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.

*Corresponding Author

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:
Puzhuo Liu, Yaowen Zheng, Chengnian Sun, Chuan Qin, Dongliang Fang, Mingdong Liu, and Limin Sun. 2023. FITS: Inferring Intermediate Taint Sources for Effective Vulnerability Analysis of IoT Device Firmware. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. 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 have 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 services for 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.[1] 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.

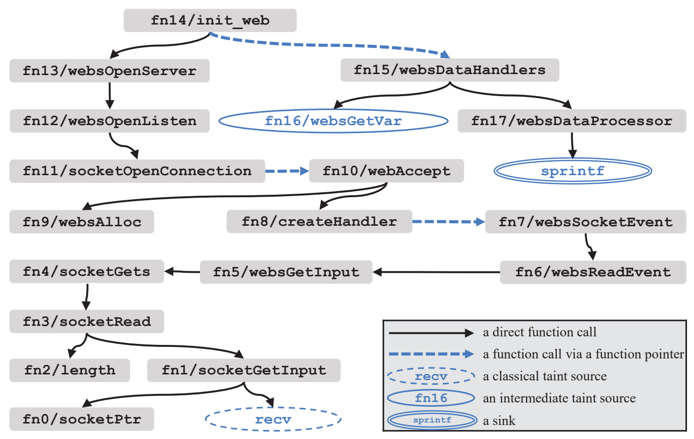
Challenges. However, performing accurate data flow analysis from taint sources to sinks has always been a challenging problem for applying taint analysis on firmware, even with the state-of-the-art techniques [9, 12, 44]. The reason is manifold. First, firmware consists of multiple binaries, and user 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 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`, `memcmp`, and `strstr`, 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.



(a) **An example of function call graph for dangerous data between `recv` and `sprintf`.**

```
char *fn16(char* index, char* src_addr, int length) {
  if (index) {
    int v1 = strlen(index);
    int i = 0;
    while (i < length) {
      if (!*(src_addr + i)) return -1;
      if (!strncmp(index, src_addr + i, v1)) break;
      i++;
    }
    if (i < length) {
      int v2 = strlen(*(src_addr + i));
      char* v3 = (char*) malloc(v1 + v2);
      if (!v3) return -1;
      memcpy(v3, arg1, v2);
      memcpy(&v3[v1], arg2 + i, v2);
      return v3; // used later by other functions
    }
  }
  return 0;
}
```

(b) **An example of intermediate taint sources.**

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.

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 (in**F**erring **I**ntermediate **T**aint **S**ources), 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 performs 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 magnitude 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 candidates 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 to 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 devices.
- We propose the first methodology FITS to automatically, accurately 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 significantly 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 efficiency 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 vulnerabilities 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

```
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 && *(char *)str1 == *(char *)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(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 R7000P.[2] 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 `fn15` 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 CTSs. 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 CTSs is the difficulty of computing the data flow from CTSs 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 CTS, 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

[2]The call graph is significantly simplified for illustration purpose. At the same time, we add easy-to-understand function names that do not exist during the reverse process, except for `recv` and `sprintf`.

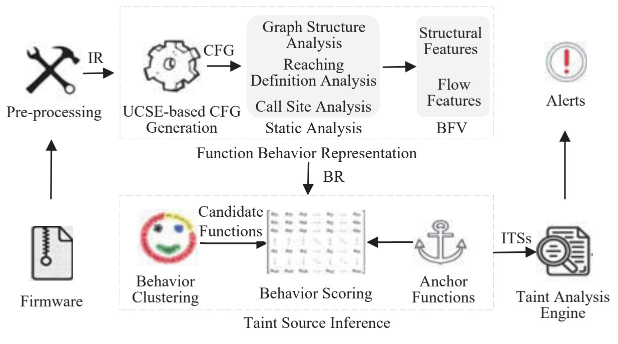


Figure 3: Workflow of FITS.

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 (IR) 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 (UCSE) [41] technique 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 data flow 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.

Category	Feature Name
Structural Feature (SF)	1. number of basic blocks
	2. existence of loops
	3. number of callers
	4. number of parameters
	5. number of call anchor functions
	6. number of call library functions
Flow Feature (FF)	7. whether parameters control loops
	8. whether parameters control conditional branches
	9. whether parameters passed to anchor functions
	10. whether arguments contain strings
	11. number of different strings in all call sites

Algorithm 1: Compute behavioral representations.

