiFi Router
18	Belkin	F9K1122	1.00.23/1.00.33	Range Extender
19	Motorola	CX2L	1.0.1/1.0.1	WiFi Router

- **RQ3**: What is the individual impact of each major component within Bond on the overall system performance?

4.1 Experiment Setup

Implementation. We implemented a prototype of Bond with 3.3K lines of core Python code, including reachability analysis (1.1K LoC), constraint analysis (0.8K LoC), and directed black-box fuzzing (1.4K LoC). Bond unpacks firmware with Binwalk [56], reconstructs CG/CFG and metadata in IDA Pro [57], and lifts assembly to VEX IR for uniform analysis. We extend EmTaint's [6] interprocedural data-flow engine to support our constraint analysis, adding tracking of input-dependent branch predicates and constraint-related library function calls. The directed fuzzer builds on BooFuzz [48] with redesigned low-level mutation operators and a priority scheduler tailored to taint analysis report validation. For LLM-assisted protocol template synthesis, we use GPT-4o [58].

Dataset. The dataset consists of two parts. ① To validate the results of firmware static taint analysis, based on SOTA research [3, 4, 6–8, 10, 44–47], we purchased 19 various types of real-world IoT devices from 8 major vendors (e.g., Netgear and Linksys) (details in Table 2), under the context of regional restrictions and device retirement. ② To demonstrate the effectiveness of Bond, we compared it with SOTA IoT fuzzing tools based on a collection of 60 real, 1-day CVE vulnerabilities (details in Table 8).

Configurations. All experiments were conducted on a Windows machine with an Intel Core i9-11950H CPU (8 cores, 16 threads) and 64 GB of RAM.

Time Metric. All experiments report validation time as the total end-to-end duration. It includes both the static analysis stage (entry point, parameter, and constraint extraction) and the subsequent fuzzing phase.

4.2 Validating Taint Analysis Reports-RQ1

This section evaluates Bond's effectiveness in validating taint analysis reports on real-world IoT devices. We use several representative open-source tools widely adopted in IoT security research—SaTC [3], Mango [4], OctopusTaint [5], and EmTaint [6]. Our goal is not to compare their taint analysis accuracy, but to assess how many of their alerts can be automatically validated by Bond's directed fuzzing framework.

4.2.1 Results Analysis

We conducted experiments on the latest firmware versions of all 19 devices in our dataset. Given the large number of taint analysis alerts, we imposed a time limit of one hour per report for validation. In total, Bond processed 2,776 alerts from the four tools, as summarized in Table 3. Of these, 1,349 were successfully triggered, yielding a 48.6% validation rate. Single-threaded execution time reaches 35 days, and 80% of alerts completed end-to-end validation within 2 minutes, fully automated without the need for manual or carefully designed test cases. For the remaining 20% of alerts, Bond completes validation within 2 minutes to 1 hour. These cases typically involve complex inputs with many parameters or suffer from static analysis imprecision, which together enlarge the search space and increase the time required to trigger the vulnerable execution path.

Table 3: Summary of large-scale validation results.

	Alerts	Validated	%	CPU Time
Bond	2776	1349	48.60%	35 days

Table 4 shows the detailed results: SaTC, Mango, OctopusTaint, and EmTaint produced 4, 193, 1,273, and 1,306 alerts, with Bond achieving validation rates of 25%, 18.13%, 35.74%, and 65.24%, respectively. Bond is not tailored to these tools; any future or more advanced taint analysis framework can integrate by exporting (source, sink, call-trace) triplets, enabling automated validation without extra engineering.

Vulnerability Validation. Given the complexity of embedded environments and networks, we replayed each PoC generated by Bond that triggered crashes or command execution. A PoC was deemed a true positive (TP) only if it could reliably reproduce abnormal behaviors (e.g., process crash/restart for buffer overflows, or successful telnet access for command injection).

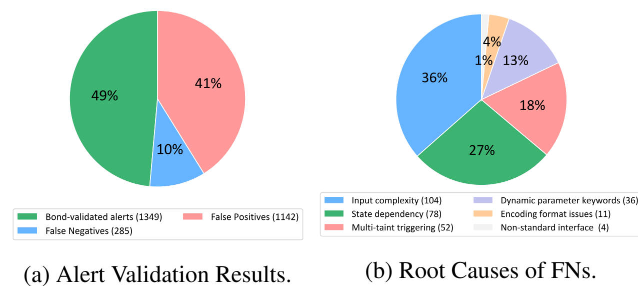
Vulnerability Analyst. For the remaining alerts that were not automatically validated, four experienced security analysts conducted a thorough manual review. Three are the first three authors of this paper, and the fourth is a security engineer from our institution; all have over five years of vulnerability analysis experience. All analysts participated voluntarily and without compensation, and no external participants were recruited for this study. This process, which consumed several

Table 4: Detailed results for large-scale validation of taint analysis reports. In the table, x indicates that the taint analysis tool does not support the corresponding firmware, while - denotes that the tool produced no alerts for that firmware.

ID	Vendor	Product	Version	SaTC			Mango			OctopusTaint			EmTaint		
				Alerts	Validated	Failed	Alerts	Validated	Failed	Alerts	Validated	Failed	Alerts	Validated	Failed
1	Netgear	R8500	1.0.2.160	1	0	1	35	2	33	50	11	39	17	5	12
2	Netgear	R7000P	1.3.3.154	1	0	1	29	0	29	50	4	46	13	2	11
3	Netgear	XR300	1.0.3.78	1	0	1	3	0	3	50	2	48	13	1	12
4	Netgear	WNDR4500	1.0.1.46	x	x	x	-	-	-	174	18	156	150	35	115
5	Netgear	EX6120	1.0.0.70	-	-	-	7	0	7	81	0	81	2	0	2
6	Netgear	D6400	1.0.0.114	x	x	x	65	5	60	50	3	47	8	5	3
7	Linksys	E1700	1.0.0.4.003	-	-	-	-	-	-	179	109	70	178	146	32
8	Linksys	RE6500	1.0.013.001	-	-	-	2	1	1	157	86	71	252	151	101
9	D-Link	DIR816	1.10CNB05	x	x	x	7	4	3	120	89	31	94	79	15
10	D-Link	DIR619L	2.06B01	x	x	x	-	-	-	24	19	5	169	123	46
11	D-Link	DNS320L	1.11	-	-	-	-	-	-	-	-	-	-	-	-
12	D-Link	DCS932L	2.18.01	-	-	-	-	-	-	24	0	24	6	1	5
13	TOTOLink	A3002R	V1.1.1	x	x	x	45	23	22	122	32	90	47	27	20
14	TOTOLink	CA300-PoE	V6.2c.884	x	x	x	-	-	-	x	x	x	19	15	4
15	Tenda	AX3	V16.03.12.10	1	1	0	-	-	-	102	25	77	92	70	22
16	Tenda	O3V2	1.0.0.12	x	x	x	-	-	-	76	52	24	59	49	10
17	TP-Link	TL-WR841N	V11_160325	x	x	x	x	x	x	x	x	x	30	21	9
18	Belkin	F9K1122	1.00.33	x	x	x	-	-	-	2	0	2	129	101	28
19	Motorola	CX2L	1.0.1	x	x	x	x	x	x	12	11	1	28	21	7
Total	\	\	\	4	1(**25%**)	3	193	35(**18.13%**)	158	1273	461(**36.21%**)	812	1306	852(**65.24%**)	454

hundred hours, included analyzing taint propagation paths and attempting to manually construct PoCs for each alert.

