# BPDA: Bidirectional Path and Data Flow Analysis-Driven Vulnerability Discovery in IoT Firmware

Anonymous Authors

***Abstract*—The rapid proliferation of Internet of Things (IoT) devices has revolutionized various sectors, from smart homes to industrial automation. However, this growth has also introduced significant security vulnerabilities. Unfortunately, current vulnerability detection methods suffer from inefficiencies and high false positive rates, making vulnerability analysis labor-intensive and time-consuming.**

**To alleviate above problems, we propose a bidirectional path and data flow analysis method to efficiently detect buffer overflow and command injection vulnerabilities in IoT devices. During the analysis of IoT vulnerabilities, we found that vulnerabilities arise in non-standard library sinks, and not all user inputs can reach each corresponding sink. Guided by these insights, we design a more comprehensive sinks identification algorithm and leverage backward data flow tracking to eliminate the non-vulnerable paths. After that, we execute forward taint analysis and generate final Proof of Concepts (PoCs). To evaluate the effectiveness of our method, we implement a prototype system, named BPDA. We evaluated it on 56 firmware samples from 7 major brands, comparing it with state-of-the-art methods (i.e., SaTC and Mango). During our evaluation, we discovered 163 real vulnerabilities, including 34 0-day vulnerabilities, of which 32 have been confirmed by CVE/CNVD. Besides, experiment results show that BPDA completed its analysis in just 6% of the time required by SaTC, and remarkably identified 21 vulnerabilities that SaTC and Mango had failed to detect. It also resolved the issue of Mango failing to analyze specific firmware. These results have demonstrated the superiority of BPDA in terms of effectiveness and efficiency in IoT vulnerability detection.**

## 1. Introduction

The proliferation of the Internet of Things (IoT) has indeed greatly facilitated people's lives, enabling various smart devices to connect seamlessly and provide a wide range of services. However, as these devices become more widespread, they also expand the potential attack surface for vulnerability exploitation, posing significant security threats. For example, Botnets such as Moobot [7], [22], Mirai [6], the Golang-based "AGoent" [7], and Gafgyt variants [39], relentlessly prey on IoT vulnerabilities (e.g., CVE-2022-26258 [16], CVE-2023-1389 [15], CVE-2024-7029 [14], etc.), consistently seeking to exploit them for launching Distributed Denial of Service (DDoS) attacks.

To uncover and repair vulnerabilities in IoT devices, researchers have proposed numerous vulnerability mining methods [9], [43]. Due to the wide variety of IoT device hardware, which encompasses different instruction set architectures and the difficulty in monitoring execution states in the real-world devices [34], researchers have attempted to conduct dynamic testing of IoT devices through a combination of Emulation and fuzz testing [19], [36], [38], [45]. Although some promising results have been achieved, the fidelity of simulated execution is not perfect [28], leading to less effective testing compared to traditional software. Furthermore, due to poor support for proprietary peripherals from specific vendors, there is currently a lack of universal emulation methods, making it difficult to perform full-system emulation for many devices [5], [8], [13], [27]. Therefore, for large-scale firmware analysis, researchers prefer methods that combine static taint analysis and symbolic execution, such as DTaint [11], Karonte [32], and so on. Such static analysis methods have discovered numerous IoT vulnerabilities.

However, the requirement for initially identifying taint sources and sinks, followed by analyzing the propagation of taint data from sources to sinks, often leads to issues such as low code coverage [17], high false positive rates [28] and limited efficiency. In response, researchers have proposed various solutions to enhance the effectiveness of taint analysis. For instance, to address the challenge of identifying taint sources, SaTC [10] determines taint sources by matching the shared strings between the front-end and back-end, with common functions like *strcpy*, *memcpy*, and *system* defaulting as sinks. It then leverages the Depth-First Search (DFS) algorithm to obtain the propagation paths from taint sources to sinks and subsequently performs taint propagation analysis based on symbolic execution. To solve the problem of incomplete data flow, EmTaint [12] employs structured symbolic expression-based (SSE-based) on-demand alias analysis to identify pointer aliases, thereby recovering indirect calls and improving the effectiveness of data flow analysis during taint propagation. To boost the efficiency of taint analysis, Mango [18] determines the sinks requiring taint propagation analysis by conducting backward data flow analysis starting from the sinks. Nevertheless, current methods rely on function names to identify sink points and restrict the scope of vulnerability sinks to the copy-related functions in standard libraries [42], thus ignoring user-defined copy functions, leading to an increase in false negative rates. Additionally, we discover that the

analysis efficiency can be further enhanced by filtering the taint analysis paths based on bidirectional path and data flow tracking.

To address above problems, we propose a novel firmware vulnerability detection method base on bidirectional path and data flow analysis, named BPDA. We observe that not all data reaching the dangerous functions (sinks) is tainted data (user input). Through bidirectional path and data flow analysis, these safe paths (sinks) can be eliminated, thereby improving the accuracy and efficiency of taint analysis. Therefore, BPDA first identifies potential vulnerability sources and comprehensive sinks, followed by performing forward analysis to traverse target vulnerability sinks and generate the potential paths leading to them. After establishing the paths to be analyzed, BPDA initiates backward traversal from the sinks through sensitive functions, filtering out the paths that are not actually tainted by determining whether the parameters are truly related to user input.

To demonstrate the effectiveness of BPDA, we evaluate it based on the dataset from SaTC (*SaTC Dataset*), the supplementary dataset based on SaTC (*BPDA Dataset*), and the dataset with vulnerabilities in user-defined loop functions (*Loop Dataset*). The experimental results indicate that, compared to SaTC, BPDA completes the analysis in just 6% of the time and discover 21 vulnerabilities that SaTC and Mango failed to detect. BPDA also address the failure problem for certain firmware in Mango. Besides, BPDA collectively identified 163 vulnerabilities and 34 in which were first discovered. In these zero-day vulnerabilities, 32 have been assigned CVE/CNVD numbers.

BPDA currently supports the detection of 9 common vulnerable functions distributed across memory overflow and command injection types. Furthermore, BPDA is capable of expanding the types of vulnerable functions supported for detection in subsequent usage.

***Contributions.*** In summary, our contributions are:

- We have enhanced the identification and localization of implicit taint sources, thereby improving the coverage of taint source detection.
- We proposed a novel method for identifying custom vulnerable functions by recognizing specific loop structure, which can detect more sink points than existing methods.
- We introduced a static vulnerability discovering approach combined with forward and backward analysis, which improves analysis efficiency over existing tools.
- We implemented a prototype system, BPDA, and evaluated it against the state-of-the-art solutions for finding taint-style vulnerabilities in firmware binaries.

## 2. Background

In this section, we illustrate our observation with a motivation example, the problems in current methods and a serial of challenges faced by BPDA.

### 2.1. Observation and Motivation Example

**2.1.1. Our Observation.** Current IoT vulnerability detection methods often generate a large number of analysis paths from source points to sink points. However, a significant portion of these taint paths are invalid. The analysis of invalid paths is often time-consuming and labor-intensive. Sometimes, due to the inherent incompleteness of forward taint analysis, the analysis of such paths will also lead to false positives [3]. Upon analyzing these invalid paths manually, we observed that numerous sensitive functions (sinks) are not vulnerable to the taint data from user input. Therefore, eliminating invalid paths in taint analysis remains a formidable task, which can improve analysis efficiency and reduce false positives. Based on our insight, we leverage the reverse (backward) data flow analysis, which can identify sinks that are unlikely to be vulnerable, thereby reducing invalid paths in forward taint analysis and enhancing the efficiency of the analysis process. Next, we will use a motivation example to illustrate our observation.

```
int __fastcall sub_41860C(int a1)
{
    int v2;
    int v3;
    int v4;
    char v6[64];

    memset(v6, 0, sizeof(v6));
    v3 = rand();
    v2 = time(0);
    snprintf(v6, 63, "%x:%x", v3, v2);
    v4 = sub_413E04(v6);
    strcpy(a1, v4);

    return _stack_chk_guard;
}
```

Listing 1: A dangerous function (sink) that is not affected by user input (taint data)

**2.1.2. Motivation Example.** As illustrated in Listing 1, within the identification outcome of `sub_41860C()` by the SaTC, the parameter *v4* is identified as the critical determinant for whether the program will transition into an insecure state with respect to the sensitive function `strcpy()`. However, an examination of the preceding instructions reveals that the value of *v4* is computed based on *v2* and *v3*, which are respectively derived from a randomly generated number and the current time stamp. Consequently, *v4* is not subject to direct manipulation by the user input. Therefore, it can be inferred that the identified vulnerability sink does not pose a substantial risk to the integrity of the program, and thus, it can be removed from the taint path to be analyzed.