```
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      $cfg_{Fn}, cg_{Fn}$ ← UCSE-based analysis on Bin, Libs, and Fn
3  for each function Fn in Bin do
4      Initialize SF, $FF_{intra}$, $FF_{inter}$ lists with zero or False
5      SF ← structural analysis on $cfg_{Fn}$ and $cg_{Fn}$
6      vars, params ← identify key vars and params in $cfg_{Fn}$
7      rdefs ← reaching definition analysis on $cfg_{Fn}$ and params
8      $FF_{intra}$ ← use vars to match rdefs
9      for each caller in Fn do
10         $args_{caller}$ ← call site analysis on $cfg_{caller}$ and Fn
11         $strs_{caller}$ ← backtrack memory on $cfg_{caller}$ and args
12     $FF_{inter}$ ← analysis $strs_{caller}$ of each caller
       // The operator + represents the operation of list concatenation.
13     BFV ← SF + $FF_{intra}$ + $FF_{inter}$
14     BR[Fn] = BFV
15 return BR
```

3.2 Function Behavior Representation

We propose BFV to capture the behavioral characteristics of functions to identify ITSs. Function behavior is generally reflected in static and dynamic properties. Function static properties, such as the number of basic blocks and the existence of loops, do not change *w.r.t.* function input. Dynamic properties mainly refer to the process of processing input at runtime by a function such as whether the input affects branches. The difficulty of extracting these properties varies. We divide them into structural features (SF) and flow features (FF) for extraction. The overall algorithm for behavior representation extraction is shown in Algorithm 1.

Structural Features. For a function, structural features include the number of basic blocks, the existence of loops, *etc.*. These features can be obtained by statically analyzing the CFG of Fn. Symbolic execution can build accurate CFGs (*e.g.*, path-sensitve,

Table 2: Backtracking principle.

IR expression	Description
$PUT(r_i) = t_i$ or constant	Assign $t_i/constant$ to r_i
$t_i = GET(r_j)$	Assign the value from r_j to t_i
$t_i = Binop(t_n, t_m \text{ or } constant)$	Assign the calculation result of t_n and $t_m/constant$ to t_i
$t_i = Load(t_j)$	Assign the value loaded from t_j to t_i
$Store(t_j) = t_i$	Store t_i to the memory at t_j

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 data flow 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 function parameters 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

Algorithm 2: Infer intermediate taint sources.

