Faster and Better: Detecting Vulnerabilities in Linux-based IoT Firmware with Optimized Reaching Definition Analysis

Zicong Gao[1,2], Chao Zhang[3,4*], Hangtian Liu[1,2], Wenhou Sun[3], Zhizhuo Tang[1,2], Liehui Jiang[1,2], Jianjun Chen[3], Yong Xie[5]
[1]School of Cyber Science and Engineering, Information Engineering University,
[2]State Key Laboratory of Mathematical Engineering and Advanced Computing,
[3]Tsinghua University, [4]Zhongguancun Laboratory, [5]Qinghai University

***Abstract*—IoT devices are often found vulnerable, i.e., untrusted inputs may trigger potential vulnerabilities and flow to sensitive operations in the firmware, which could cause severe damage. As such vulnerabilities are in general taint-style, a promising solution to find them is static taint analysis. However, existing solutions have limited efficiency and effectiveness. In this paper, we propose a new efficient and effective taint analysis solution, namely HermeScan, to discover such vulnerabilities, which utilizes reaching definition analysis (RDA) to conduct taint analysis and gets much fewer false negatives, false positives, and time costs. We have implemented a prototype of HermeScan and conducted a thorough evaluation on two datasets, i.e., one 0-day dataset with 30 latest firmware and one N-day dataset with 98 older firmware, and compared with two state-of-the-art (SOTA) solutions, i.e., KARONTE and SaTC. In terms of effectiveness, HermeScan, SaTC, and KARONTE find 163, 32, and 0 vulnerabilities in the 0-day dataset respectively. In terms of accuracy, the true positive rates of HermeScan, SaTC, and KARONTE are 81%, 42%, and 0% in the 0-day dataset. In terms of efficiency, HermeScan is 7.5X and 3.8X faster than SaTC and KARONTE on average in finding 0-day vulnerabilities.**

I. Introduction

The number of Internet of Things (IoT) devices has grown exponentially over the past decade and has reached 14.4 billion by the end of 2022 globally [13]. Meanwhile, a huge amount of IoT devices exposed in the network with weaknesses in their firmware are susceptible to further attacks. According to a report [20], up to one billion IoT devices have been attacked in 2021. Note that firmware is the most critical component of the device, responsible for running the customized programs of the vendor on the device. Due to the closed-source and hard-to-upgrade nature of device firmware [38], [32], they are often vulnerable, and particularly easily affected by taint-style vulnerabilities [7], resulting in serious security issues.

Among IoT devices, Linux-based devices are prevalent and widely used in routers. Thus, detecting vulnerabilities in their firmware is crucial for protecting the security of Linux-based IoT devices. As fuzzing is now the dominant solution to finding software vulnerabilities, many fuzzing solutions are proposed for IoT firmware. These solutions (e.g., [42], [10], [40], [39]) generally have very high true positive rates but have troubles with the testing environment because the firmware is hardware-dependent. Some solutions try to rehost firmware with emulators to enable fuzzing but have limited success rates. For example, the SOTA rehosting-based firmware fuzzing solution FirmAE [14] can only successfully emulate 79% of the network services of the Linux-based firmware on its dataset. Yet, there is no dynamic solution feasible for testing all firmware.

On the other hand, static analysis solutions do not require hardware support and could be applied to large-scale IoT firmware security analysis. Note that most IoT firmware vulnerabilities are taint-style, i.e., they are caused by insecure data flow from untrusted inputs to sensitive sinks in firmware. Thus, several static taint analysis solutions, e.g., DTaint [5], KARONTE [26], SaTC [3], and Emtaint [6] have been proposed to detect firmware vulnerabilities. Compared to dynamic solutions, they are more practical to be applied to IoT devices. However, they could be further improved to reduce false negatives, false positives, and time costs.

In this paper, we propose a new taint analysis solution HermeScan, which directly applies traditional reaching definition analysis (RDA) to the VEX intermediate representation (IR) of binaries, to detect taint-style vulnerabilities in Linux-based IoT firmware. The customized RDA generates a data dependency graph that describes the def-use and use-def data flow relationship of variables. By looking up the graph reversely, HermeScan could check whether untrusted data at the source points could reach operands of the sensitive sink operations. An alert is raised if so. Specifically, it addresses the following 3 challenges and greatly promotes the effectiveness, accuracy, and efficiency compared to existing solutions, making the solution practical.

First, existing static taint analysis solutions often suffer from high false negatives due to incomplete control flow graphs (CFG) and loss of source points. In particular, the CFGs of these solutions may be incomplete, especially when calling library functions from binaries. Additionally, they miss many taint/input sources. For instance, SaTC relies on precise keyword matching to determine input sources between the

*Corresponding author: chaoz@tsinghua.edu.cn

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24346
www.ndss-symposium.org

front-end (e.g., web pages) and back-end files (e.g., binaries) or the IPC communication. *HermeScan addresses these issues by using two complementary schemes.* One scheme is analysing the library dependency of each firmware module and links functions in different modules together to perform RDA, which can uncover source points and sink points missed by other solutions. Another scheme is a fuzzy matching strategy to find more strings shared with front-end programs, enabling the identification of back-end functions that reference these strings as *candidate source functions*.

Second, existing static taint analysis solutions often have high false positives, especially when the program path to analyze gets longer. Therefore, some solutions [3] propose to shorten the path to analyze, e.g., by setting the *source functions* in the back-end that accesses the shared strings as starting points of taint analysis. However, analysis starting from these points without considering data flow context (e.g., input sanitization) would cause over-tainting to the source functions' parameters and return value, leading to inaccurate analysis. *HermeScan examines the usage of the parameters and return values of candidate source functions to determine which variables should be marked as tainted and applies data-flow constraints on tainted variables to reduce the false positive*.

Lastly, existing static taint analyses have high time overheads due to the large number of program paths to analyze or the time cost of slow symbolic execution. Instead, HermeScan utilizes RDA to perform taint analysis, which is more lightweight. However, when applying RDA to interprocedural analysis scenarios, it still faces a performance issue due to the large number of program paths to analyze. This path explosion issue gets aggravated as the number of the paths between taint source and sink points increases. To address this problem, *HermeScan applies a lightweight, on-demand, context-sensitive RDA to achieve efficient inter-procedural taint tracking*. Furthermore, a *path-merging strategy* is designed to reduce the number of repetitive analyses, to mitigate the path explosion issue.

We have implemented a prototype of HermeScan based on angr [30] and IDA Pro [9] with 4K lines of Python code. We have conducted a thorough evaluation of HermeScan on two datasets, i.e., one 0-day dataset with 30 latest firmware and one N-day dataset with 98 older firmware with over 400 known vulnerabilities, and compared its performance with that of two state-of-the-art (SOTA) solutions KARONTE and SaTC. The results showed that HermeScan could find more vulnerabilities with lower false positives/negatives and less time than baselines, showing the solution is practical.

For the 0-day dataset, in terms of effectiveness, HermeScan, SaTC, and KARONTE find 163, 32, and 0 vulnerabilities respectively. We have reported these vulnerabilities to vendors following the responsible disclosure procedure. Among these 163 vulnerabilities found by HermeScan, 76 are known to the vendors (but not fixed in the latest firmware), and 87 are unknown to them (69 are assigned with CVE numbers). In terms of accuracy, the true positive rates of HermeScan, SaTC, and KARONTE are 81%, 42%, and 0%. In terms of efficiency, HermeScan is 7.5X and 3.8X faster than SaTC and KARONTE on average in finding 0-day vulnerabilities. For the N-day dataset, the evaluation results are similar.

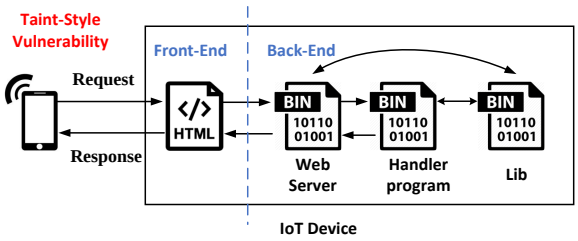

Fig. 1: The simplified threat model of taint-style vulnerabilities in IoT devices, which often come with a proxy-like web server to process user inputs (e.g., for management).

In summary, we make the following contributions:

- We present a lightweight, on-demand, context-sensitive RDA solution to detect taint-style vulnerabilities in IoT firmware, which is more effective, accurate, and efficient.
- We have implemented a prototype of HermeScan and discovered 87 0-day vulnerabilities in real-world devices.
- We build two sets of firmware samples and comprehensively evaluate the performance of existing tools.

To foster future research, we will release the source code of HermeScan.*

II. Background

A. Threat Model

In this paper, we focus on taint-style vulnerabilities [5], [11], [4], [3] in Linux-based IoT devices. Figure 1 depicts the scene of taint-style vulnerability: An attacker can access the target device over a local or wide area network and send arbitrary data to the device with no restriction. The data is first parsed in the front-end files (such as JavaScript and HTML files) provided by the firmware. Then on the device side, the data is propagated to the Web service program in the back-end. The Web service program further transmits the data to the handler program that operates the device. During this process, some library files are loaded to provide the necessary support. The binary programs that provide Web service in the back-end are often called *border binaries*, which usually include Web server and handler programs. The Web server and the handler program can also be integrated into a single binary, such as *httpd*, which router vendors frequently customize.

In this case, without considering the hardware and software protection of IoT devices, untrusted data can flow into unsafe functions in the binary, causing security issues. Table II lists the common taint-style vulnerabilities in our threat model, which are also the targets our work supports.

B. Taint analysis

Taint analysis is a widely used technique for software analysis, which abstracts the program as a 3-tuple $\langle sources, sinks, sanitizers \rangle$ [35], [22]. Sources are where the program introduces dangerous or uncontrolled data from the outside world. **Identification of taint sources** is determined manually or automatically based on the characteristics of the target. Sinks refer to specific points in an application's code

*https://github.com/f01lprophet/HermeScan

where tainted data is utilized in a potentially unsafe or insecure manner. Sink identification often relies on sensitive functions. **Taint propagation** starts by labeling the input data as tainted (**taint tag management**) at the source point and tracks how the tainted data flows through the program, using either static or dynamic methods. During taint propagation, sanitizers filter out dangerous data or transform data into secure data. The ultimate goal of taint analysis is to find a specific path along which the input of the source point can flow to the position of the sink point and break some security properties. However, taint analysis often involves redundant paths, and efficient **path merging** techniques can greatly enhance analysis efficiency.

Previous works of static taint analysis propose different optimization strategies based on the characteristics of the application scenario, from **taint source identification**, **taint tag management**, **taint propagation**, and **path merging**. We also propose our own optimization solutions in these four aspects, and the innovation is described in detail in Section VIII-A.