Based on the results, we classified untriggered alerts as either false negatives (FNs) or false positives (FPs). As shown in Fig. 4a, 285 alerts were confirmed triggerable through manual efforts, revealing that Bond missed some TPs due to fuzzing limits. The remaining 1,142 alerts remained untriggerable despite exhaustive attempts, indicating they are likely FPs caused by imprecision in the taint analysis tools. Although the classification may involve some uncertainty due to the absence of complete ground truth, each alert was independently examined by multiple analysts, and disagreements were resolved through further inspection.



(a) Alert Validation Results. (b) Root Causes of FNs.

Figure 4: Overall validation results and root causes of FNs.

FNs. We manually analyzed the untriggered alerts from Bond's automated validation and classified the causes into six categories, as shown in Fig. 4b. The details are as follows:

- **Input complexity (104 cases)**: Complex byte-level operations or data flows across multiple function calls, which lead to imprecise constraint tracking and a large mutation space, making it difficult to trigger the vulnerability within the allotted time.
- **State dependency (78 cases)**: Some vulnerabilities require specific runtime states (e.g., IPv6 enabled, certain NVRAM configurations). Bond performs fuzzing under default configuration but can successfully trigger these alerts once the required conditions are manually set.
- **Multi-taint triggering (52 cases)**: Certain vulnerabilities depend on multiple inputs (e.g., cumulative field length exceeding buffer capacity). Bond mutates only reported sources to maintain directionality, avoiding applying aggressive mutations to unrelated parameters, which could otherwise trigger unintended vulnerabilities and violate the target source-sink pairing.
- **Dynamic parameter keywords (36 cases)**: Some parameter keywords are constructed at runtime (e.g., string concatenation), making them hard to recover statically and preventing valid input generation.
- **Encoding format issues (11 cases)**: Certain inputs undergo front-end encoding (e.g., base64) and are decoded later; without recognizing this, fuzzing mutates only the encoded form, limiting effectiveness.
- **Non-standard interface (4 cases)**: A few interfaces bypass typical parsing entry points and are thus missed during static analysis.

FPs. We analyzed the root causes of FPs across tools. EmTaint more accurately models aliases, function behavior, and data flow, reducing the propagation of FPs. In contrast, OctopusTaint lacks accurate modeling for functions such as `sprintf`, `atoi`, and `malloc`, leading to over- or undertainting. Mango prioritizes scalability, but insufficient interprocedural modeling causes most of FPs. For SaTC, the number of alerts is too small to characterize common FP patterns.

Additionally, all tools struggle to model sanitization functions. For instance, recent Netgear firmware introduces custom sanitizers that effectively remove taint along critical paths

but are not captured by existing models. These limitations lead to infeasible source-to-sink paths being reported. Notably, OctopusTaint produced thousands of alerts on four Netgear devices; we randomly selected 50 alerts from these for validation in this experiment.

4.2.2 New Vulnerability Discovery

Bond successfully discovered 155 previously unknown (0-day) vulnerabilities through large-scale automated validation of taint analysis reports: 35 command injection and 120 buffer overflow vulnerabilities (detailed in Table 5). To distinguish 0-day vulnerabilities from previously known ones, we systematically compared each validated PoC against public vulnerability databases (e.g., NVD [52]) and vendor security advisories by matching affected products, firmware versions, and vulnerability semantics (source–sink pairs). A vulnerability was considered 0-day if no matching record existed at the time of discovery.

As of this writing, 108 have been assigned CVE/PSV identifiers. Based on impact assessment, 3 were classified as critical, enabling remote code execution without authentication and allowing full device takeover. Another 68 were categorized as high severity, requiring authentication but still posing substantial risks given the prevalence of weak default credentials in IoT devices [59]. Finally, 37 were rated medium severity, leading to crashes or denial-of-service conditions.

Notably, we identified many 0-day vulnerabilities on devices with IDs 1, 2, 3, 9, 10, 15, 17, and 19—despite these targets having been widely examined in prior community efforts and static analysis research [3, 4, 6–8, 10, 44–47]. This highlights the limitations of manual PoC generation and further demonstrates Bond's ability to automatically uncover deep, hard-to-trigger vulnerability paths guided solely by taint analysis reports. We detail a representative case in Appendix A to illustrate this complexity.

Table 5: 0-day vulnerabilities discovered by Bond. `BoF` indicates buffer overflow vulnerability, while `CI` denotes command injection vulnerability.

Vendor	Product	Type	Bug IDs
TP-Link	TL-WR841N	BoF	CVE-2025-53711 to CVE-2025-53715 4 unassigned
Linksys	E1700	BoF	CVE-2025-9525 to CVE-2025-9527
Linksys	E1700	CI	CVE-2025-9528
Linksys	RE6500	BoF	CVE-2025-9245 to CVE-2025-9253 CVE-2025-9355 to CVE-2025-9363 CVE-2025-9392 to CVE-2025-9393 CVE-2025-9481 to CVE-2025-9483 CVE-2025-8833, CVE-2025-14133 2 unassigned
Linksys	RE6500	CI	CVE-2025-5438 to CVE-2025-5447 CVE-2025-9244, CVE-2025-9575
D-Link	DIR816	BoF	CVE-2025-5623 to CVE-2025-5624 CVE-2025-5630, 1 unassigned
D-Link	DIR816	CI	1 unassigned
D-Link	DIR619L	BoF	CVE-2025-6114 to CVE-2025-6115 CVE-2025-6367 to CVE-2025-6374 CVE-2025-6614 to CVE-2025-6617 CVE-2025-55599, CVE-2025-55602 CVE-2025-55611, 8 unassigned
D-Link	DCS932L	CI	CVE-2025-5571
TOTOLink	A3002R	BoF	CVE-2025-6486 to CVE-2025-6487 7 unassigned
TOTOLink	A3002R	CI	CVE-2025-6485, 1 unassigned
TOTOLink	CA300-PoE	CI	CVE-2025-6618 to CVE-2025-6621 1 unassigned
Tenda	AX3	BoF	CVE-2025-55603, CVE-2025-55605 CVE-2025-55606
Tenda	O3V2	BoF	CVE-2025-7416 to CVE-2025-7423 CVE-2025-55613, 1 unassigned
Tenda	O3V2	CI	CVE-2025-7414 to CVE-2025-7415
Belkin	F9K1122	BoF	CVE-2025-7084 to CVE-2025-7094
Belkin	F9K1122	CI	CVE-2025-7081 to CVE-2025-7083
Netgear	R8500	BoF	PSV-2022-0181 to PSV-2022-0182 1 unassigned
Netgear	R7000P	BoF	CVE-2024-51004, CVE-2024-51013
Netgear	XR300	BoF	CVE-2024-51016
Netgear	WNDR4500	BoF	16 unassigned
Netgear	D6400	CI	CVE-2025-7407
Motorola	CX2L	CI	4 unassigned
Total	\	\	**108 assigned, 47 unassigned**

CVE: the vulnerability IDs assigned by CNAs (TP-Link Systems Inc., Netgear, VulDB).
PSV: the vulnerability IDs assigned by Netgear.
Unassigned: the vendor has confirmed the vulnerability but needs to fix it before releasing details.

Vulnerability Disclosure. We have responsibly disclosed all discovered 0-day vulnerabilities to the corresponding vendors. For TP-Link [60], which serves as a CVE Numbering Authorities (CNAs) [61], we contacted the vendor directly via email; they responded promptly and immediately assigned CVE IDs. For Linksys and Netgear, we communicated through their vendor-recommended Bugcrowd [62, 63] platforms and requested CVE assignments only after the vulnerabilities were confirmed and patched. For D-Link, the vendor acknowledged the reported vulnerabilities but stated that the affected devices had reached end-of-life (EOL) and no patches would be released. For Belkin, Tenda, and TOTOLink, we contacted the vendors via email but did not receive valid responses within the 90-day disclosure window. We subsequently reported the vulnerabilities to the VulDB [64], which handled vendor communication and CVE assignment. Unfortunately, these vendors have still not responded. For Motorola, we are currently coordinating with Lenovo (the CNA for Motorola) to facilitate remediation.

