FITS: Inferring Intermediate Taint Sources for Effective Vulnerability Analysis of IoT Device Firmware

Puzhuo Liu
Institute of Information Engineering,
Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
liupuzhuo@iie.ac.cn

Yaowen Zheng *
Nanyang Technological University
Singapore
yaowen.zheng@ntu.edu.sg

Chengnian Sun
School of Computer Science,
University of Waterloo
Waterloo, Ontario, Canada
cnsun@uwaterloo.ca

Chuan Qin
Institute of Information Engineering,
Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
qinchuan@iie.ac.cn

Dongliang Fang
Institute of Information Engineering,
Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
fangdongliang@iie.ac.cn

Mingdong Liu
Institute of Information Engineering,
Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
liumingdong@iie.ac.cn

Limin Sun
Institute of Information Engineering,
Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
sunlimin@iie.ac.cn

ABSTRACT
Finding vulnerabilities in firmware is vital as any firmware vulnerability may lead to cyberattacks to the physical IoT devices. Taint analysis is one promising technique for finding firmware vulnerabilities thanks to its high coverage and scalability. However, sizable closed-source firmware makes it extremely difficult to analyze the complete data-flow paths from taint sources (i.e., interface library functions such as recv) to sinks.

We observe that certain custom functions in binaries can be used as intermediate taint sources (ITSs). Compared to interface library functions, using custom functions as taint sources can significantly shorten the data-flow paths for analysis. However, inferring ITSs is challenging due to the complexity and customization of firmware. Moreover, the debugging information and symbol table of binaries in firmware are stripped; therefore, prior techniques of inferring taint sources are not applicable except laborious manual analysis.

To this end, this paper proposes FITS to automatically infer ITSs. Specifically, FITS represents each function with a novel behavioral feature representation that captures the static and dynamic properties of the function, and ranks custom functions as taint sources through behavioral clustering and similarity scoring.

We evaluated FITS on 59 large, real-world firmware samples. The inference results of FITS are accurate: at least one of top-3 ranked custom functions can be used as an ITS with 89% precision. ITSs helped Karonte find 15 more bugs and helped the static taint engine find 339 more bugs. More importantly, 21 bugs have been awarded CVE IDs and rated high severity with media coverage.

CCS CONCEPTS
• Computer systems organization → Embedded and cyber-physical systems; • Security and privacy → Systems security; • Software security engineering; • Software and its engineering → Software notations and tools.

KEYWORDS
Firmware, Taint analysis, Vulnerability

ACM Reference Format:

This work is licensed under a Creative Commons Attribution International 4.0 License.
ASFLOS ’23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0394-2/23/01
https://doi.org/10.1145/3623278.3624759
1 INTRODUCTION

Internet of Things (IoT) devices have become increasingly pervasive in modern society; by 2025 the number of IoT devices in use will reach 21.5 billion [50]. IoT devices input has greatly facilitated our daily lives, but meanwhile have gradually become the target of various cyberattacks (e.g., botnets [2], privacy theft [31], extortion [3], APT [32]) due to various bugs and design flaws in their firmware. Among all IoT devices, Internet-connected embedded devices such as routers, access points, and gateways are more vulnerable to attacks than other devices [9, 30]. This mainly because the firmware of such devices directly exposes Internet network access, which have complex implementations and often contain exploitable vulnerabilities. Moreover, these devices serve as entry points to the local network, and are often exploited as bridges to launch attacks to other proximity communication and simple-implemented IoT devices such as smart plugs and cleaning robots within the same network [61]. To this end, there is an urgent need to design effective, scalable, and practical techniques for automatically discovering vulnerabilities in the firmware of Internet-connected embedded devices.

Static taint analysis is such a promising technique for finding vulnerabilities in embedded devices, e.g., DTaint [12], Karonte [44], and SaTc [9]. It can be applied directly and statically to analyze firmware, achieving high code coverage without requiring emulation or real embedded devices for analysis. The workflow of classical firmware taint analysis contains three major steps: (1) Identifying as taint sources interface library functions that receive user data, e.g., recv, getenv, fgets. (2) Identifying as sinks unsafe library functions that can lead to buffer overflow or command hijacking, e.g., system, sprintf, strcpy. (3) Analyzing the data flow from taint sources to sinks, and determining the existence of bugs by checking whether the tainted data on the data flow reaches any sink without being sanitized. Challenges. However, performing accurate data flow analysis from taint sources to sinks has always been a challenging problem for applying taint analysis on firmware, even with the state-of-the-art techniques [9, 12, 44]. The reason is manifold. First, firmware consists of multiple binaries, and user input (i.e. the taint source) often flows across binaries, which is a complex scenario for tracking data flow. For example, the web server uses the common gateway interface (CGI) to call other binaries to process user requests. Second, even within a single binary, data aliasing, indirect calls via function pointers or jump tables, debug information stripped, etc. further complicate the analysis of the data flow. Lastly, due to the multi-binary nature of firmware and the complexity of individual binary, the data flow from taint sources to sinks can be excessively long, and difficult to be precisely identified. In the past, techniques such as symbolic execution [9, 44], value set analysis [9, 44], and alias analysis [12] were used to mitigate this problem to some extent. But at the cost of increased false positives and runtime overhead, hindering the practicality of conducting taint analysis in large and complex real-world firmware.

A New Approach via Intermediate Taint Sources. To improve the effectiveness of tainted data flow analysis, rather than relying on heavyweight techniques [9, 12, 44], we take a different, fresh perspective: we aim to shorten the length of the data-flow path from the taint source to the sink, which can significantly reduce the complexity of the analysis problem for existing data flow analysis algorithms. To this end, we propose intermediate taint sources (ITSs), as an alternative to the classical taint sources (CTSs) that are interface library functions directly receiving user input. An ITS is a custom function (not a library function) that processes user input received via library functions and returns a part of the input to be used by other functions. Concretely, the concept of ITSs is inspired by our observation that the logic of handling a user input in Internet-connected IoT devices typically involves three steps: first, receiving the structured user input, which usually contains multiple fields through interface library functions; second, saving the user input to a memory region if the input conforms to the format requirement; and finally, fetching one or more fields from the memory region for subsequent processing. Figure 1b shows such an example of ITSs. The function fn16 fetches the user input received by a library function and stored at src_addr, based on the index, and then returns the corresponding data to other functions for further processing. An ITS is a node on the data flow from a CTS to a sink; and its distance to the sink is always shorter than that of the CTS, thus reducing the complexity of the data flow for taint analysis. More details of the ITS are discussed in §2.

However, determining whether a function Fn in stripped firmware can be classified as an ITS is non-trivial. First, ITSs do not have a definitive detection criterion as CTSs, and have versatile implementations, especially across firmware of different device vendors. Second, finding ITSs requires reverse engineering not only Fn to model the semantics (i.e., behaviors) of Fn, but also the other functions to understand the interprocedural data and control dependencies between Fn and the other functions. Third, firmware is usually stripped of debug information and symbol tables, resulting in loss of semantic information (e.g., names of variables and functions) and further exacerbating the technical difficulty of understanding the behaviors of Fn. Lastly, it is usually impossible to apply dynamic analysis to find ITSs, because of vendor’s protective measures for devices (such as hidden debug interfaces and non-public memory mapping between firmware and hardware) which make it difficult to obtain runtime information that characterizes the behaviors of Fn.