C. Reaching Definition Analysis

Reaching definition analysis is a lightweight data flow analysis, which is originally used to solve the following problem: give a definition d to variable v, whether there is a path so that the program point p can reach q, and v cannot be changed in the process assignment [18]. If we regard p and q as source and sink points, and assigning value d to variable v represents marking the input as a specific taint value. Then we can determine whether the taint value can reach sink points through RDA. If it can be reached, we can observe the possible values of the variable at point q to determine whether the security property is violated. As a path-insensitive may-analysis, RDA has two advantages in applying it to taint analysis. First, due to path insensitivity, RDA does not need to explore the branches like symbolic execution methods and can efficiently traverse all statements without considering the problem of state explosion. Secondly, RDA is a *may-analysis* proven to reach the fixed point. Therefore, we can obtain the possible values of the tainted variable at the sink point and avoid the difficult problem of a symbolic solution.

D. Motivating Example

Figure 2 shows an example of an OS command injection in the binary of an ASUS router. The process of triggering the vulnerability is as follows: *httpd* queries its registered **mime_handler** array after receiving the HTTP request. When the array index points to function **appGet.cgi**, it further calls the function **do_ej**, where it checks whether the value of the output field in the HTTP request exists in the function list stored by **ej_handler**. When the value matches the string `bwdpi_monitor_info`, the function **ej_bwdpi_monitor_info** is called and obtains the value of *type* and *event* from the user's input through the function **websGetVar**. The values of *type* and *event* are finally passed to the **bwdpi_monitor_info** located in the private library *libbwdpi_sql.so*. After an inappropriate string concatenation operation, the parameters of the system function can be set by unrestricted user input.

We tried two SOTA works, SaTC and KARONTE, to analyze *httpd*, but neither could produce the alert of the above vulnerability. Our investigation indicates that existing

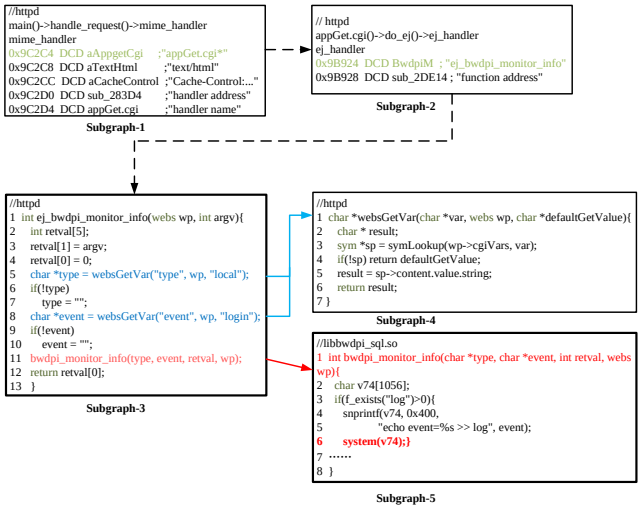

Fig. 2: A simplified example of the zero-day vulnerability in ASUS router. The function ej_bwdpi_monitor_info in the *httpd* program introduces unrestricted user input in the websGetVar function (blue part), resulting in OS command injection (red part) in the library file *libbwdpi_sql*.

tools ignore the source and sink points. In terms of source, because the program uses indirect jumps to point to the array of stored functions, the functions stored in **ej_handler** and **mime_handler** are not correctly added to the CFG. Eventually, the input from **websGetvar** is not included in the analysis. In terms of sink points, because the existing tools do not consider the control flow from the main program to the shared link library, the system-execution functions existing in the library *libbwdpi_sql.so* are also not analyzed.

A straightforward idea to deal with the problem of losing the source point and sink point is to solve the indirect calls. However, for the former, the existing methods to resolve indirect calls are not suitable for the scenario of firmware static analysis due to low efficiency [15] or limited support for multi-architecture [12]. For the latter, the reason why existing work [3], [26] does not consider the propagation of data flow in library code is the high overhead of taint tracking based on symbolic execution. The average time to analyze a sample by SaTC or KARONTE is 0.5~30h, and the time is positively correlated with the program scale and the number of paths to be analyzed.

To efficiently and accurately discover the vulnerability in the motivating example, our intuition is: taint tracking from the source point to the sink point could be essentially converted to a problem judging whether the input data is reachable at a specific point. Thus, the motivating example can be solved by using RDA based on an extended CFG and accurate identification of taint sources. If the function **ej_bwdpi_monitor_info** is successfully identified as an orphan node in the CFG, then we can obtain the source point of **websGetVar**. If we consider the control flow between the bin and library, the system function at the sink point can also be found. Finally, we can efficiently discover the vulnerability by performing lightweight RDA between the determined source point and sink point.

III. Challenges

Although the RDA based on enhanced control flow information is suitable for analyzing taint-style vulnerabilities, we have to face the three challenges categorized as follows when the intuition is adopted on real-world firmware.

A. Comprehensive CFG Recovery

In the context of static analysis, CFG recovery helps identify source and sink points in the program for subsequent taint analysis. Existing work either builds an independent CFG for each function [5] or directly adopts the CFG construction strategy [3], [26] in angr [30]: gradually expanding nodes and edges from identified program entry points.

However, these methods for constructing CFG are insufficient, resulting in some potential source points and sink points not being considered in the taint analysis. Specifically, firstly, many functions of the program in the embedded device are triggered through a callback mechanism. Such a mechanism is represented as indirect calls and is difficult to resolve. Secondly, current works do not analyze the control flow between the target program and its loaded library for efficiency. Note extending control flow analysis to libraries is not challenging (e.g., in angr can be enabled by load_libs flag). However, introducing the control flow between Bin and Lib dramatically increases the time of the analysis, making the symbolic execution-based method not scalable. Therefore, the CFG needs to accurately identify the registered functions and take into account the control flow between the program and the loaded library.

B. Precise Source Point Identification

In the scenario of firmware programs, various vendors use different functions to parse input data from the received network packet, where these functions are often regarded as the starting point of taint analysis.

Existing static analysis methods either manually specify source functions, or automatically locate the locations of these source points through shared strings. However, the former method is very dependent on expert knowledge and needs to be customized to specify source functions for different devices. The latter method also has two disadvantages. One is that the precise matching strategy does not consider the possible semantic differences between the front and back ends when parsing a key string. The second is that all the parameters and return values of the function at the source point are marked as tainted, which leads to false positives from over-tainting.

C. Efficient Taint Tracking

Schemes of taint tracking of user input affect the efficiency of firmware static analysis. As discussed in Section II-D, using RDA-based taint tracking schemes can detect taint-like vulnerabilities faster than symbolic execution-based methods [3], [26].

Unfortunately, using RDA-based taint tracking still has two obstacles in practice. Firstly, there are massive paths between source points and sink points, which leads to the problem of path explosion in RDA. Secondly, if each calling function is considered in the inter-procedural analysis, data flow tracking becomes very complicated. The data flow of some callee functions is not directly related to user input. Thus, efficient taint tracking should be capable of dealing with complex inter-procedural analysis and path explosion.

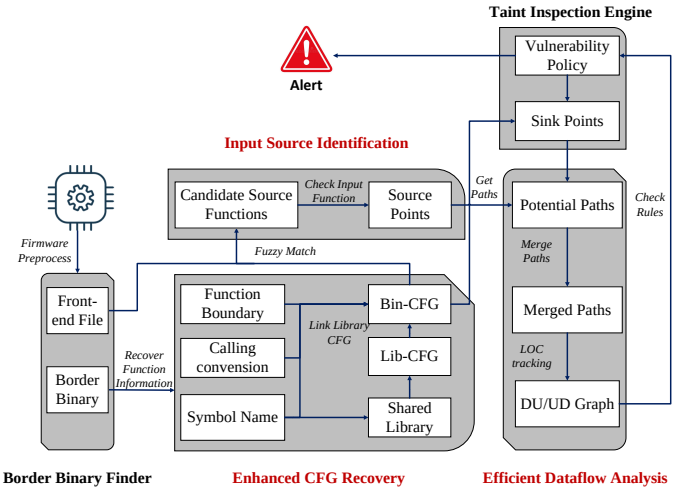

Fig. 3: Overview of HermeScan.

IV. Design

A. Overview

To address these challenges, HermeScan adopts three key schemes: enhanced CFG recovery, accurate source input identification, and efficient dataflow analysis. As shown in Figure 3, HermeScan analyzes a firmware following five steps.

Border Binary Finder: HermeScan takes the same method as SaTC [3] to locate the border binary. We extract the file system of the firmware and collect the shared strings referenced both in the front-end and back-end files. The binary file containing the most matching shared keywords is marked as the border binary.

Enhanced CFG Recovery: Then in the enhanced CFG recovery step, HermeScan obtains function information from border binaries, including function bounds, calling conventions, and symbolic names. Taking advantage of the above information, HermeScan converts the binary to an intermediate representation and builds a more **comprehensive CFG**.

Source Input Identification: HermeScan first uses fuzzy matching to locate candidate source functions through the shared strings of the front and back ends to **reduce false negatives** of exact matching. Then it uses an RDA-based method to determine the functions representing user input from the candidates and assign suitable taint marks to their parameters, which **avoids false positives** caused by over-taint.

Efficient Dataflow Analysis: The module adopts a path merging strategy to **alleviate path explosion** and designs a lightweight, context-sensitive, on-demand RDA to **improve the efficiency** of taint tracking along each path.

Taint Inspection Engine: The taint detection engine observes the data flow information from the def-use graph at sink points and checks whether it breaks the detection rule of vulnerability.

B. Enhanced CFG Recovery

The richness of information in the CFG profoundly affects the identification of source points and sink points, and further affects the false negative rate of taint analysis. From a practical perspective, we solve the shortcomings of existing solutions in terms of function boundary identification, symbol name finding, calling convention recovering, etc., to provide a solid foundation for HermeScan's RDA as much as possible.

Function boundary: The more functions are identified, the wider the scope of the analysis can be extended. Thus, in addition to functions automatically recognized by the disassembler [9], HermeScan scans the disassembled code of the whole binary to recognize function boundaries missed by the disassembler, by matching the features (e.g., stack operations) of the function prologue under different architectures.

Symbol name: The symbol name is used to identify which library functions are loaded by the binary. Both SaTC and KARONTE rely on angr to recover symbol names of external library functions in the binary program by analyzing the section header table. However, the approach is not sufficient for binaries that have stripped section header tables, leading to some function symbol information being missed.

To address this issue, HermeScan parses ELF files based on the execution view as a supplement [2]. Specifically, our approach interprets the program header table to recover symbols. Firstly, we traverse all program header tables and identify the `PT_DYNAMIC` segment corresponding to `p_type`. Then, we search for the value of the `DT_SYMTAB` tag in the segment `PT_DYNAMIC` to locate the address of the ELF symbol table. Finally, we ensure more symbol information is obtained from the symbol table.

Calling convention: HermeScan extends angr's default calling convention and implements a more aggressive but complete recovery strategy. For the function's calling convention that is not successfully acquired by angr, HermeScan assigns preset calling conventions (CC) according to the architecture of the program (e.g., the default parameter call for MIPS architecture is to use a0-a3 registers). Providing defaulting value for CC ensures subsequent data flow analysis is closer to the *may-analysis*. Since our taint tracking determines whether to step into a function based on whether its parameters are tainted, if a function's CC is not recognized, its argument is treated as None, which may lead to false negatives.