If no further manual analysis of the results is performed, what methods can be used to further optimize the analysis? We draw on the bidirectional search approach used in manual vulnerability analysis and *propose a filtering method that combines forward and backward analysis of taint analysis paths.* To clearly illustrate our method, we present the analysis process of data flow through Figure 1.

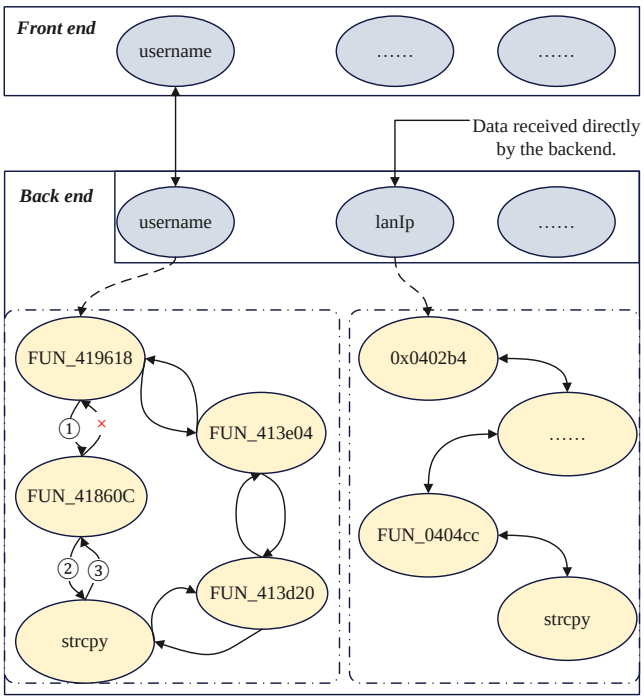


Figure 1: Illustration of the bidirectional analysis process.

As shown in Figure 1 (For clarity, we simplified it.), firstly, we identifies keywords in the front end files (It is the same as SaTC.). At this stage, we have expanded the sources of keywords to include locations where the back end directly acquires and processes data, thus incorporating these positions into the scope of taint sources. With `username` as the controllable vulnerability source in the front end, a preliminary path generation is conducted (Refer to steps 1 and 2 as illustrated in the figure). Upon completion, we obtained multiple function call paths with the risk of buffer overflow. Subsequently, we conduct backward analysis on the parameters of vulnerability sinks functions to determine whether they are related to variables in the function call chain or the vulnerability sources (Refer to steps 3 as illustrated in the figure). After filtering out functions with uncontrolled parameters (such as ones solely associated with static strings, this is reflected in stopping the backtracking and filtering corresponding paths after the analysis of the ending function `FUN_41860C`), we performed the forward taint analysis on remaining paths. Finally, we generated the Proof of Concept (PoC).

## 2.2. Problems in Current Approaches

Although automated IoT vulnerability detection methods have shown promising outcomes by discovering numerous 0-day vulnerabilities, they still face several problems that need to be tackled [37]. Specifically, some analysis techniques targeting specific applications can identify vulnerability, yet they lack the capability to determine the triggers for these crashes [24], [41], [44]; others, such as dynamic analysis, can trigger crashes, but due to limited semantic understanding, the specific inputs that lead to the crashes remain elusive [21], [26]. However, analyses that attempt both high reproducibility and high semantic understanding will face scalability issues [4], which imposes stringent requirements for maintaining equilibrium across the various dimensions of the technology [33].

In the previously state-of-the-art (SoTA) static taint analysis based methods, SaTC [10] proposed an efficient method for identifying vulnerability sources, utilizing the front-end of web services to locate the back-end code processing input data, and implemented a taint analysis engine strongly correlated with user input for input-sensitive taint analysis. Additionally, SaTC merged call traces sharing common paths. Mango [18] further optimized the vulnerability sources identification and propagation analysis on the foundation of SaTC, extending the target identification from boundary binary files to all binary files, and established MangoDFA for more granular value analysis. It filtered out some harmless paths by establishing rich expressions for variables. Following the completion of vulnerability sinks identification, Mango determined vulnerable data flows through iterative backward tracing from potential parent functions, which, compared to SaTC, reduced the false negative rate and improved analysis efficiency.

When tracing back from certain *sink* points involves a large number of branches, MangoDFA needs to maintain a substantial amount of state information. This can result in extremely high time and space costs on some firmware, thereby affecting the efficiency of analysis. In contrast, BPDA first generates path results from *src* to *sink* and performs backward analysis on every single path, only requiring minimal state information. Moreover, BPDA conducts backward analysis at the pseudo-code level, quickly eliminating irrelevant paths by assessing the relationship between function parameters and *sink* parameters. Furthermore, compared to SaTC and MangoDFA, BPDA add user-defined copy functions as *sink* points, further refining the taint information.

## 2.3. Challenges Faced by BPDA

We find that there remains potential for enhancement in both the efficiency and methodologies of analysis on the foundation of existing tools. In this section, we will discuss the challenges encountered during the implementation of BPDA and propose our solutions.

**2.3.1. Challenge 1: Custom copy function Identification.** It is apparent that the scope of support for identifying vulnerability sources and sinks fundamentally determine the analytical efficacy of a given method. Within firmware, there may exist a set of custom functions that are responsible for copying user-defined structures or data, and this copying process inherently carries the risk of memory security vulnerabilities. However, existing taint analysis tools have not yet integrated these functions into the defined range of vulnerability sinks.

The challenge in identifying custom copy functions is that they may not follow the typical patterns of standard library functions, with unique loop structures [35], memory access and parameter usage. Therefore, it is difficult to identify using traditional pattern matching or static analysis techniques. Additionally, function recognition in IoT needs address cross-architecture issues, as assembly language-based analysis methods lack generality. As a result, effectively identifying and locating these user-defined functions poses a significant challenge.

**2.3.2. Challenge 2: Efficient vulnerability analysis.** The path results generated from the vulnerability sources to the vulnerability sinks using traditional tools contain a significant portion of invalid paths, meaning that the vulnerability sinks are not actually influenced by user input.

Preliminary testing of SaTC revealed that some firmware paths reached the million level. Conducting taint analysis on such a large number of paths results in significant time consumption, severely reducing the efficiency of analysis. Similarly, the performance of Mango was not ideal for such firmware, as the large number of function branches led to system crashes. Therefore, proposing a more efficient and less resource-intensive taint analysis method is a significant challenge.

**2.3.3. Challenge 3: Backward data flow tracing.** In backward data flow tracing, how to identify and trace the key parameters of sensitive functions, and then check whether the key parameters are indeed affected by the user input stream, poses a challenging problem. In this process, analyzing each function along the path individually may incur high performance overhead [40]. Using traditional taint analysis methods for parameter tracking will reduce analysis efficiency. To achieve scalability in the analysis objects, a more effective recursive traversal method is needed.

## 3. A Comprehensive Look at Our Approach

In this paper, we introduce BPDA as a solution to aforementioned challenges, addressing the shortcomings of current IoT vulnerability detection methods. The overall of BPDA is illustrated in the Figure 2, detailing the specific workflow from firmware input to PoC generation.

BPDA primarily consists of three components: SrcFinder, LoopFinder and Bidirectional taint analysis. SrcFinder and LoopFinder are mainly used for the complete identification of *src* and *sink* points, while the bidirectional taint analysis is the core to enhance the capability and efficiency of vulnerability analysis.

Initially, BPDA unpack firmware through analysis tools (e.g., binwalk [25]). Subsequently, it employs SaTC to identify user input points through shared strings and uses the combination of code structure and functionality matching methods to locate potential input points (SrcFinder).