Inferring ITSs. We infer ITSs by leveraging the aforementioned observation on how user inputs are handled in Internet-connected IoT devices. Our insight is that an ITS function should read memory to fetch data, derive new data from the fetched data, and return the new data via return value or pointers. Therefore, an ITS behaves like a memory operation function. Considering there are various standard library functions that operate memory such as strncpy, memcpy, and strftime, we treat the implementation of these functions as anchors and analyze the similarities between anchor functions and custom functions to identify ITSs. Note that we aim to measure the similarity between anchors and a custom function in terms

---

1 In this paper we focus on static taint analysis only as dynamic taint analysis is usually not applicable to analyzing programs running in embedded devices. For brevity, taint analysis in this paper refers to static taint analysis.
of behaviors but not code self; thus our work is different from
the existing techniques that measure binary code similarity [17, 21, 34, 58]. Moreover, due to the difficulty of obtaining runtime
information from embedded devices and the low coverage problem
of dynamic analysis, we apply static analysis to extract structural
and flow features to describe the static and dynamic properties of
functions.

This paper proposes FITS (Inferring Intermediate Taint Sources), a
novel methodology to identify ITSs by extracting a descriptive
feature vector from each function and adopting a multi-stage strategy
to compare behavioral similarities between custom and anchor
functions. Specifically, FITS first identifies a set of binaries in the
firmware that export network services and converts these binaries
into an intermediate language to facilitate analysis. Then, FITS per-
forms reaching definition analysis and call site analysis based on the
control flow graph and call graph generated by under-constrained
symbolic execution [41] to extract structural and flow features for
each function. To reduce the false positives due to the order of mag-
nitude difference in feature dimensions, FITS clusters all custom functions
based on their feature vectors, and only selects functions
in the class with high complexity as candidates. Finally, FITS uses
the cosine distance to calculate the similarity between the can-
didates and the anchor functions to rank the functions that are likely
to be ITSs.

We evaluated FITS on 59 real-world firmware from 5 popular
vendors. The results show that FITS successfully found at least one
ITS in each firmware with the top-3 precision of 89%. To illustrate
that ITSs are beneficial for finding vulnerabilities, we added ITSs to
the state-of-the-art taint analysis engine Karonte [44]. Compared
with the vanilla Karonte, 15 more bugs were found with the help of
ITSs. Furthermore, since the symbolic execution technique limits
Karonte’s analysis efficacy, we implemented a static taint analysis
engine. ITSs helped the static engine find 339 more bugs. As of
now, 67 of the 141 bugs found in the latest firmware have been
confirmed by the vendors, and 21 of them have been awarded CVE IDs. The severity of these vulnerabilities is rated high by
the National Vulnerability Database (the CVSS score greater than
7.2) [16], allowing hackers to launch remote attacks on devices such
as denial of service and remote code execution.

Contributions. We make the following major contributions.

- We propose a novel concept of intermediate taint sources (ITSs) to tackle the challenges of performing effective taint
analysis on stripped firmware of Internet-connected IoT de-
VICES.
- We propose the first methodology FITS to automatically, ac-
curately discover ITSs from stripped firmware. FITS includes
two major components: a novel behavior feature representation
to summarize the static and dynamic properties of functions,
and a multi-stage strategy for measuring behavioral
similarity to infer ITSs.
- Our comprehensive evaluations on 59 real-world firmware
demonstrate the effectiveness of FITS in inferring ITSs with
the top-3 precision of 89%. More importantly, ITSs signifi-
cantly boosted the performance of taint analysis and helped
the static engine find 339 more bugs, 21 of which were
awarded CVE IDs and rated high severity.

2 MOTIVATION
Figure 1 shows how an ITS helps improve the efficacy and effi-
ciency of taint analysis [9, 12, 44] to find vulnerabilities in stripped
firmware of Internet-connected embedded devices. As mentioned in
§1, such embedded devices have more functionalities and are more
vulnerable to cyberattacks than other devices that do not directly
access the Internet [38, 51]. However, the task of identifying vul-
nerableities in the firmware of embedded devices is challenging due
to the intricate implementation that varies across different vendors.
Furthermore, the firmware is typically stripped down and highly

Figure 1: Typical user input propagation process in an Internet-connected IoT device, reverse-engineered from the firmware of
NETGEAR R7000P and manually cleaned for readability.

```c
char *fn16(char * index, char * src_addr, int length) {
    int i = 0;
    while (i < length) {
        if (!(*(src_addr + i))) return -1;
        if (i != 0) return i;
    }
    return 0;
}
```
char *strcpy(char *strDest, const char *strSrc) {
...
char *address = strDest;
while (!(*strDest++ = *strSrc++)) (*'\0') = NULL;
...
}
int memcmp(const void *str1, const void *str2, int count){
...
while (-count && (*str1) == (*str2)){
str1 = (char *)str1 + 1;
str2 = (char *)str2 + 1;
}
return ((char *)str1) - ((char *)str2));
}
char *strstr(const char *haystack, const char *needle){
char *s1, *s2, *cp = (char *)haystack;
...
while (*cp){
  s1 = cp;
  s2 = (char *)needle;
  while (!(*s2 && (!*s1 - s2)) s1++, s2++;
  if (!*s2 return return cp);
  cp++;
}
...
}

Figure 2: Examples of anchor functions.

optimized, lacking debug information and presenting difficulties in analysis.

Figure 1a shows the call graph of the workflow of processing user input from the web server, reverse-engineered from NETGEAR R7000IP. The function fn14 starts the web service as a daemon. When the interface library function recv, called in fn1, receives a user request, all the user data in the request after the format check is saved to a global variable for subsequent use. The function fn15 calls fn16 to extract certain user data fields from the global variable, e.g., extracting the user name and password for login authentication. Then fn17 calls the corresponding data processor (i.e., fn17 in Figure 1a) to execute the extracted data to complete the login authentication.

CTS. Prior work [9, 12, 44] marks interface library functions that receive user data as taint sources (e.g., recv), referred to as CTs. It is easy to identify such library functions even in stripped firmware thanks to the ubiquitous use of dynamic linking of libraries. However, the major problem with CTs is the difficulty of computing the data flow from CTs to sinks. For example, there is a dangerous data flow from recv to sprintf in Figure 1a, which is difficult to identify for the following reasons. First, the functionalities provided by Internet-connected IoT devices require large, complex implementation that inevitably involves numerous function calls and conditional branches, leading to scalability issues, such as large memory overhead and path explosion, that challenge the state-of-the-art data flow analysis. Second, the names and types of variables and functions are not available in the stripped binary, which further complicates data flow analysis. The various forms and granularities of data sharing, including global variables, pointers, arrays, and I/O addresses, make alias analysis ineffective in identifying data sharing [10, 53]. Third, indirect function calls via function pointers or jump tables further leads to the interruption of data flow. The value set analysis and its evolution technique are proposed to mitigate these problems [1, 5, 33]. However, the significant overhead and low precision of these techniques hinder their applications in analyzing real-world programs [64]. Therefore, the data flow in the firmware cannot be effectively analyzed with CTSs, resulting in a high number of false negatives.

ITS. An ITS is a custom function (not a library function) that extracts a part of the user input and passes out the result via return values, pointers, global variables, etc. ITSs can be used to mitigate the aforementioned problems of the CTSs. This new type of taint sources is inspired by our key observation of how developers handle user input obtained from the CTSs in Internet-connected IoT firmware. Concretely, such user input is usually a structure consisting of multiple fields. For example, an HTTP request to an IoT device may include a user name and a password for authentication. The input is stored in the heap, and when a field in the input is required, a custom function extracts the required field from the input, and returns the extracted user data. Such a custom function is regarded as an ITS, and we strongly believe that the ITS is a good alternative of the CTS, because it is more adjacent to the sink than the CTSs, facilitating data flow analysis. The function fn16 is an ITS. Figure 1b shows the reverse-engineered, simplified decompiled code of fn16. If fn16 is used as a taint source, the difficulty of computing the data flow from recv to sprintf can be significantly reduced. In Figure 1b, fn16 fetches the data at the src_addr according to the index and returns the data. The src_addr stores the data obtained by recv. The return value is used by sprintf and stored into another variable. An overflow bug occurs if the length of the return value from calling fn16 is not checked and exceeds the size of the variable buffer.