After obtaining the above knowledge, HermeScan enhances the construction of CFG with the aid of this information. Different from constructing CFG adopted by SaTC and KARONTE, HermeScan treats each function as a dominant node to build independent control flow subgraphs and then connects them by the jump or call instructions. At the same time, HermeScan establishes the Lib-CFG for the shared library loaded by the binary and utilizes the symbol name to connect the Bin-CFG with Lib-CFG.

C. Source Input Identification

The precise identification of input sources is essential to reducing false positives caused by over-tainting and false negatives caused by the loss of source points. HermeScan proposes Algorithm 1 to achieve precise source input identification, which comprises two parts. The first part employs a fuzzy matching strategy to obtain more candidate source functions. The second part assigns appropriately tainted initial values to return values or parameters of the source function by checking candidate functions.

1) Fuzzy Matching Strategy: HermeScan employs a fuzzy matching strategy to screen out the shared strings in front-end and back-end files. Unlike SaTC and KARONTE, we consider both word-form similarity and semantic similarity to match keywords in addition to string consistency when the back-end program parses the strings from the front end.

$$MATCH(S1,S2) = \begin{cases} True, & FormatSim(S1,S2) \geq \alpha \text{ or} \\ & SemanticSim(S1,S2) \geq \beta \\ False, & Others \end{cases} \quad (1)$$

Equation 1 defines the matching result of strings s1 and s2, where it holds in two cases: (1) when the words-form similarity FormatSim(s1, s2) exceeds threshold α, or (2) when the semantic similarity SemanticSim(s1, s2) exceeds than threshold β.

$$FormatSim(S1,S2) = 1 - Edit(S1,S2)/(L(S1) + L(S2)) \quad (2)$$

To measure word-form similarity, HermeScan normalizes the Levenshtein Distance (or Edit Distance) [21] as shown in equation 2. The function L represents the length of the string. The Levenshtein Distance represents the minimized number of edits (delete, insert, replace a character at one time) to transform one string to another. A lower edit distance indicates higher string similarity. For instance, consider the keyword `hostname_1.1` in the front-end file, which sets the parameter of a device. It shares a similar form with the back-end string `hostname_%s` concatenated with the device ID. These pairs are shared keywords and could be identified through the fuzzy matching of word form similarity.

$$SemanticSim(S1,S2) = \begin{cases} Cosine(S1,S2), & Others \\ 0, & \frac{LCS(S1,S2)}{Min(L(S1),L(S2))} < \theta \end{cases} \quad (3)$$

For semantic similarity, HermeScan utilizes the Bidirectional Encoder Representations from Transformers (BERT) [8] to calculate the cosine similarity of string embeddings. As a pre-training model, BERT is widely used for various NLP tasks such as semantic search and question answering. In our approach, we utilize the efficient and performant all-MiniLM-L6-v2 model [37], trained on a diverse dataset of over 1 billion training pairs. To further make the efficiency scalable, semantic similarity calculation is enabled when the ratio of the longest common subsequence of two strings to the length of the shorter string surpasses the threshold θ. Semantic similarity further complements fuzzy matching, enabling associations to be established based on semantic inference. For example, the string `request from %s is banned for security` appears in the front end and the `sec_ip_ban` can be linked through semantic analysis, with the latter recognized as a common keyword in the binary.

Algorithm 1 Input identification algorithm

Input: $backend\_files(BFS), frontend\_files(FFS)$
Output: $source\_funcs$
1: $shared\_strings \leftarrow FuzzyMatch(BFS, FFS)$
2: $candidate\_funcs \leftarrow GetFuncs(shared\_strings)$
3: **for** each $func \in candidate\_funcs$ **do**
4: $params, ret \leftarrow GetCallingConvention(func)$
5: $caller\_func \leftarrow GetHostFunc(func)$
6: $in\_ddg \leftarrow DefUseAnalysis(func, params)$
7: $out\_ddg \leftarrow DefUseAnalysis(caller\_func, ret)$
8: **for** each $param \in params$ **do**
9: $taint\_sources \leftarrow ValueStored(param, in\_ddg)$
10: $taint\_constraints \leftarrow ConstraintInfer(func, param)$
11: **end for**
12: $taint\_sources \leftarrow ValueUsed(ret, out\_ddg)$
13: **if** $taint\_sources \neq Null$ **then**
14: $source\_funcs \leftarrow func, taint\_constraints$
15: **end if**
16: **end for**

Finally, HermeScan recognizes candidate functions that reference these strings, which are likely to parse external inputs and store them in the function's arguments (more specifically, the memory pointed by the arguments) or return value. Corresponding to the example in Listing 2, the words *type* and *event* are matching keywords, and the function **websGetVar** that refers to these keywords as parameters is considered a candidate function.

2) Candidate Function Checking: After recognizing candidate functions, we may assume all their arguments and return values as variables storing external inputs, i.e., marking them as *taint sources*. However, overestimating all parameters as taint sources introduces false positives, and marking return values that are not subsequently used as taint sources causes additional taint analysis overhead. Thus, we further conduct a candidate function checking to remove infeasible candidate functions and mark parts of the function arguments or return values as *taint sources*.

The general idea is to check whether the parameters would receive values from external input and whether the return values would be used by following operations. Specifically, as shown in Algorithm 1, inside each candidate function, we track the data flow of each parameter via def-use analysis. If the memory pointed by the parameter is stored with some values, it may serve as a taint source that receives external inputs (lines 8-9). Otherwise, this parameter is not a taint source. Outside the candidate function, we track the dataflow of its return value and examine whether it will be used by the following operations. If so, the return value will be marked as a taint source (line 12). Thus, in Figure 2, none of the parameters of the candidate function **websGetVar** is a *taint source* (Subgraph-4). The variables *event* and *type* that are used as the return value of the function **websGetVar**(line 5 and line 8 in Subgraph-3), are *taint sources*.

In practice, we found some taint sources have length restrictions. To further improve the precision of the analysis, we also employ a constraint inference to collect some constraints through conducting the def-use analysis and value set analysis (VSA) on the taint sources. Specifically, the constraint inference occurs when the following conditions are met: 1) the taint source is related to the pointer of the destination address in the string-copy functions that are summarized in Table I. 2) the parameter that is not the taint source is used as the

```
1  char *dlink_webGetVarN(char *var, webs wp, int len)
2  {
3      char *v1, *result;
4      sym *sp = symLookup(wp->cgiVars, var);
5      v1 = (char *)malloc(len);
6      strncpy(v1, sp->content.value.string, len-1);
7      result = v1;
8      *(result + len -1) = 0;
9      return result;
10 }
```

Listing 1: Pseudocode for illustrating constraint inference.

length limitation of memory-copy functions. Then we utilize VSA to establish the constraint between the limit length in the memory-copy function and the parameter of the candidate function and propagate the constraint from the summarized function back to the taint source in the candidate function. Our VSA is flow-sensitive and accounts for changes in dataflow facts made by each statement. Finally, a value flow graph (VFG) is built based on VSA, where each node represents the states that store the value range of the register and the memory.

We take Listing 1 to better illustrate our constraint inference approach. First, through def-use analysis, we could find the third parameter of the function **dlink_webGetVarN** flows into the third parameter in the summarized function **strncpy**, and the destination address of **strncpy** eventually flows to the return value of the caller function. Then we build a VFG for the candidate function through VSA. By obtaining the status of the VFG node located at the call site of **strncpy** (line 6), the third register value of the **strncpy** is expressed as uninitialized_inital_r2+0xffffffff, where the uninitialized_inital_r2 represents the parameter *len* and 0xffffffff refer to -1. Similarly, at the basic block of the exit statement (line 9), the register stored the return value of the function is expressed as v1_(len-1)+00, which means that the return value is equal to the value of variable v1, and the value constraint is a string with a length of len-1 concatenated with a truncation character. Thus, we can infer that the constraint of the taint source of **dlink_webGetVarN** is limited by the value of its third argument minus one.

Eventually, HermeScan heuristically defines the input taint value on the appropriate parameter or return register at the sources according to the vulnerability type and collected length constraint. Variables assigned initial values are propagated to other used variables along with subsequent data flow analysis and continuously updating the data dependency graph. Although it is theoretically more precise to explore along the path and solve for inputs that satisfy the constraints, symbolic execution faces difficulties in efficiency and accuracy. Our approach is faster and is adapted to subsequent customized RDA.

D. Efficient RDA Analysis

We design a unique taint tracking scheme (LCO Interprocedural Analysis) according to three principles and adopt a path merging strategy to alleviate the path explosion problem. These optimization methods constitute an efficient data flow analysis module that balances efficiency and accuracy.

1) LCO Inter-procedural Analysis: We follow three principles to perform efficient dataflow analysis on the user input:

Algorithm 2 On-demand Interprocedural Analysis

```
Input: func
Output: result
 1: taintflag ← False
 2: params ← GetParams(func)
 3: for each param ∈ params do
 4:     if IsTaint(param) then
 5:         taintflag ← True
 6:     end if
 7: end for
 8: if taintflag then
 9:     if IsImport(func) then
10:         if IsSummary(func) then
11:             result ← SummaryValue(func)
12:         else
13:             lib ← SearchLib(func)
14:             result ← StepIntoLib(lib, func)
15:         end if
16:     else
17:         result ← StepInto(func)
18:     end if
19: end if
```

lightweight, context-sensitive, and on-demand.

Lightweight principle. As discussed in section III, the dataflow analysis is quite faster than symbolic execution. HermeScan leverages RDA-based taint tracking instead of heavy symbolic execution-based taint tracking methods used in SaTC and Karonte.

Specifically, angr's RDA module provides the most basic intra-procedure analysis for HermeScan. It lifts assembly instructions to VEX IR and performs RDA on the built CFG with the classic worklist algorithm [23]. For the input variable of interest, an indirect Def-Use graph is generated, where each node represents a *Def* of the variable, and each edge indicates the variable is *Use* after a *Def*. In a specific implementation, the given variable is distinguished by five categories: temporary variables, global variables, stacks, heaps, and registers, resulting in the better marking of the definition or use of the variable at a certain address.

Context-sensitive principle. Dataflow analysis often involves function calls, therefore interprocedural analysis is necessary. Unfortunately, the method based on angr's out-of-the-box RDA cannot be applied to actual firmware analysis because it does not consider the context of inter-procedural calls.

Thus, HermeScan extends angr's intra-procedural RDA to inter-procedural and considers context information to enable fine-grained dataflow analysis. During function calls, the function parameters are marked as the definition of temporary variables. Their values are assigned to the definition obtained from the register variable or stack variable in the caller function, following the calling convention. Upon returning to the caller function, HermeScan merges the definition values of the five types of variables (temporary variables, global variables, stacks, heaps, and registers) with different return addresses and overwrites their original values in the caller function.

In addition, for aliasing issues, such as indirect calls, HermeScan calculates the value of the relevant register from the *Def-Use* graph based on contextual information and resolves possible jump addresses for further analysis. Our approach does not pursue complete and precise aliases like Emtaint [6], but rather a demand-driven analysis without additional overhead. The limitations of the way we handle aliases are discussed in Section VII.