Subsequently, BPDA identifies *sink* points. BPDA not only recognizes commonly sensitive functions within libc

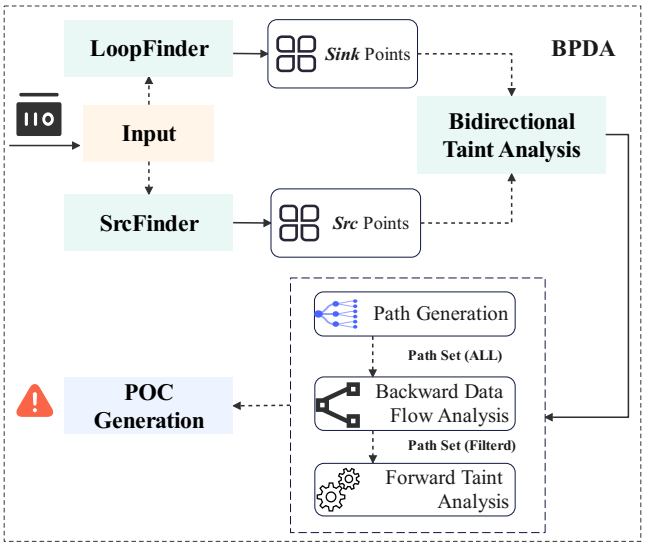


Figure 2: The comprehensive look at the approach of BPDA.

libraries, but also identifies user-customized copy-like functions (LoopFinder). By translating function code into the VEX IR intermediate language and conducting data flow analysis, it discerns functions with specific loop-based copying structure. Finally, it ascertains ultimate *sink* points by correlating function parameters with the loop structure.

Subsequently, the core component of BPDA is executed: the bidirectional taint analysis, which is divided into 3 steps. First, it generates call paths from *src* to *sink*. To ensure the integrity of the paths and enhance the efficiency of analysis, BPDA implements cross-boundary path generation and path integration. Then, BPDA conducts backward data flow tracking. Based on the decompiled pseudocode, it analyzes the parameter correlation between *sink* points and calling functions, and recursively conducts a reverse analysis along the function path. Based on the results, it eliminate call paths unrelated to the critical parameters at *sink* points. Afterwards, taint analysis from *src* to *sink* is performed.

Finally, combining the results of taint analysis and the data interaction methods of *src* points, a proof of concept (PoC) is automatically generated.

## 4. Taint Sources Identification (SrcFinder)

Current techniques typically rely on explicit identification information such as keywords shared between front-end and back-end, or predefined network encoding strings [30], to locate the positions where inputs are used in the back-end service program code. Subsequently, these positions are used as sources of taint for vulnerability detection based on taint analysis.

However, the SoTA front-end and back-end keyword identification tool, SaTC, focuses solely on Web APIs. Moreover, when processing keywords, SaTC can only identify and handle vulnerability sinks where keywords are explicitly shared between the front and back ends, leading

to a high false negative rate. Mango has built upon this foundation and enhanced the capability to identify keywords within ASP, PHP, and other file types. However, it still focuses on the combination of string literals and object fields, without addressing the previously mentioned issue of implicit keywords.

In terms of the communication method between the front-end and back-end, there are generally two approaches. (1) The back-end directly parses the request data. (2) The front-end first writes the request information into a configuration file or storage locations, which the back-end subsequently accesses to retrieve the relevant information. We found that the approach of obtaining information from the configuration file or other locations is extremely similar to the method of extracting payloads from malicious traffic packets used by Trojans. Both involve continuously reading relevant information and searching for whether the target features exist to complete the extraction of target data. Considering that the function of the front-end web page of a router is generally to facilitate user configuration, we further strengthened the code characteristics of transferring data into command execution functions.

In the implementation of BPDA, we address the previously mentioned challenges and provide characterization and definition for implicit taint sources. Implicit taint sources acquire and transmit data to be processed to the back-end, where there is no apparent correlation between the data acquisition location and the front-end. We utilize an approach reminiscent of payload detection strategies in conventional Trojan analysis to identify these implicit sources. This approach entails iteratively searching for data that fits specific target attributes. Once the target data is identified, it is subsequently processed and funneled into command execution functions, such as `system()`. The taint sources identification method (SrcFinder) is as shown in Figure 3.

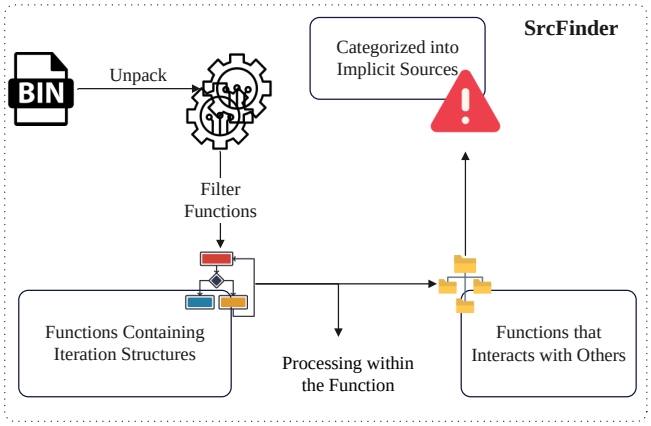


Figure 3: The structure of SrcFinder.

In summary, we have adopted the following methods to address the issue of identifying implicit taint sources:

1) Unpack the firmware and extract network-related binary files such as httpd, upnpd, etc., as the analysis targets.
2) Analyze the binary with IDAPython [20], traverse the binary and establish a function dependency graph. Check for any code structures that analyze data using traversal loops. Proceed further verification of the function only if it meets the condition.
3) Analyze the destination of the data results identified in the second step of traversal functions. If the data is processed within the function, marking is not required; if the identified results are passed to other functions (e.g., `system()`), they are considered as potential taint sources.

## 5. Sensitive Sinks Identification (LoopFinder)

Traditional taint analysis techniques typically consider sensitive functions in standard library as sink points. Sfuzz [23] identifies copy functions in standard library by virtual execution. Such methods are resource-intensive and lack identification of custom copy functions, increasing the false negative rate. In comparison, LoopFinder identifies copy functions based on the code structure and data flow characteristics.

Based on empirical analysis of firmware with different architectures, we identify that custom copy functions must exhibit function block loops and memory access, which in assembly code manifest as opcodes for memory load and store operations [42]. As shown in the Figure 4, the `strncpy` function defined under the ARM architecture meets these features.

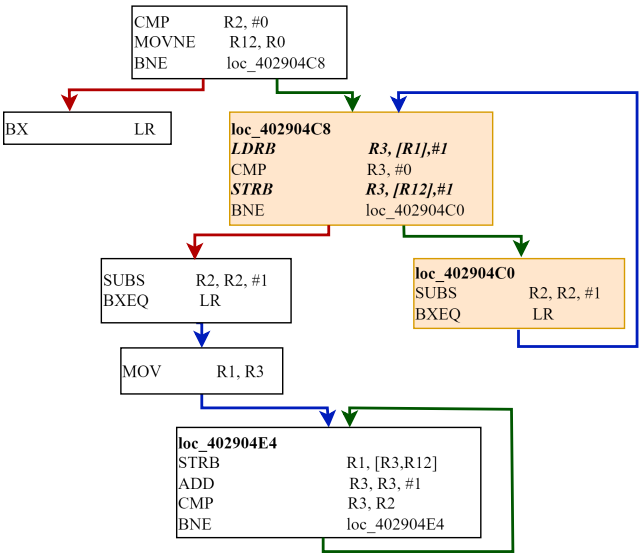


Figure 4: Custom copy functions share common characteristics and the strncpy funciton in ARM architecture exhibits such features.

Due to the variety of copy functions, pattern matching method results in low recognition accuracy. Therefore, we propose an engineering identification method based on control flow and data flow analysis. We first identify function

blocks with loop structures through control flow analysis. Then, we trace the data flow by converting corresponding assembly instructions into VEX IR intermediate language, ultimately determining whether the function has copy capabilities.

**Control flow analysis.** Control flow represents the relationships between basic blocks within a function [1]. By leveraging API functions provided by IDA, we can extract the basic blocks and their relationships within a function. The data is then used to construct a control flow graph (CFG) of the function. We finally use a cycle detection algorithm to identify loop structures within the CFG. If loop structures are detected, subsequent processing is performed. It is worth noting that, to prevent excessive system resource consumption, we have imposed limitations on the time and scale of loop structure identification. This means that we focus more on fine-grained copy functions (functions whose core functionality is loop copying with minimal additional functionality).

**Intermediate language translation and processing.** After identifying the loop structures in the CFG, we convert the instructions into VEX IR intermediate language to support multiple instruction architectures and improve analysis precision. Subsequently, we analyze VEX IR instructions to extract variables and their data relationships, recording as $\langle dst, src \rangle$ pairs according to the data flow. Additionally, we maintain sets of variables involved in memory loads (recorded as *MLD*) and memory stores (recorded as *MST*). Furthermore, we have added special handling for variables in the stack space. Temporary variables are assigned to addresses within the stack space, thus creating a more complete data flow. The correspondence between the assembly code, VEX IR instructions and analysis results is shown in the Figure 5.