Inferring ITSs. The ITS inference for stripped binaries of firmware is a brand new endeavor. Unlike prior work of inferring taint sources from Java bytecode [42], our technique FITS has to deal with stripped binaries that do not have sufficient semantic information (e.g., names of variables and functions). To tackle this challenge of missing semantic information, we propose a multi-stage, similarity-based approach to automatically and accurately identify ITSs: our key insight is that ITSs should be similar to the library functions that operate memory (i.e., anchor functions in this paper) due to the aforementioned characteristics of ITSs. For example, the ITS fn16 is behaviorally similar to the three anchor functions in Figure 2: taking arguments which include pointers and values of scalar types, processing the memory with the arguments, and returning the result via a return value or a pointer. Note that we use anchors to identify custom functions that exhibit similar patterns of memory operations, and the goal is not to identify custom functions that are behaviorally equivalent to the anchor functions.

3 METHODOLOGY
3.1 System Overview
Figure 3 shows the overall workflow of FITS, consisting of the following three stages.

1) Pre-processing Firmware. This stage unpacks the firmware to obtain the binaries and selects the binaries containing the network interface as the analysis target, because network communication
is a major source of cyber threats to IoT devices. Then, we find all libraries on which the selected binaries depend. At the same time, we identify library functions with memory operation behavior as anchor functions by standard library function names. Finally, the selected binaries and dependency libraries are converted to intermediate language (IB) to facilitate later analysis. Implementation details are available in §3.4.

2. Representing Function Behaviors. To infer ITSs, we need to analyze the behavioral characteristics of each custom and anchor function. To this end, this stage builds a feature vector to represent the behavior of functions. However, building the function behavioral representation for stripped binaries is challenging due to missing semantic information and there is no prior work yet. To tackle this challenge, we design a novel feature representation of functions, called Behavioral Feature Vector (BFV). A BFV includes two categories of features:

- **structure features** (SF) to represent the static properties of a function, e.g., the number of basic blocks, the existence of loops.
- **flow features** (FF) to represent the dynamic properties of a function, e.g., whether the function parameter controls loop or branch.

Table 1 lists the details of the features used in FITS. To compute these features, we leverage the under-constrained symbolic execution technique (UCSE) \cite{41} to generate control flow graph (CFG) and call graph (CG) for each function. First, we perform graph analysis on the CFG and CG of the function under analysis (Fn) to extract structural features. Second, we apply reaching definition analysis to the CFG of Fn to understand how the parameters of Fn affect its branches or loops to extract intraprocedural flow features. Third, for interprocedural flow features, we analyze the function calls to Fn, and backtrack through the memory structure to find the origins of the arguments. The BFVs of custom functions and anchor functions constitute the behavioral representation (BR) for further analysis.

3. Inferring Intermediate Taint Sources. This stage scores the behavioral similarity between custom functions and anchor functions to infer ITSs. However, scoring custom functions directly may lead to false positives due to the order of magnitude differences in feature dimensions. To this end, we first perform cluster analysis on all custom functions, and screen candidate custom functions by statistical characteristics of classes. In this way, some functions that may lead to false positives are discarded. Finally, each ITS candidate is scored with the anchor functions to infer ITSs.

### Table 1: Features used in FITS.

<table>
<thead>
<tr>
<th>Category</th>
<th>Feature Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural Feature</td>
<td>1. number of basic blocks</td>
</tr>
<tr>
<td></td>
<td>2. existence of loops</td>
</tr>
<tr>
<td></td>
<td>3. number of callers</td>
</tr>
<tr>
<td></td>
<td>4. number of parameters</td>
</tr>
<tr>
<td></td>
<td>5. number of call anchor functions</td>
</tr>
<tr>
<td></td>
<td>6. number of call library functions</td>
</tr>
<tr>
<td>Flow Feature</td>
<td>7. whether parameters control loops</td>
</tr>
<tr>
<td></td>
<td>8. whether parameters control conditional branches</td>
</tr>
<tr>
<td></td>
<td>9. whether parameters passed to anchor functions</td>
</tr>
<tr>
<td></td>
<td>10. whether arguments contain strings</td>
</tr>
<tr>
<td></td>
<td>11. number of different strings in all call sites</td>
</tr>
</tbody>
</table>

### Algorithm 1: Compute behavioral representations.

```plaintext
Input: Bin: the binary under analysis
Input: Libs: the dependency libraries of Bin
Output: BR: the dictionary containing all functions BFV
1 for each function Fn in Bin do
2 \[
\begin{align*}
\text{cfg}_{\text{fn}}, \text{strs}_{\text{fn}} &\leftarrow \text{UCSE-based analysis on Bin, Libs, and Fn} \\
\text{rdefs} &\leftarrow \text{reaching definition analysis on } \text{cfg}_{\text{fn}} \text{ and params}
\end{align*}
\]
3 for each function Fn in Bin do
4 Initialize SF, FF\text{intra}, FF\text{inter}, lists with zero or False
5 SF ← structural analysis on \text{cfg}_{\text{fn}} and \text{cfg}_{\text{fn}}
6 vars, params ← identify key vars and params in \text{cfg}_{\text{fn}}
7 rdefs ← reaching definition analysis on \text{cfg}_{\text{fn}} and params
8 FF\text{intra} ← use vars to match rdefs
9 for each caller in Fn do
10 \text{arg}_\text{caller} ← call site analysis on \text{cfg}_\text{caller} and Fn
11 \text{str}_\text{caller} ← backtrack memory on \text{cfg}_\text{caller} and args
12 FF\text{inter} ← analysis \text{str}_\text{caller} of each caller
13 // The operator + represents the operation of list concatenation.
14 BFV ← SF + FF\text{intra} + FF\text{inter}
15 \text{BR}[\text{Fn}] = BFV
16 return BR
```
context-sensitive, object-sensitive), but face the problem of untractable memory overhead and path explosion. To this end, we perform CFG generation based on UCSE, starting the analysis from the entry point of Fn. Then, structural features can be obtained by analyzing the generated CFG.

**Intraprocedural Flow Features.** Intraprocedural features refer to the relationship between variables of loop control, branch control, memory operation library function and Fn parameters. To understand the relationship, we apply reaching definition flow data analysis to the CFG of Fn. Specifically, we first find out all variables in Fn, which control loops or branches, or are passed as arguments to anchor functions. Then, all locations where parameters are used are identified and labeled as defaddr. At the same time, reaching definition analysis is performed based on defaddr set to form the data dependency graph (DDG). In this way, the processing logic of parameter functions can be expressed abstractly. Finally, we analyze whether the parameters of Fn are related to variables of loop control, branch control, and memory anchor functions according to DDG.

**Interprocedural Flow Features.** Interprocedural features refer to whether the arguments contain a string when the Fn is the callee. Concretely, since the network communication data is structured, user input is usually stored in the memory in the form of keywords combined with user input [44]. When some user input needs to be processed, keywords, such as username and password, are used as indexes to obtain the corresponding part of the user input. Therefore, it is reasonable to treat the arguments of Fn with strings as interprocedural flow features.