4.2.3 Fidelity

To evaluate the fidelity of Bond's directed execution, we selected four devices (ID 4, 8, 13, and 16) with root shell access and manually examined 40 PoCs automatically generated by Bond —10 per device, including 10 command injection and 30 buffer overflow cases. These PoCs were executed on real devices under remote debugging using gdbserver and gdb, enabling detailed trace collection. For each case, we were able to map the observed taint sources to the corresponding input fields, observe the activated sinks, and collect runtime call traces to verify that Bond faithfully followed the (source, sink, call traces) triplets from the taint analysis reports.

Out of the 40 PoCs, 38 followed the intended execution path as indicated by the taint analysis reports. In one of the two exceptions, the PoC diverged from the target sink and instead triggered a vulnerability on a parallel branch that was preferred due to the mutation strategy. In the other case, the PoC reached the sink but the taint propagated further, which ultimately led to a vulnerability in a downstream callee and caused an overwrite at a deeper return address.

4.2.4 Firmware Diversity

To better understand the complexity and diversity of firmware, and to demonstrate the effectiveness of Bond 's static analysis, we summarize the entry points, parameters, and constraints extracted from each firmware in the dataset (Table 4). Entry points denote the total number of distinct entry points identified by our detection module. For each firmware, we compute the number of involved parameters and extracted constraints in each taint analysis report validated by Bond, and then average these values over all reports for that firmware. Finally, we compute the mean and variance of entry points, parameters, and constraints across all firmware in the dataset.

Table 6 shows that Bond consistently identifies a large number of entry points, averaging 158.17 per-firmware. This number varies across different firmware images, with a variance of 4347.81. The average number of parameters involved in taint analysis reports also varies across firmware samples, with a variance of 74.54. Moreover, Bond extracted the corresponding constraints for an average of 52.43% of the parameters, with a variance of 226.86 for different firmware. The above results demonstrate the complexity and diversity of the firmware in the dataset, and the widespread presence of semantic constraints within real-world firmware, further illustrating the effectiveness of Bond's design.

Table 6: Per-firmware Statistics. C/P denotes the proportion of parameters that are constrained. The symbol - denotes that the tool produced no alerts for that firmware.

ID	Vendor	Product	Entry Points	Parameters	Constraints	C/P
1	Netgear	R8500	177	20.21	12.70	62.84%
2	Netgear	R7000P	235	20.32	12.71	62.55%
3	Netgear	XR300	190	16.28	9.72	59.71%
4	Netgear	WNDR4500	165	18.79	10.12	59.71%
5	Netgear	EX6120	144	15.74	7.69	59.71%
6	Netgear	D6400	159	11.43	4.81	42.08%
7	Linksys	E1700	124	22.60	15.38	42.08%
8	Linksys	RE6500	162	16.76	10.57	63.07%
9	D-Link	DIR816	340	9.01	4.14	45.95%
10	D-Link	DIR619L	129	22.11	17.76	80.33%
11	D-Link	DNS320L	-	-	-	-
12	D-Link	DCS932L	268	12.67	3.04	23.99%
13	TOTOLink	A3002R	61	22.37	16.94	75.73%
14	TOTOLink	CA300-PoE	77	4.81	1.81	37.63%
15	Tenda	AX3	112	6.87	2.51	37.63%
16	Tenda	O3V2	101	9.75	2.73	28.00%
17	TP-Link	TL-WR841N	151	22.20	10.29	28.00%
18	Belkin	F9K1122	130	41.60	24.73	59.45%
19	Motorola	CX2L	122	3.45	1.68	48.70%
Mean	\	\	**158.17**	**16.50**	**9.41**	**52.43%**
Variance	\	\	**4347.81**	**74.54**	**40.19**	**226.86**

4.3 Reproducing Known Vulnerabilities-RQ2

This section evaluates Bond's ability to reproduce known vulnerabilities. From CVE reports and vendor advisories, we collected 60 well-documented 1-day cases on our target devices. For each, we extracted the corresponding source and sink and provided them to Bond to verify whether it could successfully trigger the vulnerability.

Comparison Tools. We compare Bond with four representative open source fuzzers in the IoT field: GreenHouse [39], FIRM-AFL [36], SNIPUZZ [22], and BooFuzz [48]. Among them, GreenHouse and FIRM-AFL are emulation-based fuzzers, while SNIPUZZ and BooFuzz represent typical black-box fuzzing approaches. Labrador [23] is excluded as it is not publicly available. We acknowledge that such comparisons are inherently difficult to make entirely fair, as the evaluated fuzzers primarily adopt general-purpose coverage-guided strategies and do not rely on specific taint sources or sink locations. Since none of the four fuzzers support entry point identification, we used Selenium [65] to replay real HTTP requests, simulating realistic user interactions and providing initial seeds to unify the starting point for all fuzzers. Each experiment was repeated ten times, and the average time-to-exposure in seconds (Avg.TTE/s) were recorded.

4.3.1 Effectiveness and Efficiency

Table 7 shows that Bond successfully validated 55 of 60 known vulnerabilities within 24 hours, achieving a recall rate of 91.67%—5.5× more than the best SOTA baseline—while maintaining a fast average trigger time of 5m53s. 49 cases were triggered within 2 minutes (detailed in Table 8). Bond significantly reduces the input space through constraint analysis, achieving higher efficiency on most vulnerabilities. Although it introduces LLM invocation and symbolic solving, the overhead is minimal. The LLM is invoked only once per device to generate the fuzzing template. The Z3 solver handles only lightweight symbolic expressions derived from string modeling rather than complex path constraints, making the overhead negligible (below 0.05 seconds).

Table 7: Summary of known vulnerability validation results.

Metrics	GreenHouse	FIRM-AFL	SNIPUZZ	BooFuzz	Bond
Validated Vulns	1	0	0	10	55
Recall	1.67%	0%	0%	16.7%	91.67%
Avg.TTE/s	15h45m49s	-	-	25s	5m53s

4.3.2 Compare to SOTA

GreenHouse discovered only one vulnerability (DCS-932L) on five devices that can be emulated, requiring 15h45m49s. Manual validation revealed a memory corruption in HTTP header parsing, not the intended target. We discovered that its Havoc mutation strategy severely disrupted the original

Table 8: Detailed results for validation of known vulnerabilities , presenting the corresponding firmware and version information for each CVE-ID along with the execution results of each fuzzing tool . The symbol `x` indicates that the fuzzing tool does not support the corresponding firmware. The symbol - denotes that no vulnerabilities were triggered within 24 hours. `Factor` is the ratio of time used by a tool compared to that of Bond.