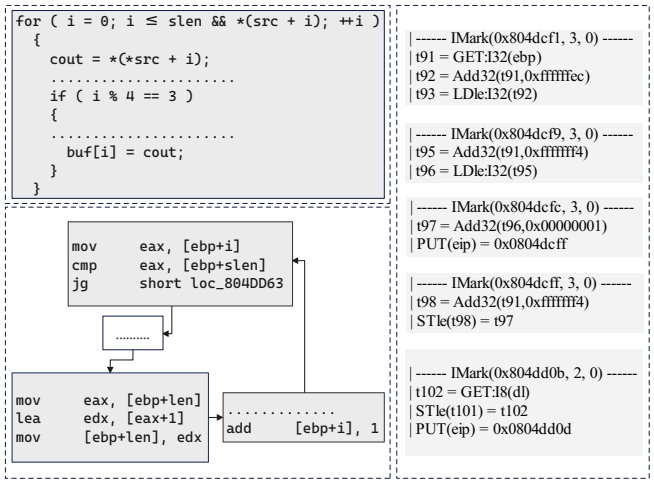


Figure 5: Schematic diagram of the relationships among various data types in LoopFinder.

**Data flow analysis.** After obtaining the data flow, a data flow graph (DFG) is constructed based on variable relationships. Drawing from empirical analysis, we have established the following matching rules. (1) Within the DFG, there exists variables in *MLD* and *MST*. (2) There exists a path from *MLD* to *MST* and *MLD* occur before *MST*. (3) There exists a loop structure in the DFG, within which there are computational operations. Additionally, points within the loop can reach *MST*. If the function's DFG satisfies above criteria, it is recognized as a copy function.

Furthermore, to improve accuracy, we analyze the parameters of potential copy functions. The function is classified as the copy function if the parameters are actually used within the loop structure (i.e., if the parameters serve as the source address, destination address or copy length). Since only functions with parameters related to the loop structure can be used as *sink* points for subsequent taint analysis, this approach ensures the accuracy of the identification process.

In addition, LoopFinder also identifies common memory copy and command execution functions, serving as foundational *sinks* for subsequent taint analysis. The common *sinks* covered by BPDA are mainly divided into two categories.

1) **Command injection:** *system(), popen(), execvp()*.
2) **Memory overflow:** *memcpy(), sprintf(), strcpy(), snprintf(), sscanf(), strcat()*.

## 6. Bidirectional Taint Analysis

The core method of BPDA is a bidirectional taint analysis system, which combines bidirectional path tracking and data flow analysis. Through bidirectional analysis, it effectively addresses the path explosion and excessive resource consumption encountered by SaTC and Mango. To improve the efficiency and accuracy of vulnerability analysis, based on previous analysis results, we primarily achieve BPDA through three designs: path generation, backward data flow tracking, forward taint analysis and Poc Generation.

### 6.1. Path generation

For subsequent bidirectional data flow analysis, it is necessary to first generate paths from source points (*src*) to sink points (*sink*). To enhance the accuracy and efficiency, we employed two optimization methods.

**Cross-boundary path generation.** Because some functions have data flow transmissions which are not based on call relationships. Directly generating paths from *src* to *sink* based on the function call graph can result in some missed paths. Therefore, we introduce transition functions to address the issue of path losses. We define four types of functions as transition functions: environment variables (*getenv, setenv...*), file read/write (*open, read, write...*) and network communications (*send, recv...*). We first define a data flow model to represent the data transmission paths between functions. In this model, nodes represent data attributes (such as environment variables, files, etc.) and edges represent the relationships of data between functions. Subsequently, based on the function call graph, we generate the reachable paths from *src* to the transition functions and from the transition functions to *sink*. Finally, based on the attribute values

of the transition functions, we connect the preceding and succeeding paths to generate the complete cross-boundary paths.

**Algorithm 1** Path Integration Analysis

**Input:** *paths* (Set of call paths)
**Output:** *merged_paths* (Merged call paths)
**function** PATH_INTEGRATION(*paths*)
  *call_trees* ← ∅
  **for** each *path* in *paths* **do**
    *group_by_initial_node*(path, *call_trees*)
  **for** each *tree* in *call_trees* **do**
    *queue* ← [*tree.root*]
    **while** queue is not empty **do**
      *node queue.pop(0)*
      **if** *node* has multiple children **then**
        *impact_sets* ← *analyze_impact*(node.child)
        **if** *consistent*(impact_sets) **then**
          *merge_paths*(node.child)
      *queue.extend*(node.child)
    **return** *merged_paths*
**end function**

**Path Integration.** When there are multiple call branches for a function and multiple layers of function calls, the final number of paths will grow exponentially. To address this issue, we design a path integration method. Through data flow analysis of branch functions, we integrate branch paths with consistent data flow impacts. We define the data flow impact as follows: the branch functions take parameters from parent functions and affect the parameters of child functions. The path integration is as shown in Algorithm 1, consists of the following steps. First, we categorize all paths based on initial nodes to generate different function call trees. Then, we perform parameter analysis on branch functions through the breadth first searching (BFS) approach. When branch functions originate from same parent functions and have a consistent impact on the parameters of child ones, we categorize them as similar functions and merge these branches. Finally, we perform analysis on each function call tree separately and merge similar paths.

## 6.2. Backward data flow tracking

Backward data flow tracking is used to eliminate uncontrollable paths from *sink* to *src* in the functions paths, targeting on improving the efficiency of subsequent taint analysis. The core idea of the algorithm is to analyze function call relationships and parameter transmission paths to identify and filter out paths that do not affect critical parameters. At the *sink* point, we first extract critical parameters. For example, for the *strcpy* function, we need to extract its source and destination addresses. By analyzing the decompiled pseudocode, we can locate the target *sink* function call. Parameter extraction requires traversing call nodes in the decompiled code, identifying calls that match the *sink* and then extracting the target parameter through compilation information.

Subsequently, by identifying the parameter and local variable declarations of the function, we extract the list of parameter and local variable in the functions. The parameter list includes arguments in the function call while the local variable list contains variable defined within the function, which aids us in identifying and tracking key parameters. To determine whether variables on the path are related to key parameters, we designed a recursive traversal algorithm, as shown in Algorithm 2.

**Algorithm 2** Backward Data Flow Analysis

**Input:** $sink, func, target$
**Output:** *Analysis result* (True of False)
**function** BACKWARD_ANALYSIS($sink, func, target$)
  $params \leftarrow$ extract_params($sink$)
**for** each $p$ in $params$ **do**
  **if** TRACE_BACK($p, func, target$) **then**
    **break**
**end function**
**function** TRACE_BACK($p, func, target$)
  $path \leftarrow$ get_path($func, target$)
**for** each $node$ in $path$ **do**
  **if** $p \in node$.rhs **then**
    $new\_p \leftarrow node$.lhs
  **if** TRACE_BACK($new\_p$, get_caller($func$), $target$) **then**
    **break**
**end function**

The algorithm examines the relationships between variables along the decompiled code path and determines whether key parameters are affected. Specifically, we start from the *sink* and traverse the function path in reverse, analyzing the code line by line, checking the definition and usage of each variable. If a variable is related to a key parameter, we continue tracing it until one of the following conditions is met. (1) The original variable is located in the function. (2) The variable is traced to an uncontrollable type (e.g., static strings, etc.). Moreover, there are some special cases to be handled, such as when the *src* string is directly passed to the function containing *src*. In such cases, the parameter index of the *src* string is recorded and used for subsequent tracking. Finally, we inspect each call path in the input file. Based on the results, we record valid and invalid paths to evaluate the effectiveness and improvement of the algorithm. The algorithm2 effectively filters out irrelevant function paths, thereby enhancing the accuracy and efficiency of subsequent taint analysis.

In addition, during the backward data flow tracking, BPDA maintains a filter list. When a function call node is filtered out by backward tracking, the filter list records the function node pair and the sequence of parameters. During subsequent path analysis, if the function call relationship and parameters are found in the filter list, the corresponding path is directly eliminated, further enhancing the efficiency.

### 6.3. Forward Taint Analysis

Based on the preliminary path results, we use taint analysis engine to analyze each path. To enhance the accuracy of taint propagation, we employ an extend analysis model with a tracking algorithm. During the taint analysis process, we not only maintain the taint status of variables but also record their defination types and operational states. Additionally, we perform sanitization during analysis.