However, diverse addressing modes and the loss of variable types pose challenges to determine whether an argument is a string. To this end, we perform call site analysis based on the CFG of the caller of Fn and backtrack according to memory structures. First, we recursively collect all program points that call Fn, and the registers that store the arguments. Then, we follow the principle shown in the Table 2 to track the register backward and end up when the register can be represented with a constant. The r in the table represents a register, and the t represents a temporary variable. Next, we analyze the constant and retain the constant that refers to an address in the program. Here we denote it as PT. If PT points to the read-only section (i.e., rodata) in the program, we can infer that the argument type is a string. In some cases, strings are generated dynamically during program execution, so they are stored in the data section. The reference method of these strings is similar to global offset table [52], which provides an intermediate table to store the pointers to these strings. Thus, if PT points to data section, we retrieve the content pointed by PT, and denote it as MT. If MT also refers to an address, we further acquire the content point by MT. This content usually contains some hint strings to wait for the runtime to populate.

**An Example.** We use fn16 in Figure 1b as an example to illustrate the BFV composition. We assume that fn16 is called in two places i.e.,

- fn16("User_Name", *((const char**) v1, 10)
- fn16(v2, *((const char**) v1, 20)

where v1 points to uninitialized memory and v2 points to a string. In the order of the feature names shown in Table 1, the BFV of fn16 is computed as [17, True, 2, 3, 5, 6, True, True, True, True, 2].

### 3.3 Intermediate Taint Source Inference

To infer ITSs, we score BFV of a custom function by measuring its similarity with that of anchor functions. The BFV is more similar to that of anchor functions, the custom function is more likely to be an ITS. But before scoring stage, we need to first cluster the custom functions based on BFV and select candidate custom functions, otherwise many false positives would be introduced due to the order of magnitude difference in feature dimensions. For example, the error output function takes a string representing the error message as input, and processes it based on the content of the string, which is somewhat similar to the anchor function. The number of anchor functions called by the error output function is less than that of the ITS, but the number of called of the error output functions is far more than that of the ITS. So the number of calls dominate the score. However, feature processing operations, such as dimensionality reduction or normalization, affect the fidelity of the data. Moreover, the features that dominate the results in different binaries are not fixed, which makes it difficult to use weights to solve this problem universally. To this end, we adopt clustering idea to weaken the individual features of a function and filter out false positives through the statistical features of the class.

**Behavior Clustering.** To cluster functions based on their BFVs, we use a classic unsupervised learning algorithm called DBSCAN [54]. Here we follow an assumption that memory operation functions have relatively complex behavior, which is reflected in the complexity of the function implementation [43]. Therefore, we calculate
the complexity of each class and then filter out classes that have less complexity. The complexity calculation of a function utilizes the number of basic blocks (bb), calls from the parent (caller), calls to library functions (lib), and calls to anchor functions (anchor) as shown in eq. (1). The $N$ is the number of custom functions in the class. Normalization of each dimension by the maximum value is performed at the calculation time. When the complexity of a class ($C_{class}$) is greater than the average ($C_{average}$) of all classes, the functions in the class are used as candidate functions for the next step of scoring. The reduction of false positives is achieved in this way, which is evaluated in §4.5.

$$
C_{class} = \frac{\sum_{i=0}^{N} [bb_i + caller_i + lib_i + anchor_i]}{N} \tag{1}
$$

**Behavior Scoring.** According to the anchor functions, we score each candidate custom function to identify ITSs. To eliminate biases introduced by certain features, we use cosine distance to determine the similarity of feature representations. Cosine distance measures similarity using vector angles, prioritizing relative differences over absolute differences in values. Moreover, the cosine distance maintains better stability in high-dimensional space than other methods. The BFVs of anchor functions are formed into a matrix for candidate custom functions scoring. The behavioral similarity score of Fn is obtained by calculating the distance from the matrix as shown in eq. (2). The $N$ represents the number of anchor functions. The vstack represents the concatenation of a vector and a matrix. Ultimately, functions with high ratings are more likely to be considered as ITSs.

$$
S_{Fn} = \frac{\sum_{i=0}^{N} \sum_{j=1}^{N} [1 - cosine(vstack([BR[Fn], Matrix]))]_{i,j}}{N} \tag{2}
$$

### 3.4 Implementation

**FITS.** FITS is implemented in Python. Unpacking firmware is implemented based on Binwalk [46] to extract the filesystem from firmware. For firmware using private encoding or encryption, FITS decrypts the firmware according to the magic byte of the file header and then unpacks it [19]. After obtaining the filesystem, FITS selects the network binary for further analysis based on PIE [14]. At the same time, the dependent libraries are determined according to the dynamic section in the header of the network binary. Based on BootStomp [43], anchor functions are determined by matching standard library function names in the dynamic link library. In the following program analysis, the binary and libraries are disassembled and converted to VEX intermediate language [40]. Based on Angr [59], we extract the CFG and CG of the functions and analyze the data flow and control flow to achieve the extraction of BFV. After obtaining the function behavior representation, we utilize scikit-learn [48] to perform the similarity between custom functions and anchor functions, thereby ranking ITSs. This approach enables FITS to perform scalable and semantic-agnostic ITS inference

**Static Taint Analysis Engine (STA).** We implemented a static taint analysis engine (STA) to find bugs in embedded device firmware. Given taint sources, together with sinks to identify all vulnerable data flow on the CFG and CG is equivalent to computing the reachability subgraph starting from the taint sources ending at the sinks.

To accomplish this, we used IDA Pro [26] to disassemble the binary and generate its CFG and CG, which we then converted to VEX intermediate language. Data flow analysis starts from the designated taint sources, with the return value or parameter register of different taint source call sites marked with unique taint labels. Taint analysis involves tracking data dependencies and sanitizing taint labels (similar to the sanitizing setting of Karonte [44]), for instance, copying constants to tainted memory. To detect vulnerabilities, STA set sinks by matching library function names to identify two types of vulnerabilities, namely buffer overflow (e.g., `strncpy`, `sprintf`, `strncat`) and command hijacking (e.g., `system`, `execve`).

### 4 EVALUATION

We designed the evaluations of FITS to answer the four research questions below:

- **RQ1:** Efficacy of FITS in inferring ITSs.
- **RQ2:** Efficacy of ITSs in detecting vulnerabilities.
- **RQ3:** Efficacy of BFV in inferring ITSs.
- **RQ4:** Efficacy of the multi-stage strategy in inferring ITSs.

#### 4.1 Evaluation Setup

**Dataset.** We evaluated FITS on 59 real-world firmware samples, encompassing both the Karonte dataset [44] and new version firmware samples. The inclusion of the new version firmware samples aims to demonstrate FITS’s capability to detect 0-day vulnerabilities, as vendors do not accept proof-of-concept (PoC) of bugs for outdated products. The samples were obtained from well-known IoT vendors, such as NETGEAR, D-Link, and TP-Link, covering WiFi routers, access points, and gateways, featuring firmware architecture that includes ARM, AARCH64, and MIPS.

**Baseline.** To answer the proposed research questions, we select different baselines as follows.

- **RQ1:** We choose BootStomp [43], which utilizes a heuristic-based taint source inference module on stripped bootloaders, as a contrast of FITS to illustrate the efficacy of ITS inference. SUSI [42] is not selected as the baseline due to its reliance on semantic information, such as function and variable names, for taint source inference, which is not applicable to stripped firmware.
- **RQ2:** We integrate ITSs into two taint analysis engines: Karonte [44] and STA (implemented in §3.4). The efficacy of ITSs in detecting vulnerabilities is demonstrated by comparing the results of taint analysis using only CTSs.
- **RQ3:** We analyze the composition of BFV and compare BFV with the state-of-the-art binary code representations to illustrate the efficacy of BFV in inferring ITSs. Firstly, we perform the ablation study of BFV with different feature combinations. Then, we compare BFV with two prominent binary code representations: Augmented-CFG of NERO [17] and Attributed-CFG of Gemini [58]. These two representations have achieved good results in function summarization and code similarity analysis in stripped binaries.
- **RQ4:** We evaluate the behavioral clustering and scoring stages independently. To demonstrate the efficacy of clustering, we remove the stage and replace the stage with dimensionality reduction [45] and data preprocessing [15] for
comparison. Regarding the behavior scoring stage, we experiment with other similarity distance calculation methods, including Euclidean distance [55], Manhattan distance [56] and Pearson correlation coefficient [57], to compare the efficacy of Cosine distance.