CVE-ID	Vendor	Product	Version	Runs	GreenHouse		FIRM-AFL		SNIPUZZ		BooFuzz		Bond
					Time	Factor	Time	Factor	Time	Factor	Time	Factor	Time
PSV-2020-0259	Netgear	R8500	1.0.2.160	10/10	-	-	x	x	-	-	-	-	1h49m53s
PSV-2021-0342	Netgear	R7000P	1.3.0.8	10/10	-	-	x	x	-	-	-	-	1m58s
PSV-2022-0201	Netgear	R7000P	1.3.0.8	10/10	-	-	x	x	-	-	-	-	4m11s
CVE-2022-44193	Netgear	R7000P	1.3.0.8	10/10	-	-	x	x	-	-	-	-	1m6s
CNVD-2022-70682	Netgear	R7000P	1.3.0.8	10/10	-	-	x	x	-	-	1m9s	1.13	1m1s
PSV-2020-0310	Netgear	R7000P	1.3.1.64	10/10	x	x	x	x	-	-	-	-	1m19s
PSV-2020-0311	Netgear	R7000P	1.3.1.64	10/10	x	x	x	x	-	-	-	-	1m20s
PSV-2020-0312	Netgear	R7000P	1.3.1.64	10/10	x	x	x	x	-	-	-	-	1m20s
PSV-2020-0314	Netgear	R7000P	1.3.1.64	10/10	x	x	x	x	-	-	-	-	1m36s
PSV-2022-0004	Netgear	XR300	1.0.3.72	10/10	-	-	x	x	-	-	-	-	2m12s
PSV-2022-0140	Netgear	XR300	1.0.3.72	10/10	-	-	x	x	-	-	-	-	1m21s
PSV-2022-0171	Netgear	XR300	1.0.3.72	10/10	-	-	x	x	-	-	-	-	1m58s
CVE-2025-4139	Netgear	EX6120	1.0.0.68	10/10	-	-	x	x	-	-	-	-	43s
CVE-2024-22544	Linksys	E1700	1.0.0.4.003	10/10	x	x	x	x	-	-	-	-	59s
CVE-2020-35713	Linksys	RE6500	1.0.10.001	10/10	x	x	x	x	-	-			39s
CVE-2018-17065	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	3m51s
CVE-2018-17068	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	35s
CVE-2022-29321	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	30s
CVE-2022-37123	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	-
CVE-2022-37134	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	38s
CVE-2022-37125	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	39s
CVE-2022-43003	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	1m16s
CVE-2023-43237	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	37s
CVE-2023-43240	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	39s
CVE-2024-24321	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	40s
CVE-2025-29743	D-Link	DIR816	1.10CNB03	10/10	x	x	x	x	-	-	-	-	-
CVE-2023-43862	D-Link	DIR619L	2.06B01	10/10	x	x	x	x	-	-	-	-	1m6s
CVE-2024-9908	D-Link	DIR619L	2.06B01	10/10	x	x	x	x	-	-	20s	0.29	1m10s
CVE-2024-9909	D-Link	DIR619L	2.06B01	10/10	x	x	x	x	-	-	20s	0.26	1m17s
CVE-2024-9566	D-Link	DIR619L	2.06B01	10/10	x	x	x	x	-	-	27s	0.39	1m10s
CVE-2024-33774	D-Link	DIR619L	2.06B01	10/10	x	x	x	x	-	-	-	-	1m12s
CVE-2025-4448	D-Link	DIR619L	2.06B01	10/10	x	x	x	x	-	-	-	-	-
CVE-2025-4451	D-Link	DIR619L	2.06B01	10/10	x	x	x	x	-	-	-	-	1m10s
CVE-2016-11021	D-Link	DCS932L	1.00	10/10	15h45m49s	1830.61	x	x	-	-	16s	0.52	31s
CVE-2019-10999	D-Link	DCS932L	1.00	10/10	15h45m49s	1719.67	x	x	-	-	-	-	33s
CVE-2022-40110	TOTOLink	A3002R	V1.1.1	10/10	x	x	x	x	-	-	23s	0.51	45s
CVE-2025-25610	TOTOLink	A3002R	V1.1.1	10/10	x	x	x	x	-	-	-	-	42s
CVE-2025-45861	TOTOLink	A3002R	V1.1.1	10/10	x	x	x	x	-	-	-	-	1m1s
CVE-2025-45865	TOTOLink	A3002R	V1.1.1	10/10	x	x	x	x	-	-	-	-	1m
CVE-2023-24144	TOTOLink	CA300-POE	V6.2c.884	10/10	-	-	x	x	-	-	-	-	3m6s
CVE-2023-24148	TOTOLink	CA300-POE	V6.2c.884	10/10	-	-	x	x	-	-	-	-	17s
CVE-2022-24145	Tenda	AX3	V16.03.12.10	10/10	x	x	x	x	-	-	-	-	-
CVE-2022-24146	Tenda	AX3	V16.03.12.10	10/10	x	x	x	x	-	-	13s	0.48	27s
CVE-2022-24149	Tenda	AX3	V16.03.12.10	10/10	x	x	x	x	-	-	-	-	29s
CVE-2022-24157	Tenda	AX3	V16.03.12.10	10/10	x	x	x	x	-	-	-	-	34s
CVE-2022-24158	Tenda	AX3	V16.03.12.10	10/10	x	x	x	x	-	-	-	-	36s
CVE-2022-24159	Tenda	AX3	V16.03.12.10	10/10	x	x	x	x	-	-	-	-	33s
CVE-2022-24162	Tenda	AX3	V16.03.12.10	10/10	x	x	x	x	-	-	15s	0.6	25s
CVE-2022-24163	Tenda	AX3	V16.03.12.10	10/10	x	x	x	x	-	-	-	-	34s
CVE-2024-7151	Tenda	O3V2	1.0.0.12	10/10	x	x	x	x	-	-	22s	0.65	34s
CVE-2024-7152	Tenda	O3V2	1.0.0.12	10/10	x	x	x	x	-	-	-	-	34s
CVE-2024-34338	Tenda	O3V2	1.0.0.12	10/10	x	x	x	x	-	-	-	-	39s
CVE-2022-30024	TP-Link	TL-WR841N	V11_160325	10/10	x	x	-	-	-	-	-	-	1m56s
CVE-2025-25898	TP-Link	TL-WR841N	V11_160325	10/10	x	x	-	-	-	-	-	-	2h36m10s
ZDI-15-348	Belkin	F9K1122	1.00.23	10/10	x	x	x	x	-	-	-	-	58s
ZDI-15-350	Belkin	F9K1122	1.00.23	10/10	x	x	x	x	-	-	-	-	1m10s
ZDI-15-351	Belkin	F9K1122	1.00.23	10/10	x	x	x	x	-	-	-	-	1m8s
CVE-2023-31528	Motorola	CX2L	1.0.1	10/10	x	x	x	x	-	-	-	-	-
CVE-2023-31529	Motorola	CX2L	1.0.1	10/10	x	x	x	x	-	-	14s	0.47	30s
CVE-2024-45880	Motorola	CX2L	1.0.1	10/10	x	x	x	x	-	-	-	-	37s
60	\	\	\	\	15h45m49s	160.76	-	-	-	-	25s	0.07	5m53s

Table 9: Results of component-wise ablation study. The symbol - denotes that this configuration did not successfully trigger any vulnerabilities, and thus no average validation time can be computed.