TABLE I: Summary of the common Libc functions.

Operation	Function Name
String Copy	strcpy, strncpy, strlcpy, strcpy_chk, strcat, strncat,sprintf, snprintf, sprintf_chk, sscanf, strdup, vsnprintf, memcpy
String Index	strstr, strchr, strrchr
String to Data	atoi, atol, atoll, strtoll
String Split	strtok, strsep
Others	memset, strlen

On-demand principle. For the sake of efficiency, static analysis often adopts on-demand tracking during interprocedural analysis. Existing methods choose whether to track or not by judging whether the parameters of the function are tainted. However, there are still two deficiencies: one is that the method relying on symbolic execution to explore the path is limited by the storage capacity of the path state, which makes it difficult to track on demand in nested functions; the other is that when it involves calling external library functions, the existing method only summarizes the common Libc library functions and skips other library functions.

HermeScan proposes on-demand tracking to solve the above difficulties. First of all, our method judged whether a variable is tainted from the data-dependent graph updated at analysis without involving path storage; second, we step into library functions whose parameters are tainted to form a deeper data flow analysis. Our approach is described in Algorithm 2 in detail: HermeScan firstly prioritizes tracking explicit taint propagation by identifying functions with tainted parameters on the data-dependent graph (lines 1-8). Then for functions that are not imported by external libraries, HermeScan directly steps into interprocedural RDA (lines 16-17). Additionally, HermeScan summarizes the return values of commonly used Libc functions in Table I. In the context of RDA, a function's summary is its effect on the use-def value of various variables, which differs from those used in symbolic execution. Finally, if the function in question does not fall into these categories, HermeScan performs an RDA on the Lib-CFG of the corresponding library function (lines 13-14).

Example. Listing 2 shows an example of LCO interprocedural analysis. Firstly, in the **setup_mydlink_wizard** function, the data flow analysis module treats **getenv** as a source point and sets a value of an overly long string for v4 as a definition according to the rules for detecting buffer overflows (line 4). Then since the second parameter of the **updownservice** function comes from v4, a tainted register variable, LCO inter-procedural analysis chooses to step into **updownservice** for analysis (line 5). The **getresponsepage** and **postnvram** functions are not analyzed because their parameters are not tainted (lines 6-7). Similarly, the parameters of the function func2 in the **updownservice** are also tainted, and the LCO inter-procedural analysis goes further into the func2 (line 10). Finally, after encountering the calling of **strcpy** in func2, we apply the function summary of **strcpy**, which fills the defined value of the stack variable corresponding to the buf variable with the value pointed to by the pointer a1 (line 16).

2) Path merging strategy: HermeScan designs a path merging strategy to identify and merge paths repeatedly passed in the function call graph, thereby reducing redundant analysis

```
// ssi
void setup_wizard_mydlink(int a1){
    char *v4;
    v4 = getenv("sys_service");
    updown_services(0, v4);
    post2nvram(a1);
    response_page = get_response_page();
}
int updown_services(int mode, char *sys_service){
    if(mode) return func2(sys_service);
    return func3(sys_service);
}
int func2(char *a1){
    char buf[1028];
    if (a1 && *a1){
        strcpy(buf, a1);
    }
}
```

Listing 2: Pseudocode for illustrating LOC Inter-procedural Analysis, which is taken from CVE-2022-41451. The setup_wizard_mydlink function reads the environment variable of sys_service, which is finally passed to the strcpy function in func2, causing a stack overflow.

effort. The strategy leverages the path-insensitive feature of RDA and employs two schemes: **multi-source taint** and **multi-sink observation** based on control flow reachability.

Multi-source taint: For a function with multiple source points, existing firmware static analysis schemes [3], [26], [11], [5] start with each source point as a taint point and track its taint propagation process independently. In the context of RDA, we can track the use-definition assignment of all variables in the program. Thus HermeScan could taint each source point with a different label in the function that contains multiple input source points. In this way, HermeScan tracks and distinguishes the propagation process of multiple taint values in one RDA analysis, thereby reducing the cost of starting analysis by treating these source points as different starting positions.

Multi-sink observation: Since we use path-insensitive RDA, any function reachable from the call graph starting from a specified function is analyzed theoretically. Consequently, all functions containing sinks in the same call graph are covered in one RDA analysis. HermeScan avoids analyzing these sink points multiple times by setting several observation points in one analysis pass.

Figure 4 depicts how paths are merged under different strategies. Enabling HermeScan's path merging strategy allows one RDA pass to cover two source points in function A and three sink points in functions A, B, and D, reducing the number of analyzed paths from 7 to 2. In contrast, SaTC's strategy partially merges duplicated paths, reducing the number of analyzed paths from 7 to 3. The path from source point C1 to sink point D1 is retained to account for cases where our on-demand data flow tracking may not enter function C if its parameters are not tainted.

E. Taint Inspection Engine

The taint inspection engine can be regarded as a collection of vulnerability pattern policies, which define vulnerability-related sensitive functions and detection rules.

For the sensitive functions, we summarize functions related to 10 types of vulnerabilities (which is more than SaTC,

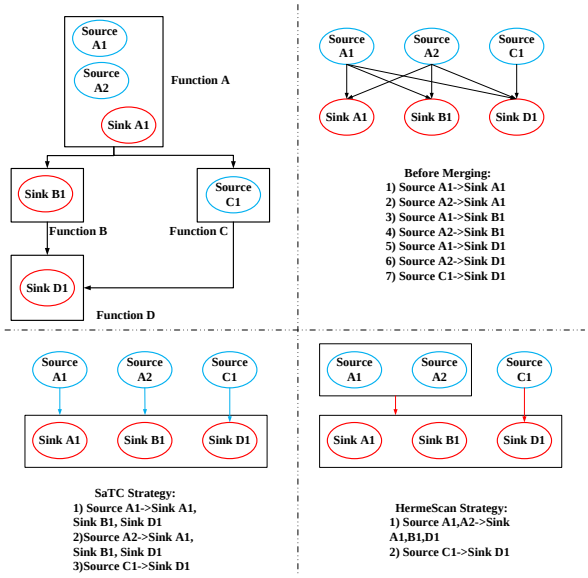

Fig. 4: Schematic diagram of path merging strategy.

KARONTE) in Table II by synthesizing the vulnerability patterns and the characteristics of the back-end programs in embedded devices. The vulnerability detection rules employed by HermeScan distinguish it from other approaches like SaTC and KARONTE. Unlike these methods, HermeScan does not focus on solving the input of collected path constraints. Instead, HermeScan observes all possible definition values of parameters within a reachable sensitive function and evaluates if these values meet the conditions for triggering a vulnerability. We take the buffer overflow and command line injection vulnerability as examples to illustrate how HermeScan detects vulnerabilities.

Buffer overflow: For detecting buffer overflow, the initial taint source value is a string of a certain length, which may be truncated or concatenated during dataflow analysis. Therefore, when it is passed to the parameter of the sensitive function, the length of the string may change. The alert is produced only if the length of the copied data exceeds the destination storage space.

Command line injection: For command-line injection vulnerabilities, alerts are generated based on the presence of sanitization of strings originating from taint sources. As a *may-analysis*, HermeScan examines all potential values of parameter variables within functions like system and popen. If the value of the string referenced by such a variable includes a string from the taint source and the information of the string is complete, we can infer that the system may execute a malicious command line command.

V. Implementation

We implemented the prototype system of HermeScan with around 4K lines of Python code. The CFG recovery module is based on IDA 7.6 [9] and angr 9.2.1 [30]. We extend IDA's automatic function identification with function prologue and export the function start address from IDA for angr to build CFG. For source input identification, we utilize the R package text2vec [27] to calculate the Levenshtein distance between words for fuzzy matching and implement candidate function

TABLE II: Sensitive functions used as sink points for different types of vulnerabilities

Vulnerability Type	Sink Functions
CWE-119	strcpy,strncpy,sprintf,snprintf,memcpy,strcat, strncat,sscanf,gets
CWE-78	execv,system,twsystem,cstesystem,dlopen,popen, dosystemcmd
CWE-134	printf,vsprintf
CWE-79	puts,printf
CWE-319	openurl, system
CWE-337	time, rand, srand
CWE-352	same as CWE-78
CWE-22	fopen, unlink
CWE-89	exec_sql,runsql,sqlite3_ex

checking with Python code. The value of hyperparameters α, β, and θ equal 0.75, 0.83, and 0.5, which are selected by a grid search to consider the trade-off of efficiency and effectiveness. The efficient data flow analysis is built on top of angr's RDA module. We extend this module from simple intra-procedural analysis to support context-sensitive inter-procedural analysis. Furthermore, we implement a path-merging strategy to alleviate the path explosion problem. In the taint inspection module, we define the sink functions of 10 kinds of vulnerabilities and the corresponding detection rules. The module is well extensible to support additional rules defined by the user.

VI. Evaluation

We conducted our evaluation to answer the following research questions:

- RQ1: How well does HermeScan find vulnerabilities on real-world devices? How effective is it compared to state-of-the-art tools?
- RQ2: How does the optimization of control flow recovery contribute to the vulnerability detection of HermeScan?
- RQ3: Can HermeScan's input source identification make the analysis more accurate? How does it work?
- RQ4: Can HermeScan's path merging strategy alleviate the path explosion problem?
- RQ5: How are HermeScan's control flow recovery and static analysis capabilities?

A. Experiment Setup

To evaluate the effectiveness, accuracy, and efficiency of our vulnerability discovery approach, we thoroughly assessed two datasets on an Ubuntu 21.04 LTS system equipped with a 4-core Intel CPU and 32 GB RAM. The datasets consist of the 0-day and N-day datasets and are all the firmware of Linux-based devices without obfuscation and encryption.

0-day dataset: To minimize the bias, We selected 30 latest firmware samples from 8 manufacturers, including LINKSYS, ASUS, Netgear, Tenda, TOTOLINK, TrendNet, D-Link, and TP-LINK, as our zero-day dataset. These firmware samples in Table III cover 19 series and were downloaded from the manufacturer's official website. Among the samples, 19 are WiFi-6 SoHo routers [30], which are the flagship products of the vendors. The average size of the samples is 28.2 megabytes, and they cover the following four architectures: ARM32, ARM64, MIPSEL, and MIPSEB.

N-day dataset: The N-day dataset consists of 98 older versions firmware, which is used for large-scale testing to complement the effectiveness of our method. We selected firmware samples covering 25 series from 9 popular IoT manufacturers. These 98 firmware contain are orthogonal to our zero-day dataset. The source of the dataset is divided into two parts, one is taken from the 85 samples disclosed in the papers of SaTC and KARONTE, and the other is taken from the 13 samples downloaded from the official website of the manufacturer. Table VI shows the serial information about the firmware sample.