**Verifying Inferred ITSs and Identifying Taint Origins.** We manually verify whether the inferred results of FITS can be used as ITSs, and if so, the register or global variable storing fetched unsanitized user data is regarded as the taint origin. The verification is challenging because it requires a deep understanding of how user input is processed in closed-source, stripped, optimized firmware. To tackle this challenge, we use totally three different verification methods for different firmware-specific circumstances: firmware rehosting, real device debugging, and semantic information analysis. More details on these methods are available in Appendix A. When we use an ITS as a taint source for taint analysis, and we need to specify the taint origin of this ITS. A taint origin is either a register or a global variable that stores unsanitized user data produced by the ITS. The identification of the taint origin is done during the verification of the ITS in Appendix A. For example, the ITS function fn16 in Figure 1b, its return value register is used as a taint origin, and any data stored in the register is marked with a taint label.

**Configurations.** All experiments were performed on a Linux workstation with an Intel Core i7-8750H CPU and 64G RAM.

### 4.2 Taint Source Inference (RQ1)

We applied FITS to the dataset for ITS inference. After getting the ranking results of FITS, we need to verify whether a custom function can be used as an ITS. As mentioned in §1, the ITS function should extract and return a part of the user input. Therefore, we adopt this criterion and methods in Appendix A to verify the ITSs inferred by FITS.

Table 3 displays the top-1, top-2, and top-3 precision rates for inferring ITSs in the firmware dataset using FITS. The top-n precision indicates that the top n results have at least one custom function that can serve as an ITS. The top-3 precision of FITS attains 89% on average across diverse firmware samples, which proves the effectiveness and generality of FITS.

Out of the total firmware samples, only six firmware samples cannot be successfully inferred by FITS. We further performed manual analysis to investigate the reasons for the failure. The findings indicate that four firmware samples could not locate the correct target for analysis in the firmware pre-processing stage. The remaining two were due to the fact that network data is directly stored in a structure. The program uses the array offset to retrieve the data in the subsequent processing. Such data processing is frequently encountered in devices with simple designs. The relatively small size of these programs allows taint analysis to commence from interface library functions.

Table 4 presents the detailed inference results of firmware randomly selected from different vendors. The results demonstrate that ITSs are accurately inferred from a vast number of functions. As it is impractical to gather all custom functions that can function as ITSs from stripped firmware, we did not calculate the recall. According to existing work [12], which manually analyzes the firmware to determine the taint source, at most one or two custom functions in a binary may be the effective taint source. Moreover, it is impossible to duplicate functions that achieve the same purpose in terms of the system development specification. We randomly selected 5 firmware samples from different vendors for analysis. After analyzing each custom function, no other functions can be used as ITSs.

**Table 3: Statistics of ITS inference results.**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vendor</th>
<th>Series</th>
<th># Firmware</th>
<th>Top-1</th>
<th>Top-2</th>
<th>Top-3</th>
<th>Avg time (hh:mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETGEAR</td>
<td>R800-NR</td>
<td>17</td>
<td>71%</td>
<td>100%</td>
<td>100%</td>
<td>9.11</td>
<td></td>
</tr>
<tr>
<td>Latest</td>
<td>D-Link</td>
<td>9</td>
<td>33%</td>
<td>33%</td>
<td>78%</td>
<td>0.37</td>
<td></td>
</tr>
<tr>
<td>Firmware</td>
<td>TP-Link TD-WA-WR-TX/KC</td>
<td>16</td>
<td>36%</td>
<td>63%</td>
<td>81%</td>
<td>2.13</td>
<td></td>
</tr>
<tr>
<td>Tenda</td>
<td>AC/WH/WH</td>
<td>7</td>
<td>43%</td>
<td>43%</td>
<td>86%</td>
<td>2.13</td>
<td></td>
</tr>
</tbody>
</table>

Table 3 shows that the average time taken by FITS to complete firmware analysis is 2 hours and 57 minutes. We further investigated the binary properties that influence the analysis efficiency and discovered that the number of functions and the file size of the target binary have a strong positive correlation with the analysis time. Specifically, we plotted the analysis time against the number of functions and the binary size in Figure 4. The results illustrate that the analysis time is positively correlated with both the number of functions and the size of the target binary. The scattered points on the figure imply that the analysis efficiency is also affected by the data structure and function call relationships within the target binary. Additionally, the majority of binaries comprise less than 800 functions with a size under 300KB, and can be analyzed in less than one hour.

**Comparison.** We compared FITS with BootStomp, which has a taint source inference module for stripped bootloaders. Since the taint source inference module only supports ARM architecture firmware and works as IDA Pro scripts, we evaluated BootStomp on 33 ARM firmware samples in our dataset. The result shows that BootStomp cannot find any taint source in these firmware samples. This is because BootStomp relies on keyword matching in strings processed by different functions, but the semantics of
We applied vulnerabilities (identify sinks. Determining Sinks. Following how Karonte identifies sinks, we may remain undetected in the firmware. To address this issue, we developed a static taint analysis engine called STA (as described §3.4). To setup STA, we used CTSs (consistent with Karonte) as taint sources for analysis. Ultimately, STA issued 202 alerts, of which 46 were verified to be bugs, resulting in a false positive rate of 77.2%. Compared to Karonte’s 35.6% false positive rate, the false positive rate of STA increased due to the inaccuracy of data flow recovery, especially in NETGEAR’s firmware. Nevertheless, STA found 9 bugs that Karonte could not. This is because STA can perform more data flow analysis that Karonte does not check due to time limitations on data flow analysis. Even without the time limitations, Karonte strings can vary or even be missing in different firmware, rendering this approach ineffective.

**Answer to RQ1:** With an 89% top-3 precision rate and an average analysis time of under three hours, FITS can accurately infer ITSs. On the other hand, BootStomp failed to identify any taint sources in the firmware dataset. The results demonstrate the efficacy of FITS.

### 4.3 Vulnerability Discovery (RQ2)

We applied FITS to two taint analysis engines, Karonte and STA, to demonstrate how ITSs may help improve the vulnerability discovery capability in firmware.

**Determining Sinks.** Following how Karonte identifies sinks, we use risky library functions as sinks that can induce buffer overflow vulnerabilities (e.g., `strncpy`, `sprintf`, `strncat`) or command hijacking vulnerabilities (e.g., `system`, `execve`). Specifically, we scan the binary instructions of the firmware, and find all function calls that call any of the risky library functions; these function calls are labeled as sinks. A taint analysis engine reports an alert if unsanitized user data flows to these sinks without sanitization. For experiments with Karonte, we directly used Karonte’s own sink identification method; in STA, we implemented the same logic to identify sinks.