ID	Vendor	Product	Vulns	Bond_n				Bond_e				Bond_{e+c}			Bond		
				Entry Points	Vulns	Recall	Avg.TTE/s	Entry Points	Vulns	Recall	Avg.TTE/s	Vulns	Recall	Avg.TTE/s	Vulns	Recall	Avg.TTE/s
1	Netgear	R8500	2	2	0	0%	-	2	0	0%	-	1	50%	64	1	50%	64
2	Netgear	R7000P	8	1	1	12.5%	17	8	3	37.5%	27.7	7	87.5%	82.9	8	100%	103.9
3	Netgear	XR300	4	2	0	0%	-	4	0	0%	-	4	100%	100	4	100%	100
4	Netgear	WNDR4500	3	2	0	0%	-	3	1	33.3%	27	2	66.7%	57	3	100%	62.3
5	Netgear	EX6120	1	0	0	0%	-	1	0	0%	-	1	100%	43	1	100%	43
7	Linksys	E1700	3	0	0	0%	-	3	0	0%	-	2	66.7%	825.5	3	100%	55.7
8	Linksys	RE6500	3	3	1	33.3%	13	3	1	33.3%	15	3	100%	41	3	100%	41
9	D-Link	DIR816	16	6	0	0%	-	16	2	12.5%	10.5	12	75%	371.3	14	87.5%	56.9
10	D-Link	DIR619L	19	17	4	21.1%	17.8	19	6	31.6%	18	16	84.2%	342.6	18	94.7%	81.7
12	D-Link	DCS932L	2	1	1	50%	10	2	2	100%	12	2	100%	32	2	100%	32
13	TOTOLink	A3002R	8	7	1	12.5%	12	8	1	12.5%	13	5	62.5%	650.4	8	100%	88
14	TOTOLink	CA300-PoE	4	3	1	25%	2	4	1	25%	3	4	100%	110	4	100%	61.5
15	Tenda	AX3	8	8	2	25%	8.5	8	2	25%	9.5	7	87.5%	31.1	7	87.5%	31.1
16	Tenda	O3V2	6	4	1	16.7%	7	6	1	16.7%	8	6	100%	35.1	6	100%	37
17	TP-Link	TL-WR841N	2	2	0	0%	-	2	0	0%	-	1	50%	116	1	50%	116
18	Belkin	F9K1122	7	2	0	0%	-	7	0	0%	-	6	85.7%	255.8	7	100%	68.1
19	Motorola	CX2L	4	4	3	75%	14	4	3	75%	15.7	3	75%	36	3	75%	36
Total	\	\	**100**	**64**	**15**	**15%**	**12.7**	**100**	**23**	**23%**	**16**	**82**	**82%**	**230.4**	**93**	**93%**	**67.5**

entry point and parameter format, not to mention simultaneously satisfying multiple parameter constraints. Despite applying coverage guidance, exploration remained focused on the HTTP protocol parser, failing to effectively penetrate functional code segments. As a result, critical constraints are rarely satisfied and directed validation is infeasible. Moreover, GreenHouse failed to emulate the other 11 firmware images, underscoring the challenges of IoT device emulation. In contrast, Bond avoids emulation by applying lightweight static analysis on binaries for vulnerability validation.

FIRM-AFL similarly managed to emulate only one device and found no vulnerabilities, with failures resembling those of GreenHouse.

SNIPUZZ did not discover any vulnerabilities in our dataset. Its response-guided field probing was time-consuming, which conflicts with the limited budget for validating taint analysis results. Byte-by-byte string inference generates redundant data, and subsequent Havoc-based fragment mutation is blind and inefficient, failing to generate constrained inputs for effective validation. In contrast, Bond analyzes the back-end to extract entry points, parameters, and path constraints, avoiding redundant mutations.

BooFuzz performed best among the baselines, validating 10 vulnerabilities in an average of 25 seconds. However, these vulnerabilities were shallow cases that required no path constraints and could be triggered simply by constructing protocol-formatted requests. Compared to Bond, which validated a total of 55 vulnerabilities, including some that were difficult to trigger, static analysis resulted in increased time overhead, but it was still acceptable.

4.4 Ablation Studies-RQ3

This section conducts two ablation studies to evaluate the contributions of Bond's components. The *Component-Wise Ablation* progressively integrates three modules—entry point identification, constraint analysis, and mutation prioritization—to measure their performance impact. The *Constraint-Type Ablation* focuses on the constraint analysis module, where we selectively disable individual constraint types to assess their effect on vulnerability validation. We conducted these experiments on 100 vulnerabilities: the 40 randomly selected 0-day cases (§ 4.2) and 60 known 1-day cases (§ 4.3). Each validation task had a one-hour time limit.

4.4.1 Component-Wise Ablation

We start with the naive version of Bond (all core techniques disabled) and progressively add each component to evaluate its individual contributions to effectiveness and efficiency. Specifically, we compare the following four configurations:

- Bond_n: Naive variant using Selenium [65] and BrowserMob Proxy [66] to capture HTTP requests from the web interface during configuration; the extracted entry points and parameters are used for evaluation.
- Bond_e: Includes entry point identification module, using random mutation without any prioritization strategy.
- Bond_{e+c}: Includes entry point identification and constraint analysis modules, still without applying prioritization.
- Bond: Full version of Bond, combining entry point identification, constraint analysis, and mutation prioritization.

Table 9 summarizes the results of our component-wise ablation study. As shown, the entry point identification and constraint analysis modules significantly improve validation recall, contributing 53% and 257% increases respectively. In addition, the mutation prioritization strategy notably enhances validation efficiency, achieving a 3.41× speedup.

The entry point identification module substantially expands the attack surface. It increases the number of recognized entry points from 64 to 100, making 36 previously unreachable vulnerabilities potentially validatable. Notably, entry point analysis is performed only once and reused across all vulnerability validation tasks for the same device, incurring only 3.3s of additional average overhead per task.

The constraint analysis module delivers the most substantial gains. Since IoT fuzzing is slow (1 request/sec), pruning infeasible inputs early is critical. Constraint analysis introduces an average static analysis overhead of 214s per test—approximately 13× that of Bond_e. However, this cost is justified: it increases the number of validated vulnerabilities from 23 to 82, achieving a 3.57× improvement. In Bond_e, only simple cases without complex branches are triggered. The remaining 77 require satisfying input-dependent constraints that random mutation rarely satisfies within time limits. Constraint analysis helps prune infeasible paths and guides the fuzzer toward inputs that meet critical branch conditions, enabling the validation of these hard-t o-reach vulnerabilities.

The mutation prioritization strategy reduces the time to trigger vulnerabilities by modeling mandatory, partial, and unconstrained fields with a staged prioritization scheme. By prioritizing mutations on key input fields, the fuzzer can more quickly reach target paths while avoiding unnecessary energy expenditure on irrelevant fields. Furthermore, this strategy also raises validated vulnerabilities from 82 in Bond_{e+c} to 93 in the full version of Bond.

We also observed a few exceptions. On devices with IDs 4 and 16, Bond_{e+c} outperformed the full Bond in validation speed. This occurred because, in the absence of a prioritization strategy, Bond_{e+c} selects input fields for mutation at random, and some critical fields were coincidentally chosen early, resulting in faster vulnerability validation. Despite such cases, the mutation prioritization strategy still achieves consistent and significant overall effectiveness.

4.4.2 Constraint-Type Ablation

Fig. 5 shows the constraint-type ablation results. With no constraints, only 23 alerts were validated, while enabling all yields 93. String matching and numeric parsing are most critical: disabling either drops validation to 60 and 71, indicating reliance on precise tokens and numeric ranges. Byte-level, null value and network format each add about 10 alerts, typically correspond to memory-level comparisons, input presence validations or format validations. Heuristic semantic constraints contribute only 3 alerts but provide a lightweight supplement that can be extended with expert knowledge.

Figure 5: Results of constraint-type ablation study.

5 Discussion

Required Target Knowledge. Bond requires two essential forms of prior knowledge: the target device binaries and taint analysis reports. The binary is used in Bond's static analysis phase, while the taint analysis result identifies vulnerable parameters and sinks. Using this information, Bond extracts precise entry points and constraints. Bond's validation capability depends on the quality of the taint report. When the report is incomplete or noisy, Bond may explore irrelevant execution paths. Users can mitigate this issue by leveraging domain expertise or performing manual inspection to specify the intended sources and sinks. Without such knowledge, Bond becomes nearly unusable, as most target devices contain a large number of potential entry points and input parameters, making exhaustive fuzzing computationally prohibitive and often less efficient than manual auditing.