```
   Input: BR: the dictionary containing all functions BFV
   Output: Rank: the rank of custom functions as ITSs
1  candidates ← ∅
2  classes ← cluster analysis on BR[custom function]
3  for each class in classes do
4      $C_{class}$ ← calculate the complexity of the class
5  $C_{average}$ ← compute the average complexity of all classes
6  for each class in classes do
7      if $C_{class} > C_{average}$ then
8          candidates ← candidates ∪ {functions in class}
9  for each function Fn ∈ candidates do
10     $S_{Fn}$ ← score BR[Fn] based on BR[anchor function]
11 Rank ← sort functions according to $S_{Fn}$
12 return Rank
```

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 DBSACN [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.

$$C_{class} = \frac{\sum_{i=0}^{i=N} [\mathrm{bb}_i + \mathrm{caller}_i + \mathrm{lib}_i + \mathrm{anchor}_i]}{N} \tag{1}$$

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.

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

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.

Dataset	Vendor	Series	# Firmware	Top-1	Top-2	Top-3	Avg time (hh:mm)
Karonte Firmware	NETGEAR	R/XR/WNR	17	71%	100%	100%	9:11
	D-Link	DIR/DWR/DCS	9	33%	33%	78%	0:37
	TP-Link	TD/WA/WR/TX/KC	16	36%	63%	81%	2:13
	Tenda	AC/WH/FH	7	43%	43%	86%	2:13
Latest Firmware	NETGEAR	R/WNR	2	100%	100%	100%	8:57
	D-Link	DIR/DAP	3	33%	33%	100%	0:25
	TP-Link	WR/AP	2	0%	0%	100%	1:21
	Tenda	AC/G	2	50%	50%	100%	1:25
	Cisco	RV	1	0%	0%	100%	10:27
Average	-	-	-	47%	63%	89%	2:57

Table 4: Partial ITS inference results.

Vendor	Firmware	Binary	# Functions	ITS addr.	Ranking
NETGEAR	R7800-V1.0.2.32	httpd	2207	0x19090	1
NETGEAR	AC1450-V1.0.0.36	httpd	1540	0x1654c	2
NETGEAR	R7000P-V1.3.0.8	httpd	2207	0x19090	1
NETGEAR	R8900-V1.0.2.40	netcgi	1248	0x2a228	1
D-Link	DIR826LA1_FW105B13	miniupnpd	456	0x412240	3
D-Link	DAP1860A1_FW104B05	uhttpd	1280	0x417574	2
D-Link	DIR1960A1_FW111B03	prog.cgi	1341	0x41be3c	1
TP-Link	AP500(US)_V1_180320	httpd	1919	0x4ed64	1
TP-Link	C2v1_0.9.1_5.0	httpd	268	0x404098	1
TP-Link	W8968v4_un_1_0_5	httpd	1145	0x41b278	3
Tenda	US_G3V3.0br_V15.11.0.6	httpd	1978	0x1c634	3
Tenda	AC9V1.0BR_V15.03.05.14	httpd	1507	0x2b9fc	3
Tenda	FH1201V1.0BR_V1.2.0.8	httpd	1142	0x432a84	1
Cisco	RV130X_FW_1.0.3.55	httpd	1338	0x1d210	3

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

Table 5: Bug finding results.

Dataset				Karonte			Karonte-ITS			STA			STA-ITS		
				# Alerts	# Bugs	Avg time (hh:mm)	# Alerts	# Bugs	Avg time (hh:mm)	# Alerts	# Bugs	Avg time (hh:mm)	# Alerts	# Bugs	Avg time (hh:mm)
Karonte Firmware	NETGEAR	R/XR/WNR	17	36	23	17:13	39	26	17:21	165	15	0:04	168	132	0:06
	D-Link	DIR/DWR/DCS	9	24	15	14:09	29	18	14:58	19	15	0:02	42	22	0:03
	TP-Link	TD/WA/WR/TX/KC	16	2	2	1:30	2	2	2:11	0	0	0:03	17	11	0:05
	Tenda	AC/WH/FH	7	12	6	1:01	15	9	1:32	8	8	0:03	112	79	0:04
Latest Firmware	NETGEAR	R/WNR	2	1	1	3:10	3	3	4:09	5	3	0:01	52	36	0:03
	D-Link	DIR/DAP	3	0	0	0:20	0	0	0:20	3	3	0:02	18	13	0:03
	TP-Link	WR/AP	2	0	0	7:02	2	1	6:57	0	0	0:01	13	8	0:02
	Tenda	AC/G	2	0	0	2:20	4	2	5:30	2	2	0:01	47	43	0:02
	Cisco	RV	1	0	0	6:45	1	1	7:17	0	0	0:02	65	41	0:02
Total	-	-	59	73	47	-	95	62	-	202	46	-	534	385	-

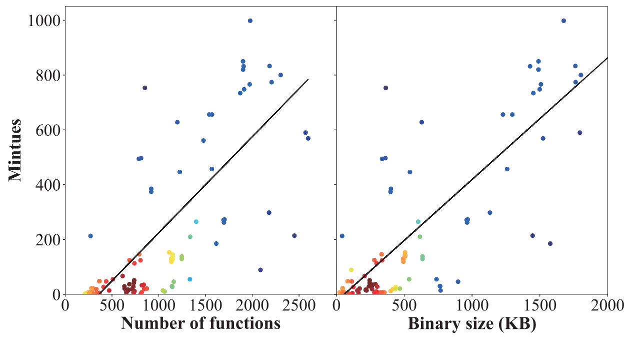


Figure 4: Time overhead.

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 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. Karonte-ITS. 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 6: False positive rates of taint analysis techniques.

Karonte	Karonte-ITS	STA	STA-ITS
35.6%	34.7%	77.2%	27.9%

Karonte vs. STA. 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. 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 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 [27]. 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 (`strncpy`) 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 [64] and symbolic execution [44] 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 `0x1d210` as the starting point for analysis requires only 2 function calls to reach the sink, greatly reducing the data flow analysis overhead and difficulty.

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.

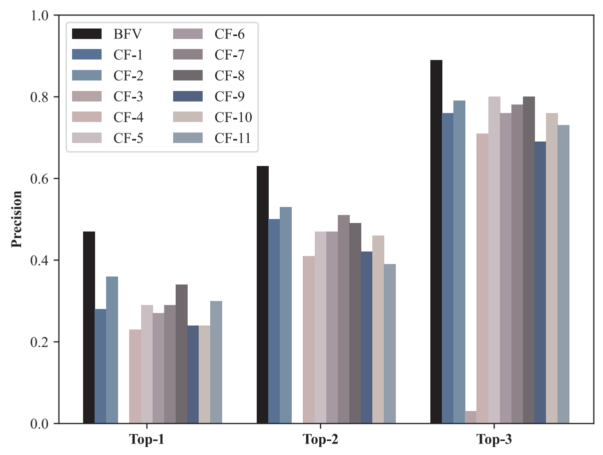


Figure 5: BFV compared with other representations that cancel one-dimensional features.

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 FITS. 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 FITS 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-3 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.

Table 7: Inference results based on different representations.

	Augmented-CFG	Attributed-CFG	BFV
Top-1	0%	0%	47%
Top-2	5%	0%	63%
Top-3	10%	1%	89%

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.

	Euclidean	Manhattan	Pearson	Cosine
Top-1	15%	20%	34%	47%
Top-2	25%	25%	50%	63%
Top-3	49%	44%	57%	89%

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-EXecute (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 ITSs improve taint analysis in efficacy and efficiency of finding vulnerabilities in stripped binaries. Especially, ITSs 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 rename 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 ITSs 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 ITSs 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 ITSs, 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, ITSs 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 PCFuzzer [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. 2022YFC3301103) 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, and its data is marked with a taint label. 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.