**Karonte vs. STA.** In our experiment, we implemented Karonte-ITS by integrating the inferred ITSs from FITS to Karonte, and then conducted taint analysis. Our results, as shown in Table 5, demonstrate that Karonte-ITS identified 15 more distinct bugs (true positives) than Karonte. Furthermore, all bugs detected by Karonte were also detected by Karonte-ITS, indicating a lower false negative rate of Karonte-ITS than Karonte. These newly discovered bugs are because with ITSs Karonte-ITS identified and analyzed more data-flow paths between ITSs and sinks than those between CTSs and sinks in Karonte. As shown in Table 6, Karonte-ITS has a false positive rate of 34.7%, better than 35.6% of Karonte. Although Karonte-ITS takes longer on average to analyze data flow due to the increased number of taint sources involved, this additional overhead is reasonable. Karonte’s taint engine relies on symbolic execution, so as the number of data flow increases, the analysis time also rises significantly. However, the average analysis time per path is reduced by roughly 20% because the inferred ITSs by FITS shorten the distance of some data flow.

#### Table 5: Bug finding results.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Karonte</th>
<th>Karonte-ITS</th>
<th>STA</th>
<th>STA-ITS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Alerts</td>
<td># Bugs (hh:mm)</td>
<td># Alerts</td>
<td># Bugs (hh:mm)</td>
</tr>
<tr>
<td>NETGEAR R/XR/WNR</td>
<td>17</td>
<td>36</td>
<td>23</td>
<td>17:13</td>
</tr>
<tr>
<td>D-Link DIR/DWR/DCS</td>
<td>9</td>
<td>24</td>
<td>15</td>
<td>14:09</td>
</tr>
<tr>
<td>TP-Link TD/WR/TX/KC</td>
<td>16</td>
<td>2</td>
<td>2</td>
<td>1:30</td>
</tr>
<tr>
<td>Tenda AC/WH/HH</td>
<td>7</td>
<td>12</td>
<td>6</td>
<td>1:01</td>
</tr>
<tr>
<td>Latest Firmware</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NETGEAR R/XR/WNR</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3:10</td>
</tr>
<tr>
<td>D-Link DIR/DAP</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0:20</td>
</tr>
<tr>
<td>TP-Link WR/AP</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>7:02</td>
</tr>
<tr>
<td>Tenda AC/G</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2:20</td>
</tr>
<tr>
<td>Cisco RV</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6:45</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>-</td>
<td>59</td>
<td>73</td>
</tr>
</tbody>
</table>

![Figure 4: Time overhead.](image)

#### Table 6: False positive rates of taint analysis techniques.

<table>
<thead>
<tr>
<th></th>
<th>Karonte</th>
<th>Karonte-ITS</th>
<th>STA</th>
<th>STA-ITS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>35.6%</td>
<td>34.7%</td>
<td>77.2%</td>
<td>27.9%</td>
</tr>
</tbody>
</table>

Karonte has limitation in finding vulnerabilities in firmware. The root cause is that its reliance on symbolic execution, which requires the analysis time of each data flow to be specified to prevent memory or path explosion. As a result, the taint analysis of some data flow may be incomplete, and some bugs may remain undetected in the firmware.
would still be unable to find these bugs due to the problem of path explosion.

**STA vs. STA-ITS.** Similarly, we set up STA-ITS by integrating ITSs to STA. During the experiment, we discovered that ITSs not only manipulated user data but also some system data (e.g., subnet mask, MAC address, IP address), which leads to false positives. Drawing on the idea of Karonte, we filtered out some false positives by matching the relevant strings on the path that triggers the alerts. After filtering out 69 false positives, STA-ITS issued 534 alerts, of which 385 were confirmed as bugs. STA-ITS found 339 more bugs than STA, including all bugs found by STA, showing fewer false negatives of STA-ITS than STA. Compared to STA, due to the shorter distance between taint sources and sinks, the difficulty of data flow recovery is reduced and more true positives are found. As shown in Table 6, the false positive rate of STA-ITS is 27.9%, significantly lower than 77.2% of STA. The reduction in false positive rate is also due to the shortened distance between taint sources and sinks, which avoids analyzing long, complex data/control dependencies and reporting infeasible code paths.

The discovered bugs are mainly caused by data being used directly for dangerous operations, such as memory copying without sanitizing. We responsibly disclosed the 141 bugs of the latest firmware samples to the corresponding vendor. As of now, 67 bugs have been confirmed, and 21 of them have been awarded CVE IDs due to their critical threat level.

**Case Study.** To illustrate the advantages of inferred ITSs in actual taint analysis of firmware, we take the CVE-2022-20825 as an example. The vulnerability exposes multiple devices to the risk of remote unauthorized arbitrary code execution attacks, with the CVSS score of 9.8, and has been reported by the media due to its seriousness. After conducting reverse analysis and dynamic debugging of the firmware, we identified the library function BIO_read as the source of dangerous data. Reaching the sink (strcpy) from BIO_read requires at least 11 custom function calls and more than 50 library function calls. In real data flow analysis, this is more complicated by the need to search through larger code paths, and the analysis also involves alias analysis and at least three indirect calls. While value set analysis and symbolic execution can alleviate these problems, they may introduce many false negatives and significant analysis overhead. For example, without limiting the analysis time, Karonte can only analyze function calls with a depth of 4 after 24 hours. Moreover, the problem of memory and path explosion makes it challenging to continue the analysis. However, using the ITS data as the starting point for analysis requires only 2 function calls to reach the sink, greatly reducing the data flow analysis overhead and difficulty.

4.4 Efficacy of the BFV Representation (RQ3)

To evaluate the efficacy of BFV, we first perform ablation study of BFV, and then compare BFV with the state-of-the-art binary code representations.

**Ablation Study.** To assess the necessity and importance of different features that comprise the BFV, we created 11 variants of BFV, and each variant is a 10-dimensional feature representation for ITSs inference by removing one feature from BFV. For example, after removing the first feature in Table 1, the representation consisting of the remaining features is referred to as CF-1 (short for Cancel Feature-1). Figure 5 shows the results of this ablation study: BFV outperforms all other feature representations in terms of top-1 to top-3 precision. In other words, each feature in BFV is necessary, important and contributes positively to the inference precision of ITSs. For example, the data of CF-3 in Figure 5 shows that without the feature ‘number of callers’, the top-1 and top-2 precision of ITS drops to zero.

To illustrate whether a single feature is sufficient to infer ITSs, we perform ITS inference based on each individual feature in Table 1. For numerical features, we took the top-5 results for verification. The results show that only the ‘number of callers’ feature contributes to infer ITSs, and the top-3 achieved a precision rate of 21%. For boolean features, we analyzed the custom functions corresponding to the feature value true. However, the results show that it is challenging or even meaningless to verify functions that match a certain feature as ITSs. For example, when using only the ‘whether parameters passed to anchor functions’ feature, the average number of functions matching this feature in a binary is 105, and the maximum number reaches 349. We have to analyze such a large number of functions to find possibly one or two ITSs. In addition, this method increases the false negative rate of ITSs. For example, for the feature ‘whether parameters control conditional branches’, the correct ITSs were distributed in both false or true results.

---

**Answer to RQ2:** The ITSs inferred by FITS can help taint analysis engine find more bugs with fewer false positives. By integrating ITSs to the static taint analysis engine STA, 339 more bugs were found, 21 of which were awarded CVE IDs and rated high severity.
Table 7: Inference results based on different representations.