Application Scope. Bond currently focuses on back-end components implemented in C/C++ code within Linux-based firmware, particularly those exposed through HTTP or other web interfaces, where more than 70% of vulnerabilities occur [67]. Furthermore, Bond is currently unable to handle cross-process data flows, as validating cross-process taint vulnerabilities requires constructing accurate inter-process control-dependency relationships to enable reliable constraint tracking [68]. At the same time, many daemon and child-process bugs remain silent in black-box settings [69], making such validation inherently difficult.

Protocol Extension. IoT devices feature alternative attack surfaces, such as proprietary protocols or services, which can limit Bond's applicability. Extending Bond to other protocols (text, semi-text, or binary) mainly requires prompt adaptation and engineering effort: prompts must be redesigned to interpret diverse data formats and encodings. Unlike HTTP's well-defined textual syntax, many IoT protocols use compact binary formats (e.g., MQTT) or vendor-specific markup (e.g., XML-based UPnP), introducing substantial variability that complicates fuzzing template generation. Binary protocols pose even greater challenges due to the absence of textual structure and semantic clarity.

Assumptions on Binary Analyzability. Bond assumes access to both dynamically linked and statically linked binaries that contain library function symbols, enabling accurate static analysis. Since dynamically linked binaries are prevalent in Linux-based firmware due to storage constraints, this assumption is generally reasonable [70]. For stripped, statically linked binaries, existing code similarity analysis techniques have been shown effective in recovering standard library function symbols [71–73], suggesting that integrating these methods could further extend the applicability of Bond.

6 Related Work

IoT Firmware Vulnerability Detection. IoT firmware vulnerability detection largely splits into static taint analysis and fuzzing. Static taint analysis [1–10] traces sources to sinks via reaching definitions and symbolic execution; SaTC [3] exploits shared front/back-end keywords to recover intermediate taint sources [10], EmTaint [6] introduces SSE-based expressions to recover indirect calls, Mango [4] adopts backward analysis for scalability, and OctopusTaint [5] employs context-sensitive reaching definitions—yet they often raise many alerts with high FPs, requiring costly manual validation. Emulation-driven fuzzing spans full-system (Firmadyne [35]), arbitration (FirmAE [37]), user-mode (FIRM-AFL [36], EQUAFL [38]), and lightweight rehosting (GreenHouse [39], HouseFuzz [40]), but commonly faces low success, heavy setup, limited fidelity, and generic coverage without source-to-sink direction. Black-box fuzzers avoid emulation: SNIPUZZ [22] guides by response differences, and Labrador [23] correlates response strings with back-end basic blocks to estimate target distance and guide fuzzing; both provide coarse guidance and cannot validate taint analysis reports. Bond bridges static detection and automated validation without emulation, relying on lightweight static analysis of back-end binaries; by combining path constraints with control-flow context, it steers fuzzing to truly reachable paths and accurately validates taint analysis reports.

PoC Generation. Directed fuzzing and directed symbolic execution are mainstream approaches for PoC generation. Directed fuzzing has been widely applied to open-source software and operating system kernels, with guidance-based methods estimating distances to target basic blocks [28–30, 74] and restriction-based methods instrumenting only dependency-related blocks [31–34]. Recently, progress has also been made in the web domain: Predator [75] applies selective dynamic instrumentation, and WDFUZZ [76] implements a hierarchical scheduling algorithm to efficiently trigger vulnerabilities; these systems typically target PHP/Java and rely on instrumentation-derived feedback. Overall, such approaches often require source code or high-fidelity environments, limiting applicability to IoT firmware. Directed symbolic execution is similar in spirit—steering exploration toward targets via distance heuristics over control/data-flow graphs or pruning unrelated paths [77–79]. In contrast, Bond requires no access to source code: via fine-grained constraint analysis it prunes unreachable inputs from the outset, avoiding fruitless program-space exploration and ultimately yielding a highly efficient and precise scheme tailored for the constrained black-box IoT setting.

7 Conclusion

We presented Bond, a constraint-directed black-box fuzzing framework that bridges static taint analysis and dynamic vulnerability validation for IoT firmware. By leveraging constraint-guided input mutation and systematic constraint collection, Bond validated large numbers of taint analysis reports with high accuracy and efficiency to generate PoCs for actionable vulnerabilities. Our evaluation on 19 real-world devices demonstrated that Bond validated 1,349 true vulnerabilities, including 155 previously unknown cases. On 60 known vulnerabilities, Bond achieved a 91.67% recall rate and significantly outperformed SOTA IoT fuzzers in both vulnerability discovery and testing efficiency. Overall, Bond advances the reliability and practicality of firmware vulnerability analysis by addressing the long-standing challenge of validating static taint analysis results.

Acknowledgments

We are grateful to our shepherd and the anonymous reviewers for their valuable guidance and insightful comments. We also gratefully acknowledge Dr. Yicheng Zeng from our institution for his assistance in manually validating a large number of vulnerability alerts during the validation process. This research was supported by the National Key Research and Development Program of China (No. 2022YFB3103904), the Ant Group Postdoctoral Program, and the China Postdoctoral Science Foundation (No. 2025M771528).

Ethical Considerations

In conducting this research, we carefully considered its ethical implications and took multiple measures to ensure that our findings were handled responsibly and in accordance with ethical best practices. We remained mindful of the potential risks inherent in security research and exercised due caution throughout the research process, adhering to widely accepted norms in the systems security research community.

Vulnerability Discovery and Disclosure. For vulnerabilities identified by Bond, we follow the disclosure procedure in § 4.2.2 to notify vendors and organizations, giving them sufficient time to patch the issues before any public disclosure. This process ensures that our research contributes positively to security without exposing users to unnecessary risks.

Experiments with Live Systems. All experiments were conducted on physical devices in a strictly controlled, isolated intranet environment, without any connection to the public internet, to ensure that our testing did not pose unintended risks to users or systems. The dataset used in this research is constructed from firmware that vendors publicly provide for download and testing on the internet.

Stakeholders and Potential Impact. The stakeholders of this study include IoT end users and device vendors, security researchers and analysts, and the broader Internet ecosystem and society. ① Bond is designed for alert validation and PoC

generation. Some devices affected by confirmed vulnerabilities may have already reached EOL and therefore cannot be patched. If information about such vulnerabilities were leveraged by attackers, it could have direct adverse impacts on end users. From the vendors' perspective, our work assists in identifying vulnerabilities that are real and practically triggerable, thereby accelerating patch deployment; however, it may also increase short-term remediation pressure and expose legacy security issues. These considerations motivate us to adopt responsible disclosure and delayed release practices, with the goal of protecting both users and vendors to the greatest extent possible. We also strongly recommend that vendors address high-severity taint vulnerabilities in EOL devices whenever feasible, in order to better protect end users. ② Bond represents a reusable system that can be replicated, extended, and integrated into future research and tools. However, such techniques may also be misused to guide exploit development or accelerate vulnerability weaponization. We therefore emphasize responsible use and explicitly oppose unethical applications, while maintaining that open and reproducible dissemination is essential for the cumulative progress of security research. If security researchers use this tool to validate vulnerabilities on new devices and discover previously unknown vulnerabilities, we strongly encourage them to promptly contact the affected vendors and to refrain from prematurely releasing PoCs that could be exploited by attackers. ③ Bond has clear public safety value, as it enables faster remediation and more accurate vulnerability classification. Although misuse could potentially lead to targeted attacks against specific devices, large-scale IoT vulnerabilities constitute a systemic risk that cannot be effectively mitigated without improving vulnerability discovery and validation practices. Addressing this risk requires timely, evidence-based remediation, which is precisely the goal of our work.