**Type inference.** We have designed a type inference method to improve the accuracy of taint analysis. By determining the possible types for each variable through static type analysis, we employ type constraints during taint propagation to filter out irrelevant paths. Specifically, we first conduct static type analysis at the beginning of the process to determine the possible type ranges for each variable. During taint propagation, we use the type information to constrain the propagation paths of variables and update the constraints during execution process, ensuring that only paths meeting the type constraints are considered. For instance, consider the code statement: `var int_ip = atoi (user_input) + 'cgi-bin/binos'. user_input` is the variable we are tracking. After executing the code, the type of `user_input` is constrained to an integar by `atoi()`, preventing overflow or command injection vulnerabilities. Additionally, we accurately transfer type information across function calls to maintain type consistency in analysis.

**Sanitization.** We propose a sanitization method based on pattern recognition. Based on public available datasets and empirical analysis, we have estabilised a pattern library containing common sanitization operations such as validation, encoding and esacaping. During taint analysis, we can automatically identify these sanitization operations and mark the corresponding variables as safe through pattern matching. To ensure the accuracy of identification, we further combine data flow analysis to track the flow and transformation of variables, ensuring that variables remain safe after sanitization. For example, the code `var sanitized_input = htmlspecialchars (use_input, ENT_QUOTES, 'UTF-8').` Through pattern matching, we can automatically identify `htmlspecialchars` as a sanitization operation and mark the variable `sanitized_input` as safe. Additionally, we conduct context-sensitive analysis, taking into account the usage of variables in different contexts to avoid false positives.

### 6.4. PoC Generation

In actual vulnerability reports, a PoC is needed for validation. Therefore, BDPA combines the results of taint analysis to achieve automated generation and verification of PoCs. Through the bidirectional taint analysis discussed in Section 6, we have established all paths where data flows from the *src* to *sink* and have verified the harmfulness of the paths.

According to the source identification method described in Section 4, combined with practical firmware testing, we categorize the source points as follows. (1) Front-end data receive functions (*websGetVar, jsonObjectGetString...*). (2) Environment variable reception functions (*getenv, nvram_get...*). (3) File reading functions (*fgets, read...*). (4) Network reception functions (*recvfrom, recv...*). Based on the data interaction methods and parameter keywords of the source functions, data packets meeting the reception conditions can be constructed. Combined with Python-related communication modules, the PoC can be generated. For some source functions where the parameter construction method cannot be identified, we pre-generate attack data based on keywords and the PoC can be generated combined with subsequent manual analysis.

## 7. Implementation

We implemented BPDA prototype system through 4600 lines of Python code. SrcFinder partially utilizes shared string recognition method by SaTC and partially uses IDAPython for function recognition. Loopfinder mainly relies on IDAPython for loop structure identification and uses the NetworkX [29] library for graph processing. For intermediate language, it primarily uses PyVEX IR and is implemented through the PyVEX library. Path generation is implemented based on the function call relationships extracted by IDAPython. Backward data flow tracing mainly utilizes the CTree module in IDAPython, with functionality achieved by modifying and inheriting the CTree class. Forward taint analysis engine is implemented by Angr [2], enabling multi-architecture binary analysis. Due to the cross-architecture capabilities of IDAPython and Angr, BPDA is suitable for multi-architecture custom copy function recognition and taint analysis, applicable to X86, MIPS, ARM, PowerPC, etc.

## 8. Evaluation

We evaluated BPDA on real-world embedded devices to answer the following research questions:

**Q1:** How effective is the identification of custom loop copy functions outside of the libc library?

**Q2:** Compared to SaTC and Mango, how effective is BPDA in actual vulnerability discovery?

**Q3:** How does the bidirectional taint analysis method improve the efficiency of vulnerability discovery?

### 8.1. Datasets

We evaluate BPDA through three different firmware sets. (1) *SaTC Dataset.* The dataset used by SaTC, which includes totaling 39 firmware from six vendors such as Netgear, Tenda, Totolink and D-link. (2) *BPDA Dataset.* The supplementary dataset for BPDA, which includes 45 firmware images from vendors such as Cisco, TP-Link and Linksys. (3) *Loop Dataset.* Including Datacon2023 [31]

dataset and a custom-compiled dataset of copy functions. Additionally, to validate the effectiveness of LoopFinder on real firmware, we add the set with vulnerabilities in user-defined loop copy functions including both linux-based and VxWorks firmware, totally 11 firmware images.

## 8.2. Capability of Loopfinder

The custom-compiled dataset includes 7 common copy functions from the libc library (including strcpy, strncpy, memcpy, strcat, strncat, memmove, strlcpy). Along with other frequently used functions in firmware, we compiled them into binary files with different architectures. The Datacon2023 dataset contains 9 binary files, including 5 predefined copy functions. We use *precision* and *recall* to measure the capability of *LoopFinder*. The *precision* and *recall* are defined as formula, where TP represents true positives, FP represents false positives and FN represents false negatives.

$$precision = \frac{TP}{TP + FP}, \quad recall = \frac{TP}{TP + FN} \tag{1}$$

The comparison result of LoopFinder with SaTC and CPYFinder [42] is shown in the Table 1. It can be seen that LoopFinder outperforms the other two methods. Both LoopFinder and CPYFinder exhibit high recall rates. For self-compiled firmware, both methods accurately identify 7 copy functions across different architectures. However, the former demonstrates a significantly higher precision. This improvement is attributed to the analysis of relationship between functions parameters and loop structures, which effectively filters out irrelevant loop structures and enhances analysis accuracy.

TABLE 1: Comparison of LoopFinder, CPYFinder and SaTC.

| Binary program | SaTC | | CPYFinder | | LoopFinder | |
|---|---|---|---|---|---|---|
| | R | P | R | P | R | P |
| Dataset(arm) | 0.14 | 0.17 | 1.00 | 0.44 | 1.00 | 0.64 |
| Dataset(mips) | 0.14 | 0.17 | 1.00 | 0.44 | 1.00 | 0.64 |
| Dataset(powerpc) | 0.00 | 0.00 | 1.00 | 0.54 | 1.00 | 0.88 |
| Dataset(Datacon) | 0.00 | 0.00 | 0.60 | 0.23 | 1.00 | 0.38 |

Note: Precision(P) and recall(R) are measured to evaluate the effectiveness of copy function identification by 3 different tools.

By conducting an in-depth analysis of the relationship between function parameters and loop structures, we can more accurately capture the characteristics of memory copy functions, such as the usage of source address, destination address and length parameters. This targeted analysis enhances the logical rigor of the identification process, reducing false positives caused by structural similarities in code. By strengthening the semantic understanding of function calling conventions and runtime-like behavior, the identification process becomes more robust. It not only relies on pattern matching of loop structures but also focuses on the actual functionality and practical use of parameters, which improves identification accuracy.

It is worth noting that, for the Datacon dataset, LoopFinder successfully identify 5 copy functions, outperforming CPYFinder. As mentioned in Section 5, LoopFinder employs special handling for stack variables, which results in a significant improvement in recall rates for the x86 architecture.

**False positive.** Static analysis has inherent limitations and cannot fully capture runtime behavior and actual data flow. It relies on code structure characteristics, making it difficult to effectively distinguish between functions with similar structures but different functionalities. Additionally, there are limitations in assessing function return values. Notably, LoopFinder exhibits variations in false positives across different architectures. These differences arise due to compiler optimizations and the sensitivity of various instruction sets, leading to differences in analysis results.

## 8.3. Real Vulnerabilities

We tested actual vulnerabilities detection capability of BPDA. The tested firmware was primarily divided into two categories: one from the *BPDA Dataset* and the other from the real firmware in the *Loop Dataset*.

**Real vulnerabilities in Loop Dataset.** The real firmware in the *Loop Dataset* includes both linux and VxWorks types. We first use LoopFinder to identify suspicious *sink* points. Then we manually marks potential *src* functions and execute taint analysis through BPDA.

The effectiveness of actual vulnerability detection is shown in the Table 2. The *Loop Dataset* contains a total of 11 firmware images and 78,608 functions, of which 2,403 functions were identified as loop copy functions. Totally, 9 unknown vulnerabilities and 12 known vulnerabilities were identified. All 21 identified vulnerabilities are buffer overflow. Among them, the vulnerability in Citrix is located in the core service file *nsppe*, while the remaining Linux-based vulnerabilities are found in web service files.