<table>
<thead>
<tr>
<th></th>
<th>Augmented-CFG</th>
<th>Attributed-CFG</th>
<th>BFV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-1</td>
<td>6%</td>
<td>0%</td>
<td>47%</td>
</tr>
<tr>
<td>Top-2</td>
<td>5%</td>
<td>0%</td>
<td>63%</td>
</tr>
<tr>
<td>Top-3</td>
<td>10%</td>
<td>1%</td>
<td>89%</td>
</tr>
</tbody>
</table>

Comparison with Other Binary Code Representations. To further evaluate the effectiveness of BFV, we compared it with other representations in binary code summarization and similarity analysis, including Augmented-CFG and Attributed-CFG. Table 7 shows that ITS inference with BFV achieves 89% top-3 precision, while ITS inference with Augmented-CFG and Attributed-CFG only has less than 10% top-3 precision. We further investigated the poor performance of existing code representations in inferring ITS and summarized the following two reasons: (i) they only focus on code structure features, which are helpful for identifying similarities in the code-level but are insufficient for identifying behavior-level. (ii) their similarity recognition methods rely on a large amount of labeled data and complex network structures, which are not suitable for unsupervised learning. Based on the comparison results, we conclude that BFV can better represent the behavioral features of functions by extracting structural features and flow features.

Answer to RQ3: BFV-based ITS inference outperforms other subset feature combinations, indicating that various BFV features are necessary for ITS inference. Moreover, compared to Augmented-CFG and Attributed-CFG, ITS inference using BFV can achieve higher precision in stripped embedded firmware.

4.5 Efficacy of Inference Strategy (RQ4)

To illustrate the necessity of the behavior clustering stage in ITS inference. We used anchor functions to score custom functions directly. In 59 firmware samples top-1, top-2 and top-3 precision are about 5%, 5%, and 7% respectively. We analyzed the results obtained by inference and found that the top-ranked functions are often dominated by a certain dimension feature, such as the number of calls and basic blocks. The dominant features in different firmware are different. For comparison, we also processed BFV using principal component analysis, standardization, and normalization before scoring. Based on them, the top-3 precision values are no more than 10%. In addition, we further analyzed whether cluster-based filtering introduces false negatives. By manually inspecting the filtered functions, we found that they cannot be ITSs.

Table 8: Inference results based on different scoring methods.

<table>
<thead>
<tr>
<th></th>
<th>Euclidean</th>
<th>Manhattan</th>
<th>Pearson</th>
<th>Cosine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-1</td>
<td>15%</td>
<td>20%</td>
<td>34%</td>
<td>47%</td>
</tr>
<tr>
<td>Top-2</td>
<td>25%</td>
<td>25%</td>
<td>50%</td>
<td>63%</td>
</tr>
<tr>
<td>Top-3</td>
<td>49%</td>
<td>44%</td>
<td>57%</td>
<td>89%</td>
</tr>
</tbody>
</table>

To demonstrate the performance of scoring with Cosine distance, we compared it with other distance metrics, including Euclidean distance, Manhattan distance, and Pearson correlation coefficient. Table 8 presents the precision of inference results based on different methods. We can find that Cosine distance is more suitable for capturing the relative distance of vectors in high dimensional space.

Answer to RQ4: The clustering stage effectively reduces the false positives caused by the feature dimensions on the score. Moreover, Cosine distance is more suitable for scoring than other similarity algorithms.

5 DISCUSSION

IoT Devices. The IoT devices evaluated by FITS are directly connected to the Internet. However, there are many other types of IoT devices, such as smart plugs, smart cleaning robots, programmable logic controllers, which work in local area networks. These devices use Zigbee, Bluetooth, Can Bus and other proximity communication protocols to complete cooperation. These devices typically have high real-time requirements or a simple functional design. FITS does not infer an effective ITS to aid in taint analysis for these devices. The reasons are as follows: First, the program processing logic and data indexing are simple to ensure real-time performance. Functions directly read data in global variables using offsets or obtain data through a fixed memory address, eliminating the need for an additional function to serve as the data index. Second, the communication protocols have the fixed formats, simple field designs, and limited functionality. When processing communication data, the function relies on the instruction format, i.e., it directly retrieves the corresponding field data using offsets for processing without indexing. Despite the absence of an ITS, the relatively small size and simple implementation of these devices’ programs make it feasible to perform analysis from classical taint sources to sinks [43].

Firmware Unpacking. As threats to IoT devices increase, vendors have recognized that firmware distribution is one of the critical reasons for security breaches. To protect their intellectual property from reverse engineering attacks, some vendors encrypt or encode their firmware. However, these measures also limit the security analysis of firmware by third-party testers. While this paper mitigates popular encrypted firmware, vendors can easily release new firmware versions by replacing algorithms, keys, and salts, which limits the capability of FITS. Nevertheless, if vendors use FITS, it can help discover vulnerabilities. Additionally, community enthusiasts or research work publish new firmware unpacking methods to facilitate analysis [19, 39].

Application. Third-party security analysis for stripped binary is a common application scenario [23–25, 28, 37, 44, 47, 63], given that third-party security researchers often lack access to the source code of target applications. Therefore, the primary aim of FITS is to assist these researchers in effectively detecting vulnerabilities in stripped binaries. However, vendors who have access to the source code can leverage more semantic information, such as function names, to improve the performance of FITS. ITSs can help find more vulnerabilities than CTSs and reduce the analysis time of large-scale code. Apart from taint source identification, we find
that high-scoring custom functions tend to have sensitive operations, such as file writing and operation selection. These functions cannot be used as taint sources, but analyzing these functions first is more conducive to sorting out the code logic, rather than aimlessly analyzing from the main function. Moreover, FITS can assist in analyzing malware, which is the stripped binary. For example, malware often requires the control of remote or local instructions to trigger malicious behavior. High-scoring functions help detect critical operations in malware.

**Vulnerability Mitigation.** There are techniques to mitigate vulnerabilities, but they cannot completely avoid vulnerabilities. First, code-level vulnerability mitigation techniques, such as replacing unsafe library functions with safer library functions with boundary checks (e.g., replacing strcpy with strncpy), are not effective for all types of vulnerabilities [6]. For example, there is no general mitigation method for command hijacking vulnerabilities; we need to add a domain-specific sanitizer to sanitize the command from user input. Even with the safer library functions, developers can still make mistakes and introduce vulnerabilities. For example, the case study in §4.3, the CVE-2022-20825 is induced by a function call to strncpy. Second, system-level vulnerability mitigation techniques, such as Address Space Layout Randomization (ASLR) and No-Executive (NX), have performance overhead issues that make them impractical for some application scenarios such as the IoT devices [62]. On the other hand, testing— an effective vulnerability mitigation technique—can complement the mitigation techniques above [49]. Taint analysis is such a static testing technique, and ITTs improve taint analysis in efficacy and efficiency of finding vulnerabilities in stripped binaries. Especially, ITTs have found vulnerabilities that cannot be mitigated by either the code-level or system-level mitigation techniques.

### 6 RELATED WORK

#### 6.1 Taint Source Inference

Effective taint analysis depends on identifying the appropriate taint source. However, stripped binaries lack the high-level semantics available in Java programs that can aid in determining taint sources. While methods like SUSI [42] can assist in identifying taint sources in Java, they are not suitable for firmware, as their debug information is often stripped and cannot rely on semantic information. Nero [17], Debin [24], and Nomine [4] attempt to restore or re-name the semantic information of stripped binaries. However, these methods cannot be used in real programs due to the limitation of dataset size, program size and accuracy rate, making them unsuitable for real programs. Existing taint analysis efforts for firmware rely on manual work [12] or heuristic rules [9, 11, 43, 44], limiting the performance of taint analysis in firmware. The FITS approach overcomes these limitations by enabling semantic-agnostic ITTs inference through the analysis of function behavioral similarity and the scoring of custom functions with anchor functions. This approach significantly improves the performance of firmware taint analysis.