SOTA solutions to compare with: We compared HermeScan with KARONTE [26] and SaTC [3], the state-of-the-art static analysis tools for detecting taint-style vulnerabilities in IoT devices. Both KARONTE and SaTC use symbolic execution to achieve their taint tracking. To ensure the fairness of the experiment, we tested two types of vulnerability detection supported by all three tools: **bof** (buffer overflow) and **cli** (command line injection).

Results validation: For each generated alert, we manually confirm whether there is a vulnerability based on the input information generated by the three tools. If an alert is identified as a vulnerability it is a true positive, otherwise, it is a false positive. In this paper, we marked the number of true positive alerts as **TP** and the number of false positive alerts as **FP**. Considering that multiple true positive alerts may be generated for the same vulnerability, we mark the number of unique vulnerabilities found as **Vul**.

B. Bug Finding (RQ1)

In this section, we first count the **number of vulnerabilities** discovered by HermeScan and then compare HermeScan with the SaTC and Karonte in terms of **effectiveness**, **accuracy**, and **efficiency** on two datasets.

1) Zero-day Dataset: **Number of vulnerabilities:** After excluding some duplicate vulnerabilities, we confirmed the vulnerabilities found by HermeScan in Table IV. HermeScan ultimately found 76 known vulnerabilities (we also discovered these known vulnerabilities for the first time but have been assigned to other researchers.) and 87 unknown vulnerabilities (69 are assigned CVE numbers, and the rest are pending), proving its ability to detect vulnerabilities on real-world firmware.

We further analyzed vulnerabilities found only by HermeScan and identified the underlying reasons. The reasons could be summarized in the following three points: First, HermeScan builds the CFG that identifies more functions containing the source and sink points. Second, HermeScan uses input source recognition to determine the source function instead of the shared keyword reference used by SaTC and Karonte, which reduces missing some source points. Third, HermeScan has set up detection rules for more types of vulnerabilities, which can detect other types of vulnerabilities except for bof and cli.

As a comparative SOTA, 5 of the 32 vulnerabilities found by SaTC are unknown vulnerabilities and 27 are known vulnerabilities. As shown in Figure 5-A, 29 of the 32 vulnerabilities can also be found by HermeScan, and 3 unique bof-type vulnerabilities can only be found by SaTC. The reason why HermeScan can not find these vulnerabilities is that angr does not correctly calculate the size of the stack space, resulting in the tainted data not covering the top of the stack when the strcpy-like function is called, thus no alert is generated.

TABLE III: **0-day dataset evaluation** results of HermeScan, SaTC, and KARONTE. The program name refers to the analyzed boundary binary. We counted the number of alerts generated by the tool (Alerts), the number of confirmed vulnerabilities (Vuls), and the tool execution time (Time). *Italics* in the time column represent unexpected exits or non-execution of the sample. **Bold** in the Vendor-Model column means HermeScan finds more vulnerabilities on this sample than other tools. The number of Vuls in () indicates the number of unique vulnerabilities after deduplication.

		HermeScan				SaTC			Karnote		
Vendor&Model	**Program Name**	**Alerts**	**Vuls (bof+ci)**	**Vuls (other)**	**Time**	**Alerts**	**Vuls (bof+cli)**	**Time**	**Alerts**	**Vuls (bof+cli)**	**Time**
LINKSYS MR7350	bluetoothd	0	0	0	3min	1	0	30min	0	0	2h22min
LINKSYS E9450	httpd	0	0	0	11min	0	0	24min	0	0	2h08min
LINKSYS EA4500	twonkymediaserver	1	**1**	0	17min	0	0	26h	0	0	1h21min
ASUS GT-AX6000	httpd	0	0	0	9min	5	0	27h28min	0	0	4h31min
ASUS GT-AC2900	cfg-server	0	0	0	12min	2	0	23h58min	0	0	*7min*
ASUS RT-AX56U	httpd	4	**4**	0	13min	5	0	26h21min	0	0	48min
Tenda AX-12	httpd	9	**6**	3	1h20min	0	0	*12min*	0	0	3h36min
Tenda AX-3	httpd	17	**11**	0	2h23min	27	6	36h	0	0	1h24min
Tenda AX-1803	thttpd	20	**12**	2	2h03min	38	8	16h28min	0	0	*5min*
Tenda AX-1806	thttpd	20	**14**	0	2h11min	44	13	18h53min	0	0	42min
Tenda W15E	httpd	17	**15**	2	2h39min	50	5	21h11min	0	0	1h22min
TOTOLINK T8	cstecgi	14	**4**	0	14min	0	0	*3min*	0	0	*2min*
TOTOLINK LR350	cstecgi	24	**9**	0	13min	0	0	47min	0	0	24min
TOTOLINK A7000	cstecgi	18	**12**	0	12min	0	0	4h09min	0	0	39min
TOTOLINK A8000	cstecgi	29	**13**	0	13min	2	0	39min	0	0	6h04min
D-LINK COVR-1201	prog.cgi	9	**4**	2	5h27min	0	0	*10min*	0	0	4h42min
D-LINK COVR-1210	prog.cgi	8	**4**	2	5h16min	0	0	*10min*	0	0	4h28min
Netgear RAX-10	net-cgi	6	0	0	19min	0	0	49min	0	0	2h20min
Netgear RAX-30	ntgr_ra_iot	0	0	0	6min	8	0	5h54min	0	0	3h10min
Netgear RAX-120	net-cgi	1	0	0	37min	0	0	1h09min	0	0	72h
Netgear MR-62	httpd	1	**1**	0	51min	0	0	23h56min	4	0	2h56min
Trendnet twe 829	samba_multicall	1	**1**	0	30min	0	0	*2min*	0	0	1h57min
Trendnet tew 823	ssi	39	**17**	0	9min	0	0	*11min*	0	0	1h54min
Trendnet tew 827	ssi	31	**18**	2	18min	0	0	*27min*	0	0	59min
Trendnet tew 818	rc	12	**5**	0	14min	0	0	18h32min	0	0	2h14min
Trendnet tew 752	cgibin	5	**1**	0	9min	0	0	20min	0	0	2h10min
TP-LINK AX3000	fapi_wlan_cli	0	0	0	*0min*	0	0	*15min*	0	0	2h42min
TP-LINK XDR1850	dms	2	0	0	18min	0	0	*3min*	0	0	*2min*
TP-LINK XDR3060	dms	3	**2**	0	1h44min	0	0	*2min*	0	0	28min
TP-LINK XTR7880	dms	6	**2**	0	2h10min	0	0	*3min*	0	0	19min
Total	/	297	**156(152)**	13(11)	30h4min	182	32(32)	252h19min	4	0	127h58min
Average	/	9.9	5.2	/	1h7min	6.66	1.9	8h25min	0.13	0	4h16min

TABLE IV: Known and Unknown vulnerabilities found by HermeScan on the zero-day dataset.

Vendor & Model	**Number**	**Known Vulnerabilities**	**Unknown Vulnerabilities**
LINKSYS EA4500	1	/	1 unassigned
ASUS RT-AX56U	4	/	CVE-2022-46039 to CVE-2022-46042
Tenda AX-12	9	CVE-2022-45995, CVE-2022-45979, CVE-2022-45980, CVE-2022-27375, CVE-2022-27374	CVE-2022-37292,CVE-2022-28082, CVE-2021-45392, CVE-2021-45391
Tenda AX-3	11	CVE-2023-27042, CVE-2022-27239, CVE-2022-24995 ,CVE-2022-24163, CVE-2022-24162, CVE-2022-24160, CVE-2022-24158, CVE-2022-24142, CVE-2022-24145 to CVE-2022-20147	/
Tenda AX-1803	14	CVE-2022-37817 to CVE-2022-37824, CVE-2022-34595, CVE-2022-34596, CVE-2022-42086, CVE-2022-42087	/
Tenda AX-1806	14	CVE-2022-32071 to CVE-2022-32073, CVE-2022-25546 to CVE-2022-25555, CVE-2022-32069	/
Tenda W15E	17	/	CVE-2022-40448 to CVE-2022-40455, CVE-2022-40457 to CVE-2022-40461, CVE-2022-40462 to CVE-2022-40466
TOTOLINK T8	4	/	CVE-2021-46373 to CVE-2021-46376
TOTOLINK LR350	9	/	CVE-2022-44249 to CVE-2022-44252, 5 unassigned
TOTOLINK A7000	12	CVE-2022-37076 to CVE-2022-37084, CVE-2022-27003 to CVE-2022-27005	/
TOTOLINK A8000	13	/	CVE-2022-44328 to CVE-2022-44340
D-LINK COVR-1201	6	/	CVE-2022-42155 to CVE-2022-42161
Netgear MR-62	1	/	1 unassigned
Trendnet TEW-829	1	/	1 unassigned
Trendnet TEW-823	17	/	CVE-2022-41449 to CVE-2022-41459 CVE-2022-41461 to CVE-2022-41466
Trendnet TEW-827	20	CVE-2019-13276 to CVE-2019-13279, CVE-2021-14074 to CVE-2021-14081, CVE-2019-13148 to CVE-2019-13155,	/
Trendnet TEW-818	5	/	5 unassigned
Trendnet TEW-752	1	/	1 unassigned
TP-LINK XDR3060	2	/	2 unassigned
TP-LINK XTR7880	2	/	2 unassigned
Total	163	76	87

Effectiveness: The result in Table III shows the effectiveness of HermeScan. In detecting bof and cli vulnerabilities, HermeScan reports 120 (=152-32) more vulnerabilities than SaTC, and 152 (=152-0) more vulnerabilities than KARONTE. HermeScan outperforms SaTC in detecting vulnerabilities on 22 samples (as indicated in **bold**). In addition, HermeScan also found 11 other types of vulnerabilities, including CWE-337 and CSRF.

Accuracy: To illustrate the accuracy of the analysis, we counted the true positives (TP) and false positives (FP) for alerts produced by SaTC and HermeScan. As shown in Table V, HermeScan outperforms SaTC in TPR by 39%. The reason for the gap in TPR between the two tools is that SaTC ignores the input value constraints at the source points, while HermeScan checks the source function to provide reasonable input values. For example, the function websGetVarN("PPPOE_USERNAME", 32) reads the value with the key "PPPOE_USERNAME" from the local form. SaTC treats the string value returned by this function as an unconstrained symbolic variable, leading to inaccuracy in subsequent analysis. In contrast, HermeScan summarizes the function and determines that the value returned from this function has a 32-byte length constraint.

TABLE V: 0-day dataset evaluation results of HermeScan, SaTC, and KARONTE in terms of accuracy. HermeScan(Dis-FM) refers to disabling fuzzy matching, and HermeScan (Dis-IC) refers to disabling input function checking. TPR stands for true positive rate, and FPR stands for false positive rate.