The total time taken for each part of the analysis process is shown in the Table 3. The loop function identification process took a total of 4,927 seconds. After identifying sink points and manually marking input functions, a total of 1,030,887 function call paths were generated, taking 1145 seconds. After backward data flow tracking, 4,296 function call paths were remained for taint analysis, taking 1,923 seconds. Finally, forward taint analysis took 36,276 seconds. Based on the duration of each phase, it can be concluded that forward taint analysis occupies the major of analysis time in BPDA.

BPDA generated a total of 139 alerts. After manual analysis, 21 of these alerts were identified as real vulnerabilities. The test result indicates a certain level of false positives. In LoopFinder, we track the relationship between parameters and loop structures. During taint analysis, the alert is triggered when taint propagates to these parameters. However, BPDA does not conduct an in-depth analysis of the loop structure, leading to a certain degree of false positives. For

TABLE 2: The effectiveness of BPDA in detecting vulnerabilities in actual firmware from the *Loop Dataset*.

| Vendor | Series | Arch | Type | FN | LN | Prop (%) | Bug IDs |
|---|---|---|---|---|---|---|---|
| ASUS | RT-AC86U | ARM | Linux-based | 1,033 | 37 | 3.58 | CNVD-2024-33*** |
| | RT-AC53 | MIPS | Linux-based | 191 | 8 | 4.19 | CVE-2017-65*** |
| | RT-AC68U | ARM | Linux-based | 97 | 9 | 9.28 | CVE-2017-12*** |
| TP-Link | TL-WR940N | MIPS | Linux-based | 4,017 | 185 | 4.61 | CVE-2017-13***<br>CVE-2019-69*** |
| | TL-R600 | ARM | Linux-based | 3,878 | 322 | 8.30 | CVE-2018-39***<br>CVE-2018-39*** |
| | TL-WR841N | MIPS | Linux-based | 4,427 | 214 | 4.83 | CVE-2020-84*** |
| | TL-WDR7660 | ARM | VxWorks | 8,550 | 391 | 4.57 | CVE-2023-46***<br>CVE-2023-46***<br>CVE-2024-48***<br>CVE-2024-48***<br>CVE-2024-48***<br>CVE-2024-48***<br>2 unassigned |
| | TL-WR886N | MIPS | VxWorks | 7,692 | 248 | 3.22 | CVE-2023-46***<br>CVE-2023-46*** |
| D-Link | DIR-816 | MIPS | Linux-based | 1,170 | 53 | 4.53 | CVE-2018-11*** |
| Netgear | R6260 | MIPS | Linux-based | 388 | 11 | 2.84 | CVE-2021-34*** |
| Citrix | ADC Netscaler | x86 | Linux-based | 47,165 | 925 | 1.96 | CVE-2023-35*** |
| Total | 11 | - | - | 78,608 | 2,403 | 3.06 | 21 |

Note: *FN* represents the total number of functions in the firmware and *LN* represents the number of functions identified as loop copy functions. *Prop* represents the proportion of loop functions.

TABLE 3: Time consumption in *Loop Datset*.

| Phase | Time consumption (s) | proportion (%) |
|---|---|---|
| Loop function identification | 4,927 | 11.1 |
| Path generation | 1,145 | 2.6 |
| Backward data flow tracking | 1,923 | 4.3 |
| Forward taint analysis | 36,276 | 81.9 |

example, when LoopFinder detects a parameter used as the destination address in a loop structure, but the loop copy length is limited, BPDA may generate a false positive.

**Comparison with CPYFinder, SaTC and Mango.** For the real vulnerabilities discovered in the *Loop Dataset* by BPDA, SaTC and Mango were both unable to detect because they could not identify the actual *sink* points. The comparison between CPYFinder, SaTC, Mango and BPDA for the *Loop Dataset* is shown in the Table 4. CPYFinder successfully identified 20 vulnerabilities. However, it missed the vulnerability in the Citrix firmware for x86 architecture due to the lack of special handling for stack variables.

In terms of time consumption, BPDA shows a significant improvement over CPYFinder. CPYFinder does not analyze the parameters of Loop functions, leading to a high rate of misidentification of *sink* points. This results in a large

TABLE 4: The comparison with BPDA, CPYFinder, SaTC and Mango in analysis of *Loop Dataset*.

| Tool | Loop Number | Code Paths | Alerts | TruPocs | Time (s) |
|---|---|---|---|---|---|
| BPDA | 2,403 | 10,296 | 139 | 21 | 44,271 |
| CPYFinder | 3,549 | 21,175 | 394 | 20 | 92,793 |
| SaTC | – | – | 0 | 0 | – |
| Mango | – | – | 0 | 0 | – |

number of invalid function call paths, thereby increasing the false positive rate of alerts. As the result shows, compared to BPDA, CPYFinder identified 47.7% more loop copy functions, had 2.1 times the number of function paths and generated 2.8 times the number of alerts. The vulnerability detection efficiency of BPDA improved by 80.7% overall compared to CPYFinder.

**Real vulnerabilities in BPDA Dataset.** *BPDA Dataset* overlaps with *SaTC Dataset* to some extent but includes additional firmware vendors such as Cisco, Linksys, and TP-Link. All the firmware in the dataset is developed based on Linux. The experimental result for the *BPDA Dataset* is shown in the Table 5, listing some of 0-day vulnerabilities discovered by BPDA.

The vast majority of vulnerabilities in *BPDA dataset* are found in web services, including but not limited to Boa, GoAhead, and Httpd web service files. The vulnerabilities mainly include two types: buffer overflow and command injection. The involvement of various dangerous functions at *sink* points demonstrates the versatility of BPDA. In the *BPDA dataset* of 45 firmware, a total of 397 alerts were generated. After manual analysis, 142 were confirmed as vulnerabilities and 25 unknown vulnerabilities were assigned CVE/CNVD numbers.

The comparison with BPDA and SaTC in vulnerability analysis is as shown in Table 6. In the *BPDA Dataset*, SaTC generated 543 unknown alerts but only 104 were confirmed as vulnerabilities. BPDA employs type inference and sanitization as optimization methods in taint analysis process. Type inference allows for more accurate tracking and identification of potential taint sources, enabling precise marking and propagation of tainted data. Sanitization identifies and excludes taint propagation paths that are unlikely to cause security issues, further reducing false positives. Optimization methods combine precise data flow analysis with context-aware filtering, enhancing the effectiveness and accuracy of taint analysis. Furthermore, the reduction in paths also lowers the rate of false positives.

In path generation, BPDA utilizes a cross-boundary path generation method to ensure that the complete propagation paths of tainted data are captured, allowing for accurate analysis of cross-function call paths. As a result, compared to SaTC, it generates more effective alerts, further enhancing vulnerability analysis capabilities.

## 8.4. Efficiency

To verify the significant improvement in vulnerability detection efficiency through bidirectional taint analysis

TABLE 5: The effectiveness of BPDA in detecting vulnerabilities in actual firmware from the $BPDA\,Dataset$.

| Vendor | Series | Arch | PNumber | Bug IDs | Sink | Type |
|---|---|---|---|---|---|---|
| Dlink | DIR-822 | MIPS | 1,972 | CNVD-2024-40*** | sprintf | Bof |
| | DIR-816 | MIPS | 1,428 | CNVD-2024-68*** | strcat | Bof |
| | | | | CNVD-2024-64*** | system | CI |
| TPLink | TL-WPA8630 | MIPS | 407 | CNVD-2023-29*** | popen | CI |
| | | | | CNVD-2023-29*** | popen | CI |
| | | | | CVE-2023-27*** | system | CI |
| | | | | CVE-2023-27*** | system | CI |
| | TL-WPA7510 | MIPS | 346 | CVE-2023-29*** | execvp | CI |
| Tenda | AX12 | MIPS | 122 | CNVD-2024-33*** | strcpy | Bof |
| | | | | CNVD-2023-(***69∼***72) | strcpy | Bof |
| | | | | CNVD-2023-(***19∼***24) | sprintf | Bof |
| | | | | CNVD-2023-09*** | sscanf | Bof |
| | AC23 | MIPS | 3,185 | CNVD-2023-27*** | strcpy | Bof |
| | | | | CNVD-2023-19*** | strcpy | Bof |
| | W15E | ARM | 17,054 | CNVD-2024-40*** | strcpy | Bof |
| Linksys | E1500 | MIPS | 2,184 | CNVD-2024-21*** | sprintf | Bof |
| | | | | CNVD-2024-21*** | sprintf | Bof |
| Total | – | – | 26,698 | 25 | – | – |

Note: *PNumber* represents the number of paths generated during the taint analysis process. CI refers to Command Injection vulnerabilities and Bof refers to Buffer Overflow vulnerabilities.