#### 6.2 Binary Similarity Analysis

Binary similarity analysis techniques [18, 21, 34, 58, 60] have shown promising results in vulnerability association and patch identification, as they use lexical and syntactic analysis in combination with deep learning to determine the similarity between functions. These techniques primarily address code-level nuances introduced by architecture, compiler, and compilation optimizations. However, for taint source inference, functions that behave similarly may have completely different code-level implementations. Moreover, existing similarity analysis techniques require a sufficient number of labeled samples, which is unattainable for heterogeneous and closed-source firmware. To this end, FITS extracts structure and flow features through program analysis to characterize the static and dynamic properties of functions. It then measures the feature representation to identify functions that behave similarly, making it an effective approach for inferring ITTs in stripped firmware binaries.

#### 6.3 Firmware Vulnerability Analysis

Vulnerability analysis for IoT device firmware can be divided into static and dynamic.

Static vulnerability analysis refers to the completion of vulnerability discovery through code analysis without the specific execution of the program. Taint analysis has been proven effective in identifying vulnerabilities in firmware [9, 11, 12, 43, 44], and existing techniques such as value set analysis and variable structure matching have been used to address the problem of false negatives caused by incomplete data flow. However, real-world IoT firmware is often complex and bulky, making it challenging to perform accurate and comprehensive analysis of indirect calls and aliases, resulting in a significant analysis overhead. The introduction of ITTs, as demonstrated by the evaluation of FITS, has shown to enhance the effectiveness of real-world stripped firmware taint analysis. By using FITS to infer taint sources and focusing on custom functions with high scores, ITTs can reduce analysis overhead and improve the accuracy of static vulnerability analysis in IoT device firmware.

Dynamic vulnerability analysis is another approach to discovering vulnerabilities, and it involves executing test cases and monitoring the program’s behavior. Two primary methods for testing IoT devices are real-device-based and firmware-rehosting-based methods. Real-device-based fuzzing techniques, such as IoTFuzzer [8], Snipuzz [22], and PCFuzzzer [35], have proven to be effective. However, these methods can be expensive and have limited coverage due to the need for physical devices. Additionally, black box testing often leads to inefficient testing and makes it difficult to identify exceptions. On the other hand, firmware-rehosting techniques, FrimAFL [65], P2IM [20], and HALucinator [13], offer a less expensive way to test firmware. However, these methods have lower fidelity and can lead to shallow testing and unreproducible results. Furthermore, firmware-rehosting techniques require manual operation, which makes them less scalable. Therefore, static analysis technology, such as FITS, is more suitable for scalable vulnerability analysis of IoT firmware.
7 CONCLUSION
In this paper, we present a novel approach to address the challenges in performing effective data flow analysis on stripped firmware of IoT devices. We introduce the concept of ITSs and propose the first methodology, FITS, to automatically discover ITSs from stripped firmware. To achieve this, FITS employs static analysis to extract structural and flow features of functions, which serve as behavioral representations to characterize their static and dynamic properties. Using these representations, FITS ranks custom functions as ITSs by leveraging behavior clustering and anchor function scoring. By doing so, FITS shortens the path of data flow analysis, improving the performance of firmware taint analysis.

Our comprehensive evaluations on 59 real-world firmware demonstrate the effectiveness of FITS in inferring ITSs, with a top-3 precision of 89%. More importantly, ITSs helped Karonte find 15 more bugs and the static taint analysis engine find 339 more bugs. As of now, 67 of the 141 bugs found in the latest firmware version have been confirmed, of which 21 have been awarded CVE IDs and rated as high severity. These results demonstrate the potential of FITS in improving the security of IoT devices.

8 ACKNOWLEDGEMENTS
We thank the anonymous reviewers for their valuable comments to improve our paper. This work was supported by the the National Key R&D Program of China (No. 2022YFC330103) and the National Natural Science Foundation of China (No. 61931019 and No.62002342). Moreover, we would like to thank Cisco, D-Link, TP-Link and MITRE for their prompt confirmation of our reported vulnerabilities; we especially appreciate Cisco’s feedback on the vulnerabilities, including analyzing the threats and quickly assigning multiple CVE IDs.

A VERIFYING INFERRED ITSS AND IDENTIFYING TAINT ORIGINS
A function in the inferred result of FITS is considered an ITS if it extracts and returns a part of the user input. Therefore, during the verification process, our main concern is whether the function takes part of the data from the memory region storing user input (passed in through a parameter register or a global variable) and return the fetched data (output through a parameter/return register or a global variable). When an ITS is confirmed, the corresponding register or global variable of the returned data is used as the taint origin. Specifically, we use the following three methods to verify according to the actual situation of different firmware.

Firmware rehosting. Existing work such as Firmdyne [7] and FirmAE [29] propose rehosting embedded device firmware on the computer. This convenient and reliable verification method allows us to debug programs in firmware through emulation dynamically. For example, we can determine an ITS by setting breakpoints in the target function and observing the values in registers or memory region. However, existing technologies cannot support all firmware. In addition, successfully rehosted firmware is not necessarily high fidelity. That is, it crashes during debugging and verification. In the end, 11 firmware were verified this way.

Real device debugging. Some devices support enabling Telnet and SSH services. We can debug remotely by passing the debug server to the device. Therefore, we can obtain program dynamic running information through breakpoint debugging to determine ITSs and the corresponding taint origin. However, this method requires purchasing the real device, which is an expensive overhead. Moreover, many vendors do not allow users to connect to the device terminal. We verified 2 firmware samples by this means.

Semantic information analysis based on relevant version firmware. The historical version of firmware often does not strip debug information and symbol tables. There are differences between versions due to patches and upgrades, but the overall workflow is similar. We can match some of the same functions through similarity analysis to help reverse engineering of firmware [66]. Then we manually analyze the objective function and its context logic to clarify the data processing flow. At the same time, combined with the previous taint analysis work [12], we verified 46 firmware samples and identified the taint origin.

B ARTIFACT APPENDIX
B.1 Abstract
This artifact provides the program of FITS, which is the code related to our proposed behavior representation extraction and similarity analysis for finding ITSs. The artifact is packaged as a Docker image for standard X86 machines.

B.2 Artifact check-list
- Data set: Binaries for ITS inference in 59 real firmwares are included.
- Run-time environment: Docker.
- Hardware: X86
- Output: Function ranking of candidate ITSs.
- Experiments: Write a configuration file to run with Python code.
- How much disk space required (approximately)?: 15GB.
- How much time is needed to prepare workflow (approximately)?: 10 minutes.
- How much time is needed to complete experiments (approximately)?: Depending on the complexity of the binary the time can range from a few minutes to a dozen hours.
- Publicly available?: Yes
- Archived DOI: 10.5281/zenodo.8376901

B.3 Description
B.3.1 How to Access. The artifact can be downloaded from [36].
B.3.2 Hardware dependencies. A standard X86 machine.
B.3.3 Software dependencies. Docker.

B.4 Installation
Please use the following command to install the artifact.
unzip fits-artifact.zip
docker load –input fits-artifact.tar
docker run -it fits-artifact
B.5 Basic Test
Please use the following command to test the artifact:

```bash
cd app/FITS
conda activate FITS
python3 fits.py config/config_karonte/tmplink/TPLINK_chttp_djson
```

B.6 Experiment workflow
The workflow is as follows, see [36] for details.
(1) Write a config file.
(2) The config file as a parameter to run FITS.py.

B.7 Evaluation and expected results
After the FITS operation is completed, the generated results are in the real directory. The result shows the top 1-4 rankings in the result directory. Moreover, the BFV of functions is saved in the feature directory. Understanding the mirai botnet.


REFERENCES

[16] Malwarebytes Labs. 150,000 verkada security cameras hacked—tomeak a point.