COMPARE	**ALERT**	**TP**	**FP**	**TPR**	**FPR**
HermeScan	297	241	56	0.81	0.19
HermeScan(Dis-FM)	228	178	50	0.78	0.22
HermeScan(Dis-IC)	386	241	135	0.62	0.38
SaTC	182	77	105	0.42	0.58
KARONTE	4	0	4	0	1.00

Efficiency: In terms of execution time, HermeScan has a great advantage in the efficiency of analysis. Table III shows that the average execution time of HermeScan on each sample is one hour and seven minutes, which is 7.5x times faster than SaTC and 3.8x times faster than KARONTE. The time cost of taint analysis is positively related to the number of paths to be detected, in other words, it is also related to the number of source points and sink points. Compared with KARONTE, SaTC locates more source points through the keywords shared by the front and back ends, which generates

TABLE VI: *N-day dataset evaluation results* of HermeScan, SaTC, and KARONTE. For each vendor, we report the device series, the number of firmware samples, the total number of alerts (ALERT), the total number of true positives (TP), the average true positive rate (TPR), and the total analysis time (TIME: hour).

Vendor	Device Series	Samples	HermeScan				SaTC				KARONTE			
			ALERT	TP	TPR	TIME	ALERT	TP	TPR	TIME	ALERT	TP	TPR	TIME
NETGEAR	R/XR/WNR/AC	31	668	505	0.76	56.6	582	299	0.51	540.8	35	26	0.74	137.3
D-Link	DIR/DWR/DCS/COVR	17	36	29	0.81	12.8	20	14	0.70	20.9	16	13	0.81	39.1
TP-Link	TD/WA/WR/TX/KC	18	6	2	0.33	5.9	8	3	0.38	79.1	5	2	0.40	16.8
Tenda	AC/WH/FH/AX	20	289	251	0.87	34.2	262	229	0.87	227.9	23	15	0.65	7.9
TOTOLINK	T/A/LR	6	64	52	0.81	1.6	6	4	0.67	33	0	0	/	1.3
MOTOROLA	C1/M2	2	8	8	1	0.4	0	0	/	0.33	0	0	/	1.1
AXIS	P/Q	2	0	0	/	0.5	0	0	0	4.1	0	0	/	0
Others	/	2	0	0	/	0	0	0	/	0	4	4	1.00	1.6
Total	/	98	1071	847	/	112.0	878	549	/	906.1	83	60	/	205.1
Average	/	/	10.93	8.64	0.79	1.14	8.96	5.60	0.63	9.25	0.85	0.61	0.72	2.09

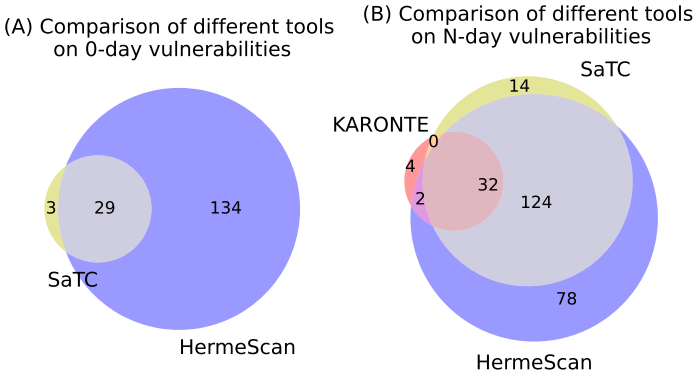

Fig. 5: Number of vulnerabilities found by HermeScan and baselines on the 0-day, N-day vulnerability dataset.

more paths that require taint tracking. However, the overheads of using symbolic execution for taint tracking on each path for input variables are enormous, which also results in high execution time for SaTC and KARONTE. By contrast, on the premise of ensuring sufficient source points to explore, HermeScan uses lightweight, on-demand data flow analysis to achieve taint tracking.

2) N-day Dataset: **Effectiveness:** We measure the efficiency of the three tools by counting the number of known vulnerabilities discovered on the N-day dataset. Figure 5-B shows that HermeScan found 204 known vulnerabilities, 66 (=204-138) more than SaTC and 164 (=204-40) more than KARONTE. The overall results prove that HermeScan also has advantages in finding N-day vulnerabilities.

We also compare and analyze the unique N-day vulnerabilities found by the three tools. The finding of an additional 78 vulnerabilities by HermeScan benefits from two aspects: one is that the more comprehensive CFG covers more source and sink points; the other is that the automatic input point recognition helps it identify more source points. The additional 14 vulnerabilities discovered by SaTC benefit from the precise symbolic execution solution that can construct inputs that meet certain constraints to trigger these vulnerabilities. The four additional vulnerabilities found by KARONTE come from a Huawei motherboard device, and neither SaTC nor HermeScan can find a valid source point because there are no front-end files in the firmware.

Accuracy: We counted the TP of the three tools on the N-day dataset and calculated the TPR. As shown in Table VI, The TPR of HermeScan is 79%, which is the highest among the three tools, compared with 16% higher than SaTC and 7% higher than KARONTE. Although the results of HermeScan and KARONTE are similar in terms of average TPR, the average TP found by HermeScan is much higher than that of KARONTE. On the other hand, HermeScan is not only better than SaTC in TPR but also has more true alerts than SaTC. The results of large-scale data sets prove that HermeScan can effectively reduce false positives by accurately identifying input points and assigning reasonable taint values.

Efficiency: On a large-scale sample dataset, HermeScan takes an average of 1.14 hours to analyze a firmware sample, which is about half of the time required by KARONTE and one-eighth of the time required by SaTC. The results once again confirm our intuition that RDA-based taint tracking is more efficient than symbolic execution-based taint tracking.

Conclusion: HermeScan can detect taint-style vulnerabilities in firmware and outperform existing methods in effectiveness, accuracy, and efficiency. HermeScan takes an average of one hour and seven minutes to process each sample and find 87 zero-day vulnerabilities with 81% true positives on the 0-day dataset.

C. *Effectiveness of taint tracking on enhanced CFG (RQ2)*

We evaluated HermeScan on six samples containing more than four unknown vulnerabilities with different configurations to figure out the contribution of each control flow optimization. The number of vulnerabilities is the metric to measure the contribution. The five configuration cases are annotated as follows:

- **HermeScan-Plain.** This configuration only uses HermeScan's dataflow analysis and taint engine on CFG constructed by angr.
- **HermeScan-En-B.** This configuration enables the optimization of function boundary identification.
- **HermeScan-En-B&S.** Not only does this enable recognition of function boundaries, but this configuration also recovers the symbol names of the functions.
- **HermeScan-En-B&S&C.** In addition to considering the CFG of the shared link library, the other enhancement optimizations are enabled.
- **HermeScan.** In this mode, all control flow optimizations are enabled, which is also the configuration used by HermeScan in Section VI-B.

Figure 6 shows that HermeScan performs best when enabling all control flow enhancement measures. The optimization of HermeScan-En-B, HermeScan-En-B&S, and

HermeScan-En-B&S&C has its contribution to the detection of vulnerabilities.

We further analyze in detail how these optimizations help HermeScan. The reason why HermeScan-En-B can find more vulnerabilities is that it identifies more function boundaries in the program, and these functions may contain potential source and sink points. The recovery of symbolic names also effectively expands the analysis scope of HermeScan. Because the taint engine of HermeScan uses function names to locate addresses of the sink points. In D-Link COVR 1203, the recovery of function names helps HermeScan-En-B&S to find all vulnerabilities. The recovery of the calling convention assists the LOC inter-procedural analysis. In Tenda W15 and TrendNet TEW-823, with the help of recovering calling conventions by HermeScan-En-B&S&C, the dataflow of the tainted input could be correctly passed to the callee function including sink points. Considering the control flow connectivity between the main program and the library inherently expands the target range of static analysis. In the cases of ASUS RT-AX56u and TOTOLINK T8, a total of 4 vulnerabilities are found in their private libraries.

Conclusion: The optimization methods used to enhance control flow all extend the scope of static analysis from different aspects and improve the vulnerability detection ability of HermeScan.

D. Effectiveness of Input Source Identification (RQ3)

In this section, we take HermeScan as the baseline and compare the true positives and true positive rate of HermeScan (Dis-FM) with fuzzy matching strategy disabled and HermeScan (Dis-IC) with input function checking strategy disabled.

As shown in Table V, HermeScan can not only guarantee the highest TPR (81%), but also generate the most TPs (241) after enabling the two strategies. The reasons are analyzed as follows:

When the fuzzy matching strategy is turned off, HermeScan (Dis-FM) misses about 63 TPs, and its TPR is close to that of the original HermeScan. Enabling fuzzy matching does introduce some false positives, but the benefit is minimal compared to the additional true positives. The results in Table VII show that our fuzzy matching strategy can find an additional 27% of keywords. While HermeScan (Dis-FM) only uses the

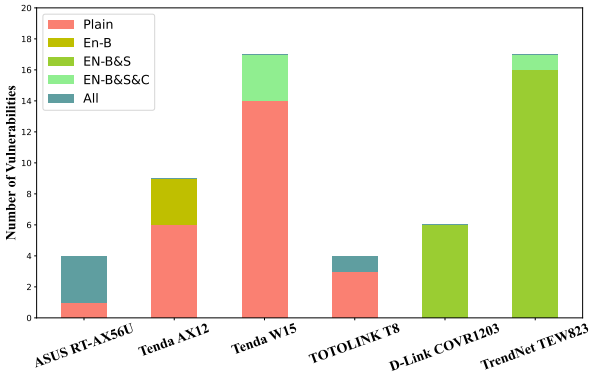

Fig. 6: Contribution of various control flow recovery optimization methods to vulnerability detection.

TABLE VII: The number of shared keywords, filter functions, and constrained functions identified by source input identification. The (S) and (H) annotations represent the SaTC's matching and HermeScan's matching.

Vendor & Model	Shared Keywords(S)	Shared Keywords(H)	Increased Proportion	Constrained Source Functions
LINKSYS MR7350	47	52	10.64%	1
LINKSYS E9450	56	57	1.79%	1
LINKSYS EA4500	65	66	1.54%	1
ASUS GT-AX6000	180	187	3.89%	2
ASUS GT-AC2900	180	184	2.22%	2
ASUS RT-AX56U	404	504	24.75%	2
Tenda AX-12	201	222	10.45%	1
Tenda AX-3	246	253	2.85%	1
Tenda AX-1803	254	255	0.39%	1
Tenda AX-1806	262	269	2.67%	1
Tenda W15E	437	535	22.43%	1
TOTOLINK T8	67	69	2.99%	1
TOTOLINK LR350	66	67	1.52%	1
TOTOLINK A7000	79	80	1.27%	1
TOTOLINK A8000	107	116	8.41%	1
D-LINK COVR-1201	506	625	23.52%	3
D-LINK COVR-1210	495	618	24.85%	3
Netgear RAX-10	860	897	4.30%	1
Netgear RAX-30	107	237	121.50%	1
Netgear RAX-120	866	1005	16.05%	1
Netgear MR-62	862	870	0.93%	0
Trendnet TEW-829	35	35	0.00%	0
Trendnet TEW-823	1042	1459	40.02%	1
Trendnet TEW-827	103	365	254.37%	1
Trendnet TEW-818	176	249	41.48%	1
Trendnet TEW-752	36	36	0.00%	0
TP-LINK AX3000	237	238	0.42%	0
TP-LINK XDR1850	99	188	89.90%	1
TP-LINK XDR3060	96	117	21.88%	1
TP-LINK XTR7880	108	213	97.22%	1
Average	276	336	27.81%	1

exact matching keyword reference address as the source point, resulting in some false positives.