TABLE 6: The comparison with BPDA and SaTC in vulnerability analysis.

| Tool | Firm-Number | Alerts | Vul-Number | Proportion (%) |
|---|---|---|---|---|
| BPDA | 45 | 397 | 142 | 35.8 |
| SaTC | 45 | 543 | 104 | 19.2 |

method, we set a comparative efficiency experiment. During testing on the *BPDA Dataset*, metrics such as the number of path generations and analysis time were collected and compared with SaTC. Additionally, the *BPDA Dataset* was tested on Mango for further comparison.

Compared to SaTC, the path filtering and efficiency improvements of BPDA are shown in the Table 8. As the result shows, for the 45 firmware samples in the *BPDA Dataset*, SaTC generated a total of 5,825,859 paths. After Backward data flow filtering, BPDA generated 253,637 paths, achieving a 94% reduction. During the test of SaTC, we set a timeout of 96 hours and severe timeouts occurred in five different vendors. In contrast, BPDA completed within the limited time and only a timeout occurred with one firmware, achieving on overall efficiency improvement of 94.0%. Through backward data flow analysis, BPDA effectively identifies and eliminates a large of parameter-independent paths, thereby reducing the paths for subsequent taint analysis.

Additionally, during backward tracing, BPDA does not require fine-grained taint analysis. Backward tracking only needs to identify the data flow relationships in parameters, making its propagation efficiency much higher than forward taint analysis. Therefore, it can quickly filter out invalid paths and significantly enhance the efficiency of subsequent forward taint analysis.

BPDA's efficiency improvements vary across different firmware due to differences in structural complexity and code organization. When firmware contains commonly called or calling functions which are intermediate nodes in the call chain, BPDA will significantly enhance path filtering. Conversely, when such structures are less prevalent, the filtering effect is reduced. However, experiments and analysis on multiple vendors show that such structures are prevalent, which is a primary reason for the excessive paths from *src* to *sink*. For firmware like Linksys, backward tracking can still reduce more than 20% invalid paths. Taint analysis execution time is related to path length and the amount of function code. The average time of taint analysis for function call chains shows minimal differences across different firmware.

We randomly selected 12 firmware from the *BPDA Dataset* and evaluated Mango on them. The effectiveness of SaTC, Mango and BPDA is shown in the Table 7. The efficiency improvement of BPDA over SaTC has been discussed earlier. The experimental results show that, for firmware with a smaller number of call paths, the time consumption of Mango is slightly less than or equal to BPDA and both have relatively low overall times. As the number of call paths increases, Mango's analysis time significantly rises and it frequently crashes with larger function call paths.

As the result shows, for some firmware in Dlink and Tenda, where function call volumes are high, Mango and SaTC are both unable to produce the final analysis result. The reasons for Mango's analysis failure can be summarized as follows. First, complex program logic and deeply nested function calls can cause an explosion in the number of paths during analysis. Mango needs to track the data flow of each function call and path. With multiple layers of function calls, the combination of paths grows exponentially as each layer can generate multiple return paths, creating new composite status upon merging. This exponential growth in paths leads to increased memory consumption and significantly longer analysis times.

In analyzing multi-layered paths, the number of states Mango needs to maintain increases sharply. During reverse taint analysis, Mango must maintain the taint status of multiple variables. Each data flow requires analysis and updating of taint information, increasing computational overhead. Especially for programs with deep function calls or complex control flows, updating and propagating taint states significantly slows analysis. Additionally, the mentioned path explosion leads to an explosion of taint state space, demanding substantial performance and resource consumption.

In contrast, the backward data flow tracing in BPDA is based on path generation results, avoiding to track all paths in the entire program. This reduces state space explosion, lowers memory consumption and decreases time overhead. BPDA avoid exhaustive data flow analysis for every function

TABLE 7: Efficiency comparison of BPDA, SaTC and Mango.

| Vendor | Sery | Arch | Bin | SaTC | | Mango | | BPDA | | P-Impr (%) | T-Impr1 (%) | T-Impr2 (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | N-Path | Time | N-Path | Time | N-Path | Time | | | |
| Dlink | DAP-1665 | mips | boa | 907,460 | Killed | – | Killed | 2,550 | 6.57 h | 99.7 | – | – |
| Dlink | DIR-605 | mips | boa | 20,623 | 53.21 h | – | 22.68 h | 2,503 | 6.46 h | 87.9 | 87.8 | 71.5 |
| Dlink | DIR-615 | mips | httpd | 1,623 | 4.18 h | – | 2.45 h | 1,394 | 3.63 h | 14.1 | 13.1 | -32.5 |
| Linksys | E1500 | mips | httpd | 2,797 | 7.41 h | – | 6.28 h | 2,184 | 5.64 h | 21.9 | 23.9 | 10.2 |
| Linksys | E2000 | mips | httpd | 3,026 | 7.87 h | – | 6.13 h | 2,324 | 6.01 h | 23.2 | 23.6 | 2.9 |
| Tenda | AC6 | mips | httpd | 471 | 1.26 h | – | 0.33 h | 122 | 0.37 h | 74.1 | 70.6 | -10.8 |
| Tenda | O6 | arm | httpd | 1,299,555 | Killed | – | Killed | 69,935 | 173.28 h* | 94.6 | – | – |
| Tenda | AX1803 | arm | tdhttpd | 1,296 | 3.36 h | – | 2.06 h | 912 | 2.46 h | 29.6 | 26.8 | -16.3 |
| Tenda | W15E | arm | httpd | 426,413 | Killed | – | Killed | 17,054 | 42.63 h | 96.0 | – | – |
| Tenda | W20E | arm | httpd | 742,336 | Killed | – | Killed | 26,977 | 67.44 h | 96.4 | – | – |
| TPLink | TL-WPA8630 | mips | httpd | 2,731 | 7.12 h | – | 2.75 h | 407 | 1.12 h | 85.1 | 84.2 | 40.7 |
| TPLink | TL-WPA7510 | mips | httpd | 2,964 | 7.69 h | – | 2.92 h | 346 | 0.95 h | 88.3 | 87.6 | 67.5 |

Note: *N-Path* represents the number of paths in taint analysis, *P-Impr* indicates the path filtering ratio of BPDA, *T-Impr1* shows the time improvement rate of BPDA compared to SaTC and *T-Impr2* shows the improvement rate of BPDA compared to Mango. *killed* indicates an abnormal termination of the tool during the analysis process. * means that the timeout occurred in BPDA but we still recorded actual runtime.

TABLE 8: Efficiency comparison of BPDA and SaTC for $BPDA\,Dataset$.

| Vendor | Sample | SaTC | | BPDA | | Impr (%) |
|---|---|---|---|---|---|---|
| | | Path | Time | Path | Time | |
| Cisco | 2 | 45,192 | Killed | 2,823 | 7.06 h | 93.8 |
| Dlink | 7 | 2,557,610 | Killed | 21,006 | 52.51 h | 99.2 |
| Linksys | 8 | 19,781 | 49.45 h | 15,435 | 38.59 h | 22.0 |
| Tenda | 11 | 2,655,056 | Killed | 170,358 | 425.36 h* | 93.6 |
| TPlink | 7 | 386,646 | Killed | 35,169 | 89.85 h | 90.9 |
| Netgear | 5 | 149,676 | Killed | 8,920 | 22.33 h | 94.0 |
| Totolink | 5 | 11,898 | 29.75 h | 926 | 2.32 h | 92.2 |
| Total | 45 | 5,825,859 | – | 253,637 | 638.02 h* | 94.0 |

Note: *Impr* represents the efficiency improvement rate of BPDA compared to SaTC. To ensure fairness in comparison, firmware that timed out in SaTC was not included in the calculations.
*: One firmware timed out in BPDA but we still recorded actual runtime.

call but focus on analyzing data relationships from the path results. Therefore, in backward data flow tracking of BPDA, maintaining taint states is unnecessary, resulting in lower computational resource demands and more efficient vulnerability analysis.

# 9. Discussion

In this section, we mainly discuss the capabilities and limitations of BPDA.

## 9.1. Strengths and Scope

BPDA demonstrates a significant capability in enhancing the detection of security vulnerabilities within IoT firmware by employing a bidirectional path and data flow analysis approach. The advantage lies in its ability to perform both forward and backward analysis, offering a comprehensive view of data flow from user input to potential vulnerability points. By integrating forward analysis to identify paths leading to vulnerabilities and backward analysis to filter out invalid paths, BPDA significantly enhances vulnerability analysis efficiency. It also enhances vulnerability detection by supporting the identification of custom *sink* point.