Decision to Publish. The decision to proceed with the publication of this work was made after carefully weighing its potential benefits against possible ethical risks. As described above, we designed our experiments and disclosure process to minimize harm, including controlled testing environments, responsible disclosure, and delayed public release for vulnerable devices. While we acknowledge that vulnerability-related research may carry a residual risk of misuse, we consider this risk to be limited and acceptable given the anticipated security benefits. In particular, improving the accuracy and efficiency of validating taint analysis reports can reduce false positives, prioritize real vulnerabilities, and support more timely remediation efforts. We therefore believe that making this work publicly available, under the safeguards described above, is justified and serves the broader public interest.

Open Science

We provide all artifacts for evaluation, including datasets, Bond prototype code, and runtime environment configuration, available at `https://doi.org/10.5281/zenodo.17921159`

- **Dataset:**
  - 19 firmware images from 8 vendors, covering all versions listed in Table 2.
  - 60 known vulnerabilities collected for comparison with existing IoT fuzzing tools.
- **Fully Open-sourced Bond Prototype:**
  - The IDA scripts for the preprocessing module.
  - Code for reachability analysis, constraint analysis and directed black-box fuzzing.

References

[1] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. DTaint: detecting the taint-style vulnerability in embedded device firmware. In *DSN*, 2018.

[2] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *SP*, 2020.

[3] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *USENIX Security*, 2021.

[4] Wil Gibbs, Arvind S Raj, Jayakrishna Menon Vadayath, Hui Jun Tay, Justin Miller, Akshay Ajayan, Zion Leonahenahe Basque, Audrey Dutcher, Fangzhou Dong, Xavier Maso, et al. Operation Mango: Scalable discovery of taint-style vulnerabilities in binary firmware services. In *USENIX Security*, 2024.

[5] Abdullah Qasem, Mourad Debbabi, and Andrei Soeanu. OctopusTaint: Advanced data flow analysis for detecting taint-based vulnerabilities in iot/iiot firmware. In *CCS*, 2024.

[6] Kai Cheng, Yaowen Zheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, Kejiang Ye, and Limin Sun. Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis. In *ISSTA*, 2023.

[7] Zicong Gao, Chao Zhang, Hangtian Liu, Wenhou Sun, Zhizhuo Tang, Liehui Jiang, Jianjun Chen, and Yong Xie. Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis. In *NDSS*, 2024.

[8] Xiaokang Yin, Ruijie Cai, Xiaoya Zhu, Qichao Yang, Enzhou Song, and Shengli Liu. Precise discovery of more taint-style vulnerabilities in embedded firmware. *TDSC*, 2024.

[9] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, and Limin Sun. LLM-powered static binary taint analysis. *TOSEM*, 2025.

[10] Puzhuo Liu, Yaowen Zheng, Chengnian Sun, Chuan Qin, Dongliang Fang, Mingdong Liu, and Limin Sun. FITS: Inferring Intermediate Taint Sources for Effective Vulnerability Analysis of IoT Device Firmware. In *ASPLOS*, 2023.

[11] Abdullah Qasem, Paria Shirani, Mourad Debbabi, Lingyu Wang, Bernard Lebel, and Basile L Agba. Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies. *ACM Computing Surveys (CSUR)*, 54(2):1–42, 2021.

[12] Xiaotao Feng, Xiaogang Zhu, Qing-Long Han, Wei Zhou, Sheng Wen, and Yang Xiang. Detecting vulnerability on iot device firmware: A survey. *IEEE/CAA Journal of Automatica Sinica*, 10(1):25–41, 2022.

[13] Statista. Internet of things (iot) - statistics and facts. https://www.statista.com/topics/2637/internet-of-things/#topicOverview, 2024.

[14] HiveMQ Team. The coolest smart cities of 2025: How iot is changing urban living in america. https://www.hivemq.com/blog/the-coolest-smart-cities-2025-how-iot-changing-urban-living-america/, 2025.

[15] HiveMQ Team. Expanding industrial iot in 2025: Survey reveals significant and sustained growth. https://www.hivemq.com/blog/expanding-industrial-iot-in-2025-survey-reveals-growth/, 2025.

[16] Statista. Number of users of the smart home segment smart appliances in the united states, 2019–2028. https://www.statista.com/forecasts/887601/number-of-smart-homes-in-the-smart-home-segment-smart-appliances-in-the-united-states, 2025.

[17] Taimur Bakhshi, Bogdan Ghita, and Ievgeniia Kuzminykh. A review of iot firmware vulnerabilities and auditing techniques. *Sensors*, 24(2):708, 2024.

[18] Joseph R Ruthruff, John Penix, J David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE*, 2008.

[19] Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, Brendan Dolan-Gavitt, et al. Arvo: Atlas of reproducible vulnerabilities for open source software. *arXiv preprint arXiv:2408.02153*, 2024.

[20] Peiran Wang, Xiaogeng Liu, and Chaowei Xiao. Cve-bench: Benchmarking llm-based software engineering agent's ability to repair real-world cve vulnerabilities. In *NAACL*, 2025.

[21] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, et al. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *USENIX Security*, 2022.

[22] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *CCS*, 2021.

[23] Hangtian Liu, Shuitao Gan, Chao Zhang, Zicong Gao, Hongqi Zhang, Xiangzhi Wang, and Guangming Gao. Labrador: Response guided directed fuzzing for black-box iot devices. In *SP*, 2024.

[24] Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe. Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems. *IoT-J*, 2021.

[25] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of embedded systems: A survey. *ACM Computing Surveys*, 55(7):1–33, 2022.

[26] Lingyun Situ, Chi Zhang, Le Guan, Zhiqiang Zuo, Linzhang Wang, Xuandong Li, Peng Liu, and Jin Shi. Physical devices-agnostic hybrid fuzzing of iot firmware. *IoT-J*, 2023.

[27] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity*, 2022.

[28] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *CCS*, 2017.

[29] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *USENIX security*, 2021.

[30] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *ICSE*, 2022.

[31] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *SP*, 2022.

[32] Changhua Luo, Wei Meng, and Penghui Li. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *SP*, 2023.

[33] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference. In *SP*, 2024.

[34] Penghui Li, Wei Meng, and Chao Zhang. SDFuzz: Target states driven directed fuzzing. In *USENIX Security*, 2024.

[35] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[36] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security*, 2019.

[37] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. FirmAE: Towards large-scale emulation of iot firmware for dynamic analysis. In *ACSAC*, 2020.

[38] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation. In *ISSTA*, 2022.

[39] Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, et al. Greenhouse: Single-service rehosting of linux-based firmware binaries in user-space emulation. In *USENIX Security*, 2023.

[40] Haoyu Xiao, Ziqi Wei, Jiarun Dai, Bowen Li, Yuan Zhang, and Min Yang. Housefuzz: Service-aware grey-box fuzzing for vulnerability detection in linux-based firmware. In *SP*, 2025.

[41] Yuhao Wu, Jinwen Wang, Yujie Wang, Shixuan Zhai, Zihan Li, Yi He, Kun Sun, Qi Li, and Ning Zhang. Your firmware has arrived: A study of firmware update vulnerabilities. In *USENIX Security*, 2024.

[42] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In *USENIX Security*, 2022.

[43] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. Afgen: Whole-function fuzzing for applications and libraries. In *SP*, 2024.

[44] Bo Yu, Ying Zhang, Yongyi Zhang, and Qiang Yang. Firm vea: vulnerability discovery optimisation for iot firmware via version evolution analysis. In *GLOBECOM*, 2023.

[45] Anis Lounis, Anthony Andreoli, Mourad Debbabi, and Aiman Hanna. Seum spread: Discerning security flaws in iot firmware via call sequence semantics. In *DIMVA*, 2024.