When the source function checking is turned off, HermeScan (Dis-IC) mistakenly regards all functions referenced by shared strings as source functions. As shown in Table VII, 26 of the 30 samples involved at least one function with input constraints. HermeScan (Dis-IC) over-taints the parameters of these functions without restriction, resulting in an 18% increase in false positives. For example, in the case of Tenda W15, HermeScan (Dis-IC) mistakenly sourced the function cJSON_AddItemToObject and tainted its parameters. In fact, the behavior of this function is to add an item to the JSON structure, rather than a reasonable source input.

Conclusion: The fuzzy matching strategy and candidate source function checking can effectively help HermeScan reduce false negatives and false negatives, making its analysis more accurate. Input source identification helps to reduce the false positives by 18% on the zero-day dataset.

E. Effectiveness of Path Merging Strategy (RQ4)

As introduced in Section IV-D2, HermeScan uses a path merging strategy to alleviate the path explosion problem. In this section, we verify the effectiveness of path merging by comparing the number of analyzed paths before and after adopting the strategy.

Table VIII shows the proportion of the reduction in the number of paths resulting from the merger strategy. Overall, the path merging strategy reduced paths by an average of 89.4%. Among them, 22 out of 30 samples merged more than 90% of the paths. In the TrendNet TEW-823, HermeScan reduced up to about 160,000 duplicate paths, greatly decreasing the time of data flow analyses. After investigating its boundary program *ssi*, we found that there are 2346 call sites of the sensitive source functions and 6147 call sites of sensitive sink functions, and 167,554 paths between them are reachable by control flow. The path merging strategy of HermeScan regards multiple source points in the same caller function as the same starting position and includes all sink points on the reachable

TABLE VIII: Number of paths for taint tracking. The (bf) and (af) annotations represent the before and after path merging strategy.

Vendor & Model	Paths(BF)	Paths(AF)	Decreased Proportion
LINKSYS MR7350	69	23	66.67%
LINKSYS E9450	472	121	74.36%
LINKSYS EA4500	1838	143	92.22%
ASUS GT-AX6000	1010	63	93.76%
ASUS GT-AC2900	1062	88	91.71%
ASUS RT-AX56U	636	129	79.72%
Tenda AX-12	1673	72	95.70%
Tenda AX-3	9413	96	98.98%
Tenda AX-1803	5112	109	97.87%
Tenda AX-1806	4789	101	97.89%
Tenda W15E	11113	186	98.33%
TOTOLINK T8	7126	101	98.58%
TOTOLINK LR350	12688	75	99.41%
TOTOLINK A7000	13944	77	99.45%
TOTOLINK A8000	610	105	82.79%
D-LINK COVR-1203	686	19	97.23%
D-LINK COVR-1210	644	18	97.20%
Netgear RAX-10	1299	57	95.61%
Netgear RAX-30	171	11	93.57%
Netgear RAX-120	910	345	62.09%
Netgear MR-62	2008	214	89.34%
Trendnet TEW-829	231	32	86.15%
Trendnet TEW-823	167554	265	99.84%
Trendnet TEW-827	12160	116	99.05%
Trendnet TEW-818	20014	219	98.91%
Trendnet TEW-752	594	9	98.48%
TP-LINK AX3000	0	0	0.00%
TP-LINK XDR1850	2294	17	99.26%
TP-LINK XDR3060	2747	18	99.34%
TP-LINK XTR7880	2689	18	99.33%
Average	9518	95	89.40%

TABLE IX: The results on function information recovery and static analysis statistics. The (bf) and (af) annotations represent before and after optimization. The Function Call Depth is the maximum value of statistics.

Vendor & Model	Function(bf)	Function(af)	Symbol(bf)	Symbol(af)	Library	Function Call Depth	Function Coverage
LINKSYS MR7350	3001	3363(+362)	365	365	7	3	7.23%
LINKSYS E9450	1340	1463(+123)	468	471(+3)	39	7	9.77%
LINKSYS EA4500	3444	3617(+173)	0	144(+144)	3	4	8.49%
ASUS GT-AX6000	611	929(+318)	399	399	17	6	21.85%
ASUS GT-AC2900	878	934(+56)	235	235	17	7	26.02%
ASUS RT-AX56U	1218	1452(+234)	385	387(+2)	23	6	14.74%
Tenda AX-12	1363	1462(+99)	220	280(+60)	11	3	20.79%
Tenda AX-3	1507	1508(+1)	227	228(+1)	10	3	26.99%
Tenda AX-1803	6661	7527(+866)	294	294	35	4	4.26%
Tenda AX-1806	5053	6505(+1452)	290	290	34	4	4.52%
Tenda W15E	1621	1892(+271)	271	271	11	4	33.83%
TOTOLINK T8	370	528(+158)	106	123(+17)	9	3	14.58%
TOTOLINK LR350	313	434(+121)	0	281(+281)	9	2	27.88%
TOTOLINK A7000	309	428(+118)	0	128	7	3	28.50%
TOTOLINK A8000	398	585(+187)	155	155	6	3	27.01%
D-LINK COVR-1201	1203	1367(+164)	1	202(+201)	11	6	30.72%
D-LINK COVR-1210	1116	1278(+162)	1	162(+161)	11	6	31.06%
Netgear RAX-10	2052	2186(+134)	207	208(+1)	16	5	37.74%
Netgear RAX-30	1501	1711(+210)	213	213	41	6	4.73%
Netgear RAX-120	2032	2120(+88)	218	219(+1)	14	5	32.55%
Netgear MR-62	2728	2947(+219)	565	567(+2)	56	3	29.32%
Trendnet TEW-829	11750	12671(+921)	52	296(244)	5	3	0.92%
Trendnet TEW-823	908	1095(187)	0	126(+126)	11	2	32.05%
Trendnet TEW-827	1391	1399(+8)	223	251	13	2	24.59%
Trendnet TEW-818	492	536(+44)	170	170	7	3	27.61%
Trendnet TEW-752	297	317(+20)	0	109(+109)	3	3	17.67%
TP-LINK AX3000	1121	1121	547	559(+12)	7	0	0.00%
TP-LINK XDR1850	5783	5783	240	242(+2)	9	4	23.79%
TP-LINK XDR3060	6704	6762(+58)	292	292	8	4	29.15%
TP-LINK XTR7880	6919	6985(+66)	292	292	8	4	19.86%

path in CFG by one analysis. Eventually, the number of paths used for dataflow analysis is reduced to only 265. In TP-LINK AX3000, HermeScan did not discover one potential path even before the paths merged. The reason is the call site to the sensitive sink function is not detected in the boundary binary *fapi_wlan_cli*.

Conclusion: The path merging strategy reduces the number of paths by 89.4% on average. Avoiding the analysis of repeated paths can reduce the analysis cost and alleviate the problem of path explosion to a certain extent.

F. Performance of Control Flow Information Recovery and Static Analysis (RQ5)

1) Results of Control Flow Information Recovery: To demonstrate the performance of HermeScan in control flow information recovery, we compare the number of function identifications and the number of symbol names before and after taking optimization measures. Besides, we count the number of libraries that boundary binaries depend on in our dataset.

As shown in Table IX, after control flow recovery optimization, the number of functions identified is improved on 28 samples, and the number of symbol name recovery is increased on 19 samples. In particular, in the cases of TEW-752, TEW-823, COVR-1210, COVR-1203, LR350, A7000, and EA4500, angr recognizes few symbol names of the external library functions, while HermeScan successfully recovers symbol names of functions from the *LOAD* segment by parsing the ELF in the way of link view. Meanwhile, we found that the number of libraries that each boundary binary depends on varies from 3 to 56, many of which are privately implemented.

2) Results of Static Analysis: We use the maximum depth of the function call and function coverage to measure the performance of HermeScan in Table IX. In terms of the function call depth, the analysis of HermeScan involves more than 2 layers of function calls on samples except for TP-LINK AX-3000, which not only reflects that multi-layer function calls are common in the dataset, but also proves that our inter-procedural analysis is feasible. In terms of function coverage, on average 26% of functions are covered by HermeScan, indicating that our analysis applies to real firmware programs.

Conclusion: The results prove that the control flow information recovery of HermeScan is effective and HermeScan is suitable for real-world firmware binaries.

VII. Discussion

We highlight the boundaries and limitations of HermeScan's capabilities and propose possible improvements.

The false positive of the HermeScan: Despite imposing constraints on taint sources and providing sanitizers based on functions, HermeScan still faces false positives. The reasons come from three aspects: the first is the constraints on the path between the real entry point of external input data (such as the recv function) and the source function in the program. For instance, user-driven sanitizer filtering of some syntax in the program leads to stronger constraints at the source point. Second, the approach we infer taint source constraints is limited by the capabilities of VSA, and overestimation constraints cause certain false positives. Third, our sanitizer could not handle explicit checks in the path condition and may introduce false positives. To address these issues, we recommend combining partial simulation execution [24], [33] to dynamically observe the data flow that involves complex constraints, which can compensate for the shortcomings of pure static methods.

The false negative of the HermeScan: In HermeScan, the sinks are derived from function calls, and the pointer aliasing is demand-driven, which brings some false negatives to the analysis. For the former, our LCO-RDA is a general approach to identify how inputs flow to dangerous addresses. It can be extended to detect more sink points (such as array out of bounds) in the future. For the latter, it is well known that alias analysis in binary is an open and challenging problem. We suggest more precise pointer analysis could benefit from Emtaint's [6] SSE-based alias analysis and AI-assisted indirect call identification [44].

Targets of taint tracking: Currently HermeScan can track the data flow in binary files but fails to process the data flow transfer between programs written in different languages. In some devices, data from the user is processed across language files, such as binary files changing some configuration files that are then parsed by JS scripts on the Web front-end. It is an open problem to lift the files that are written in multi-language to a unified IR and analyze them, which can further expand the scope of HermeScan's application.

VIII. Related work

A. Static Taint Analysis

In recent years, the targets of static taint analysis work can be divided into two categories: one is interpreted language files or script files with source code, and the other is binary files without source code. The novelty of HermeScan compared with existing work is summarized in Table X and details are explained as follows:

Taint source identification: Most works on taint source identification rely on expert knowledge, either obtained by parsing the documentation of the target [19], [1], [26] or directly manually specified [43], [6], [5], [31], [17]. To our knowledge, HermeScan is the first work that combines AI technology [8] with fuzzy matching keywords to automatically identify taint sources.

Taint tag management: In taint analysis of interpreted language files, the less attention is paid to the value constraints on the taint tags. Only ARGUS [19] implements a taint summary database to assign values to each unique Action or reusable Workflow. In SOTA of firmware analysis [3], [26], the purpose of taint analysis is to guide symbolic execution, so they also use a simple tag to label the taint data. In contrast, HermeScan adds constraint checks to taint tags to reduce false positives, especially for detecting buffer overflow vulnerabilities.