BPDA is well-suited for IoT firmware, effectively addressing common vulnerabilities like command injection and buffer overflow. It can scale to analyze numerous binary files and firmware samples without sacrificing accuracy. This adaptability makes it highly effective in the IoT ecosystem.

## 9.2. Capability Development

While BPDA is suitable for IoT devices, some of the modules offer excellent application extensibility. Based on the identification mechanism through code structure and data flow characteristics, LoopFinder demonstrates strong extensibility beyond IoT device analysis. By recognizing loop structures and memory access patterns, it can detect custom copy functions. Combined with other software vulnerability analysis method, it can effectively improve the ability in software vulnerability detection.

## 9.3. Limitations.

**Coarse-grained backtracking has filtering limitations.** In backward data flow tracing, the granularity directly impacts the accuracy of path filtering. Coarse-grained backtracking may fail to accurately identify and filter out all invalid paths, resulting in false positives and affecting the precision of the analysis.

**Ignoring hidden source points.** Although in SrcFinder, BPDA addresses the limitations of shared string by adding feature recognition for *src* points, it still misses some hidden or user-defined *src* points. Without detecting critical input points, BPDA can not trace the related data flow and construct function paths, result in missing some vulnerability points.

**The false positive of LoopFinder.** LoopFinder has a certain false positive reate when identifying copy funcitons. The implementation relies on code structural features, such as loop structures and memory access patterns. Therefore, LoopFinder might mistakenly identify some non-copy functions with similar structures but different functionalities. False positives at sink points can further increase the number of subsequent path generations, affecting the efficiency and accuracy of vulnerability detection.

### 9.4. Future work

For addressing the limitations of BPDA, future work will focus on enhancing analysis precision and efficiency. First, more refined and efficient backtracking algorithms should be developed to improve path filtering accuracy and reduce false positives. Second, SrcFinder can be enhanced using advanced technologies like machine learning to identify hidden or user-defined source points, ensuring comprehensive data flow tracking. Additionally, LoopFinder can be optimized through in-depth semantic analysis, thereby reducing false positives. Furthermore, we can combine static and dynamic analysis, using runtime information to validate static analysis results. Such improvements will enable BPDA to identify security vulnerabilities in IoT firmware more effectively, enhancing its reliability and efficiency in practical applications.

## 10. Conclusion

In this paper, we propose a bidirectional path and data flow analysis method (named BPDA) to efficiently detect buffer overflow and command injection vulnerabilities in IoT devices. By refining the sources and sinks, then conducting bidirectional path and data flow analysis, BPDA constructs more comprehensive and effective taint analysis paths and efficiently execute taint analysis. Experiments demonstrate that BPDA can recognize user-defined copy-like functions effectively and bidirectional taint analysis can enhance the efficiency of vulnerability analysis significantly. In 56 firmware samples, BPDA identified a total of 163 vulnerabilities, including 34 0-day vulnerabilities and 32 were assigned CVE/CNVD numbers. The vulnerabilities were located in custom copy functions and common dangerous functions, indicating the effectiveness of LoopFinder and bidirectional taint engine. Our evaluation shows that BPDA outperforms the similar tools in detecting vulnerabilities in IoT devices.

## Acknowledgments

The authors would like to thank...

## References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1):1–40, 2009.

[2] Angr. Github -angr/angr: A powerful and user-friendly binary analysis platfom!, 2024. https://github.com/angr/angr.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[4] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094, 2014.

[5] Feng Bo and Lu Alejandro, Meraand Long. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[6] Oliver Buxton. Mirai botnet ddos attack: What is the mirai botnet?, 2022. https://www.avast.com/c-mirai.

[7] Vincent Li Cara Lin. Botnets continue exploiting cve-2023-1389 for wide-scale spread, 2024. https://www.fortinet.com/blog/threat-research/botnets-continue-exploiting-cve-2023-1389-for-wide-scale-spread.

[8] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium*, 2016.

[9] 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 *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[10] 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 *30th USENIX Security Symposium*, 2021.

[11] 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 *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 430–441. IEEE, 2018.

[12] Kai Cheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, and Limin Sun. Finding taint-style vulnerabilities in linux-based embedded firmware with sse-based alias analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 360—-372. ACM New York, NY, USA, 2023.

[13] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium*, pages 1201–1218, 2020.

[14] CVEDetails. A buffer overflow in the httpd daemon on tp-link tl-wr841n v10. https://www.cvedetails.com/cve/CVE-2024-7029.

[15] CVEDetails. A command injection in the web interface of tp-link archer ax21 (ax1800). https://www.cvedetails.com/cve/CVE-2023-1389.

[16] CVEDetails. Nvd-cve-2022-26258. https://nvd.nist.gov/vuln/detail/CVE-2022-26258.

[17] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium*, pages 303–317, 2014.

[18] Wil Gibbs and Raj. Operation mango: Scalable discovery of taint-style vulnerabilities in binary firmware services. In *USENIX Security Symposium*, pages 312–326, 2024. https://www.usenix.org/system/files/sec24fall-prepub-1634-gibbs.pdf.

[19] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, pages 135–150, 2019.

[20] Hex-ray. The ida pro disassembler and debugger, 2021. https://www.hex-rays.com/products/ida/.

[21] Tay Hui Jun, Zeng Kyle, Vadayath Jayakrishna, Menon, Raj Arvind, S, Dutcher Audrey, Reddy Tejesh, and Gibbs Wil. Greenhouse: Single-service rehosting of linux-based firmware binaries in user-space emulation. In *Proceedings of the the 32nd USENIX Security Symposium*, 2023.

[22] Alex.Turing Hui Wang. The botnet cluster on the 185.244.25.0/24, 2019. https://blog.netlab.360.com/the-botnet-cluster-on-185-244-25-0-24-en/.

[23] Zhenbang Ma ibo Chen, Quanpu Cai. Sfuzz: Slice-based fuzzing for real-time operating systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[24] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[25] ReFirm Labs. Binwalk: Firmware analysis tool, 2021. https://github.com/ReFirmLabs/binwalk.

[26] Peiyu Liu, Shouling Ji, Xuhong Zhang, Qinming Dai, Kangjie Lu, Lirong Fu, Wenzhi Chen, Peng Cheng, Wenhai Wang, and Raheem Beyah. Ifizz: Deep-state and efficient fault-scenario generation to test iot firmware. In *36th IEEE/ACM International Conference on Automated Software Engineering*, pages 805–816. IEEE, 2021.

[27] Kim Mingeun, Kim Dongkwan, Kim Eunsoo, Kim Yongdae, Jang Yeongjin, and Kim Suryeon. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *In Annual Computer Security Applications Conference (ACSAC 2020)*, 2020.

[28] 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 *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[29] Networkx. Network analysis in python, 2021. https://networkx.org/.

[30] 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*, 54(2):1–42, 2021.

[31] Qianxin. Datacon 2023 big data security analysiscompetition. https://datacon.qianxin.com/datacon2023.

[32] 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 *IEEE Symposium on Security and Privacy*, pages 1544–1561. IEEE, 2020.

[33] BALDONI ROBERTO, COPPA EMILIO, CONO D'ELIA DANIELE, DEMETRESCU CAMIL, and FINOCCHI IRENE. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39, 2018.

[34] Yu Ruotong, Nin Francesca, Del, Zhang Yuchen, Huang Shan, Kaliyar Pallavi, Zakto Sarah, and Conti Mauro. Building embedded systems like it's 1996. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

[35] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236, 2009.

[36] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise mmio modeling for effective firmware fuzzing. In *31th USENIX Security Symposium*, 2022.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, pages 138–157. IEEE, 2016.

[38] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: Automated iot firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pages 15–21, 2019.

[39] LIU Ya. The gafgyt variant vbot seen in its 31 campaigns, 2020. https://blog.netlab.360.com/the-gafgyt-variant-vbot-and-its-31-campaigns/.

[40] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[41] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.

[42] Xiaokang Yin, Bin Lu, Ruijie Cai, Xiaoya Zhu, Qichao Yang, and Shengli Liu. Memory copy function identification technique with control flow and data flow analysis. *Journal of Computer Research and Development*, 60(2):326–340, 2023.

[43] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium*, volume 23, pages 1–16, 2014.

[44] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *IEEE Symposium on Security and Privacy*, pages 813–832. IEEE, 2021.

[45] 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 *28th USENIX Security Symposium*, pages 1099–1114, 2019.