[46] Jiaxu Zhao, Yuekang Li, Yanyan Zou, Zhaohui Liang, Yang Xiao, Yeting Li, Bingwei Peng, Nanyu Zhong, Xinyi Wang, Wei Wang, et al. Leveraging semantic relations in code and data to enhance taint analysis of embedded systems. In *USENIX Security*, 2024.

[47] Xiaokang Yin, Ruijie Cai, Yizheng Zhang, Lukai Li, Qichao Yang, and Shengli Liu. Accelerating command injection vulnerability discovery in embedded firmware with static backtracking analysis. In *Proceedings of the 12th International Conference on the Internet of Things*, pages 65–72, 2022.

[48] BooFuzz. https://github.com/jtpereyda/boofuzz, 2017.

[49] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[50] Puzhuo Liu, Yaowen Zheng, Chengnian Sun, Hong Li, Zhi Li, and Limin Sun. Battling against protocol fuzzing: Protecting networked embedded devices from dynamic fuzzers. *TOSEM*, 2024.

[51] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. Pie: Parser identification in embedded systems. In *ACSAC*, 2015.

[52] National Vulnerability Database. https://nvd.nist.gov/, 2025.

[53] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[54] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Rfc2616: Hypertext transfer protocol–http/1.1, 1999.

[55] Haiyang Wei, Ligeng Chen, Zhengjie Du, Yuhan Wu, Haohui Huang, Yue Liu, Guang Cheng, Fengyuan Xu, Linzhang Wang, and Bing Mao. Unleashing the power of llm to infer state machine from the protocol implementation. *arXiv preprint arXiv:2405.00393*, 2024.

[56] Binwalk. https://github.com/ReFirmLabs/binwalk, 2024.

[57] IDA Pro. https://hex-rays.com/ida-pro, 2024.

[58] OpenAI. GPT-4o: Fast, intelligent, flexible gpt model. https://platform.openai.com/docs/models/gpt-4o, 2025.

[59] Junjian Ye, Xavier De Carné De Carnavalet, Lianying Zhao, Mengyuan Zhang, Lifa Wu, and Wei Zhang. Exposed by default: A security analysis of home router default settings. In *AsiaCCS*, 2024.

[60] TP-Link product security advisory. https://www.tp-link.com/us/press/security-advisory/, 2025.

[61] Common vulnerabilities and exposures numbering authorities. https://www.cve.org/programorganization/cnas, 2025.

[62] Coordinated vulnerability disclosure program: Linksys. https://www.linksys.com/pages/security-form.html, 2025.

[63] Bug bounty program: Netgear cash rewards engagement. https://bugcrowd.com/engagements/netgear, 2025.

[64] VulDB. https://vuldb.com/, 2025.

[65] Selenium. https://github.com/SeleniumHQ/selenium, 2025.

[66] Browsermob proxy. https://github.com/lightbody/browsermob-proxy, 2025.

[67] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *AsiaCCS*, 2016.

[68] Jiaqian Peng, Puzhuo Liu, Yicheng Zeng, Kai Cheng, Yongji Liu, Yun Yang, and Hongsong Zhu. Bridge: High-order taint vulnerabilities detection in linux-based iot firmware. In *SP*, 2026.

[69] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[70] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. One size does not fit all: security hardening of mips embedded systems via static binary debloating for shared libraries. In *ASPLOS*, 2022.

[71] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *CCS*, 2022.

[72] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *CCS*, 2018.

[73] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *USENIX Security*, 2017.

[74] Yujian Zhang, Yaokun Liu, Jinyu Xu, and Yanhao Wang. Predecessor-aware directed greybox fuzzing. In *SP*, 2024.

[75] Chenlin Wang, Wei Meng, Changhua Luo, and Penghui Li. Predator: Directed web application fuzzing for efficient vulnerability validation. In *SP*, 2025.

[76] Zihan Lin, Yuan Zhang, Jiarun Dai, Xinyou Huang, Bocheng Xiang, Guangliang Yang, Letian Yuan, Lei Zhang, Fengyu Liu, Tian Chen, et al. Effective directed fuzzing with hierarchical scheduling for web vulnerability detection. In *USENIX Security*, 2025.

[77] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *SAS*, 2011.

[78] Paul Dan Marinescu and Cristian Cadar. Katch: High-coverage testing of software patches. In *FSE*, 2013.

[79] Fengjuan Gao, Linzhang Wang, and Xuandong Li. Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities. In *ASE*, 2016.

A Case Study

As shown in Listing 1, there exists a buffer overflow introduced by the `curTime` parameter (line 5). In this case, the `sprintf` call in `formSetWAN` (line 8) depends on the constraint checks inside `PPPoE_Handler` (line 11). This function enforces multiple constraints on user-controlled parameters, including IP address format, reconnect mode, and MAC address. The `checkmacaddr` constraint is particularly complex, as it performs a loop with byte-by-byte comparison.

Manual analysis is challenging: it first requires reverse-engineering the entry point `formSetWAN` and identifying its related parameters, which involve 31 in total. Moreover, `PPPoE_Handler` exceeds 1,000 lines of code. Tracking how each parameter is processed is already difficult, let alone manually recovering the full set of constraints. Consequently, human-only constraint analysis is time-consuming and error-prone. Bond solves this problem by modeling semantic constraints: it abstracts `inet_aton` as an IPv4 format constraint, treats `mode_radio` as a byte-level constraint, and uses heuristic semantics to provide dictionary values for the `mac` keyword. Based on these models, Bond automatically constructs the feasible value space and prioritizes mutations on the parameters that directly affect the constraints. With this strategy, Bond validated the overflow vulnerability within just two minutes.

```c
// D-Link DIR619L
websFormDefine("formSetWAN", formSetWAN);
int formSetWAN(int a1) {
    char v0[200];
    char *v1 = webGetVar(a1, "curTime");
    // Constraint function: validates PPPoE settings
    if (PPPoE_Handler(a1))
        sprintf(v0, "/t.asp?t=%s", v1); // overflow
}

int PPPoE_Handler(int a1) {
    int result = 1;
    // ... (processing of the other 20 parameters
        omitted, ~700 code lines) ...
    // === Constraint 1: PPPoE IP address ===
    char *v0 = webGetVar(a1, "config_address");
    struct in_addr tmp;
    if (!inet_aton(v0, &tmp))
        result = 0; // invalid IP
    // ... (processing of the other 7 parameters
        omitted, ~100 code lines) ...
    // === Constraint 2: connect mode ===
    char v1 = *(char *)webGetVar(a1, "mode_radio");
    if (!(v1 == '0' || v1 == '1' || v1 == '2'))
        result = 0; // missing or unsupported mode
    // === Constraint 3: MAC addr ===
    char *v2 = webGetVar(a1, "macaddr");
    if (!checkmacaddr(v2))
        result = 0;
    return result;
}

// Byte-level Complex Constraints
int checkmacaddr(int a1) {
    const char *v0 = (const char *)a1;
    int v1 = strlen(v0);
    int v2 = strchr(v0, ':') ? 17 : 12;
    int v3 = 0;
    if ((!strchr(v0, ':') && v1 != 12) ||
        (strchr(v0, ':') && v1 != 17))
        return 0;
    do {
        if (*v0 == ':')
            ++v3, ++v0;
        if (!checkrange(v0[0]) || !checkrange(v0[1]))
            return 0;
        v0 += 2, v3 += 2;
    } while ( v3 < v2 );
    return 1;
}

int checkrange(char a1) {
    if ( (unsigned __int8)(a1 - '0') < 10
      || (unsigned __int8)(a1 - 'A') < 6
      || (unsigned __int8)(a1 - 'a') < 6 )
        return 1;
    return 0;
}
```

Listing 1: D-Link DIR619L Vulnerability.