Taint propagation: Works such as FlowDroid [1], ARGUS [19], Tchecker [17], etc. can perform flow-sensitive or even field-sensitive data flow analysis when the source code is available. However, the cost of doing flow-sensitive analysis in binary is the introduction of symbolic execution. For example, Emtaint [6] uses a structured symbolic expression for alias analysis, but complex variable propagation leads to difficulties in symbol storage. Similarly, the path exploration capabilities of SaTC [3] and KARONTE [26] are limited by symbolic execution, resulting in false negatives. HermeScan's philosophy is to be context-sensitive and track data flows on demand, which is proven more effective in the evaluation.

Path merging: Path merging or other methods that replace complete path analysis are important methods of taint analysis. Splendor [31] extracts the overlapping paths of statements analyzed forward from the source points and statements analyzed backward from the sink points. HermeScan designs a path merging strategy based on multi-source and multi-sink, which is more efficient than SaTC.

B. Binary Static Analysis

Many works [7], [36], [25] use symbolic execution to detect vulnerabilities in specific architectures. The scope of these workings is often focused on a certain class of vulnerabilities, and the efficiency is limited by symbolic execution.

TABLE X: Comparison of static taint analysis works in the four dimensions of taint source identification, taint assignment, taint propagation, and path merging.

	Target	Taint Source Identification	Tains Sink Identification	Taint Tag Management	Taint Propogation Strategy	Pathes Merging
Zexin Z	Microservices	Manually specified	based on sensitive API	Tag	Filed-sensitive, on-demand	/
ARGUS	Github workflow	Modeling based on parsing Github document	based on sensitive statment	Tag with Summary	Flow-sensitive, on-demand	/
Splendor	Webs Applications	Manually specified	based on sensitive function	Tag	Flow-sensitive, on-demand	Based on pathes matching
FlowDroid	APP	Modeling based on parsing APK config files	based on sensitive API	Tag	Flow,context, field-sensitive,on-demand	/
TChecker	Webs Applications	Manually specified	based on sensitive function	Tag	Flow,context-sensitive, on-demand	/
DTaint	Binary	Manually specified	based on sensitive function	Tag	Context-insensitive	/
KARONTE	Binary	Modeling based on finding IPC	based on sensitive function	Tag	Flow-sensitive(SE), on-demand	/
SATC	Binary	Modeling based on precise matching	based on sensitive function	Tag	Flow-sensitive(SE), on-demand	Based on multi sinks
Emtaint	Binary	Manually specified	based on sensitive function	Tag	Flow-sensitive(SSE), on-demand	/
HermeScan	Binary	**Modeling based on fuzzy matching**	based on sensitive function	**Tag with constraint**	**Context-sensitive(RDA), on-demand**	**Based on multi sources&sinks**

In recent years, some static analysis methods combined with dynamic characteristics have made some progress. Saluki [11] collects the execution data flow of a specific source by simulating the execution code fragment, and judges whether it violates the security property according to the solver designed by itself. Arbiter [34] has designed a hybrid method that combines static analysis and dynamic symbol execution with both accuracy and efficiency and has excellent performance on a large-scale evaluation of x86-64 programs. Although these methods require a certain amount of manual effort to write detection rules and specify input sources, they inspire us that static analysis can play a more efficient role than symbolic execution in the stage of collecting data dependencies.

C. Static Analysis in Firmware

Since IoT firmware is often closed source, the white-box auditing [28] and grey-box fuzzing like coverage-guided fuzzing [16] can not be directly applied to the firmware. In contrast, static analysis has fewer application preconditions that do not depend on a complicated execution environment. Thus, several works utilize static analysis to detect vulnerabilities in firmware [5], [26], [3], [4], [29].

For detecting taint-style vulnerabilities, Dtaint [5] is the first work to utilize the dataflow analysis to track the flow of insecure input in firmware. However, the CFG built by Dtaint is not complete, and the pointer alias analysis method it proposes to resolve indirect calls is limited. Emtaint [6] is a recent work to resolve the indirect call by SSE-based alias analysis. However, its alias analysis is limited by the ability of symbol storage, resulting in false negatives in taint analysis starting from the network receiving function. KaiCheng et al. [4] propose a static taint analysis framework, which infers taint sources by identifying functions with key-value features. The model of using the key-value function cannot cover many source points and is likely to miss potential vulnerabilities. KARONTE [26] models the interactions between multiple binaries and tracks the data flow between them, aiming to find vulnerabilities that exist in insecure interactions. SaTC [3] finds that developers usually use shared keywords between front-end files and back-end binaries, which can be used to locate the source points by these keywords. Both SaTC and KARONTE's taint tracking is based on symbolic execution, suffering from high overhead and lack of accuracy.

In addition, some works are using static analysis methods to detect other weaknesses of the firmware. Firmalice [29] is a framework aimed at finding authentication bypass flaws in firmware based on symbolic execution and program slicing. However, the authentication bypass model is limited and the computing overloads are overwhelming. CryptoREX [41] is designed to identify cryptography misuse in IoT using taint analysis. Though CryptoREX performs taint analysis across multiple binaries, its analysis is still insufficient due to ignoring vendor-specific libraries.

IX. Conclusion

In this paper, we present a lightweight reaching definition analysis (RDA) solution HermeScan to perform taint analysis on IoT firmware binaries, which is able to find taint-style vulnerabilities in IoT firmware with fewer false negatives, false positives, and time costs. Specifically, we point out that existing solutions have three drawbacks or challenges, which limit their performance. First of all, existing solutions overlooked the incomplete control flow graph issue, which brings non-negligible false negatives. Second, existing solutions did shorten the paths to analyze to get fewer false positives, but have limited precision. Third, existing solutions have a performance issue due to heavyweight symbolic execution and the path explosion issue. HermeScan addresses these challenges by adopting a lightweight, on-demand, context-sensitive RDA-based taint analysis, with the help of enhanced CFG recovery and more precise taint source identification. Our prototype system has successfully discovered 87 zero-day bugs in 30 firmware samples. Compared to SOTA tools, HermeScan could find more bugs in less time with much fewer false positives.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful suggestions on our paper. This work is supported in part by the National Key Research and Development Program of China (2021YFB2701000), the Key R&D Special Program of Henan Province (No.221111210300), and the National Natural Science Foundation of China (grant # 61972224, #62272265).

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[2] H. Casanova, "Linking and loading," 2010.

[3] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems," in *30th USENIX Security Symposium*, 2021, pp. 303–319.

[4] K. Cheng, D. Fang, C. Qin, H. Wang, Y. Zheng, N. Yu, and L. Sun, "Automatic inference of taint sources to discover vulnerabilities in soho router firmware," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2021, pp. 83–99.

[5] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2018, pp. 430–441.

[6] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye, and L. Sun, "Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 360–372.

[7] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *2006 22nd Annual Computer Security Applications Conference*. IEEE, 2006, pp. 269–278.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[9] C. Eagle, *The IDA pro book*. no starch press, 2011.

[10] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 337–350.

[11] I. Gotovchits, R. Van Tonder, and D. Brumley, "Saluki: finding taint-style vulnerabilities with static property checking," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, vol. 2018, 2018.

[12] J. Grossklags and C. Eckert, "πcfi: Type-assisted control flow integrity for x86-64 binaries," in *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, vol. 11050. Springer, 2018, p. 423.

[13] M. Hasan, "State of iot 2022: Number of connected iot devices growing 18% to 14.4 billion globally," https://iot-analytics.com/number-connected-iot-devices, 2022.

[14] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Annual Computer Security Applications Conference*, 2020, pp. 733–745.

[15] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level." in *NDSS*, 2021.

[16] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[17] C. Luo, P. Li, and W. Meng, "Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2175–2188.

[18] A. Møller and M. I. Schwartzbach, "Static program analysis," *Notes. Feb*, 2012.

[19] S. Muralee, I. Koishybayev, A. Nahapetyan, G. Tystahl, B. Reaves, A. Bianchi, W. Enck, A. Kapravelos, and A. Machiry, "{ARGUS}: A framework for staged static taint analysis of {GitHub} workflows and actions," in *32nd USENIX Security Symposium*, 2023, pp. 6983–7000.

[20] E. N, "Sam iot security report," https://securingsam.com/wp-content/uploads/2022/04/SAM_IOT-Security-Report.pdf, 2022.

[21] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[22] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[23] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2004.

[24] N. A. Quynh and D. H. Vu, "Unicorn: Next generation cpu emulator framework," *BlackHat USA*, vol. 476, 2015.

[25] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: On the security of bootloaders in mobile devices." in *USENIX Security Symposium*, 2017, pp. 781–798.

[26] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1544–1561.

[27] D. Selivanov and contributors, "text2vev," https://text2vec.org/, 2022.

[28] Semmle, "Codeql for research," https://securitylab.github.com/tools/codeql, 2022.

[29] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna,

"Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, vol. 1, 2015, pp. 1–1.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 138–157.

[31] H. Su, F. Li, L. Xu, W. Hu, Y. Sun, Q. Sun, H. Chao, and W. Huo, "Splendor: Static detection of stored xss in modern web applications," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1043–1054.

[32] C. J. Tan, J. Mohamad-Saleh, K. A. M. Zain, and Z. A. A. Aziz, "Review on firmware," in *Proceedings of the International Conference on Imaging, Signal Processing and Communication*, 2017, pp. 186–190.

[33] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, Z. Smith *et al.*, "Greenhouse: Single-service rehosting of linux-based firmware binaries in user-space emulation," in *32nd USENIX Security Symposium*, 2023, pp. 5791–5808.

[34] J. Vadayath, M. Eckert, K. Zeng, N. Weideman, G. P. Menon, Y. Fratantonio, D. Balzarotti, A. Doupé, T. Bao, R. Wang *et al.*, "Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs," in *31st USENIX Security Symposium*, 2022, pp. 413–430.

[35] L. Wang, F. Li, L. Li, and X. Feng, "Principle and practice of taint analysis," *Journal of Software*, vol. 28, no. 4, pp. 860–882, 2017.

[36] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution." in *NDSS*, 2009.

[37] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[38] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys*, vol. 54, no. 1, pp. 1–36, 2021.

[39] W. Xie, J. Chen, Z. Wang, C. Feng, E. Wang, Y. Gao, B. Wang, and K. Lu, "Game of hide-and-seek: Exposing hidden interfaces in embedded web applications of iot devices," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 524–532.

[40] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2525–2527.

[41] L. Zhang, J. Chen, W. Diao, S. Guo, J. Weng, and K. Zhang, "Cryptorex: Large-scale analysis of cryptographic misuse in iot devices," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019, pp. 151–164.

[42] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl:high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th USENIX Security Symposium*, 2019, pp. 1099–1114.

[43] Z. Zhong, J. Liu, D. Wu, P. Di, Y. Sui, and A. X. Liu, "Field-based static taint analysis for industrial microservices," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 149–150.

[44] W. Zhu, Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang, "Callee: Recovering call graphs for binaries with transfer and contrastive learning," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2357–2374.