```
cd app/FITS
conda activate FITS
python3 fits.pyc config/config_karonte/tplink/TPLINK_c2http
d.json
```

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.pyc.

B.7 Evaluation and expected results

After the FITS operation is completed, the generated results are in the result directory. Moreover, the BFV of functions is saved in the feature directory. The inference result shows the top 1-4 rankings possible as an ITS. Manual verification is then required to confirm the real ITS from the possible ITSs.

REFERENCES

[1] Kapil Anand, Khaled Elwazeer, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos Keromytis. An accurate stack memory abstraction and symbolic analysis framework for executables. In *2013 IEEE International Conference on Software Maintenance*, pages 90–99. IEEE, 2013.

[2] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, and Michalis Kallitsis. Understanding the mirai botnet. In *26th USENIX security symposium*, pages 1093–1110, 2017.

[3] Pieter Arntz. Threat spotlight: Wastedlocker, customized ransomware. https://blog.malwarebytes.com/threat-spotlight/2020/07/threat-spotlight-wastedlocker-customized-ransomware, 2020. Accessed 2022-9-10.

[4] Fiorella Artuso, Giuseppe Antonio Di Luna, Luca Massarelli, and Leonardo Querzoni. In nomine function: Naming functions in stripped binaries with neural networks. *arXiv preprint arXiv:1912.07946*, 2019.

[5] Gogul Balakrishnan, Thomas Reps, David Melski, and Tim Teitelbaum. Wysinwyx: What you see is not what you execute. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 202–213. Springer, 2005.

[6] Muhammad Arif Butt, Zarafshan Ajmal, Zafar Iqbal Khan, Muhammad Idrees, and Yasir Javed. An in-depth survey of bypassing buffer overflow mitigation techniques. *Applied Sciences*, 12(13):6702, 2022.

[7] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

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

[9] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 303–319, 2021.

[10] GUO Xiang-Ying MF CHEN Rui, YANG Meng-Fei. Interrupt data race detection based on shared variable access order pattern. *Journal of Software*, 27(3):547–561, 2016.

[11] Kai Cheng, Dongliang Fang, Chuan Qin, Huizhao Wang, Yaowen Zheng, Nan Yu, and Limin Sun. Automatic inference of taint sources to discover vulnerabilities in soho router firmware. In Audun Jøsang, Lynn Futcher, and Janne Hagen, editors, *ICT Systems Security and Privacy Protection*, pages 83–99, Cham, 2021. Springer International Publishing.

[12] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE, 2018.

[13] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. USENIX Association, August 2020.

[14] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. Pie: Parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 251–260, 2015.

[15] Preprocessing data. https://scikit-learn.org/stable/modules/preprocessing.html, 2022. Accessed 2022-10-10.

[16] NATIONAL VULNERABILITY DATABASE. https://nvd.nist.gov/, 2022. Accessed 2022-10-10.

[17] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[18] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, 2019.

[19] E-M-B-A. Emba. https://github.com/e-m-b-a/embak, 2022. Accessed 2022-9-10.

[20] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254, 2020.

[21] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[22] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 337–350, 2021.

[23] Fabio Gritti, Fabio Pagani, Ilya Grishchenko, Lukas Dresel, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. Heapster: Analyzing the security of dynamic allocators for monolithic firmware images. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1082–1099. IEEE, 2022.

[24] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[25] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. Rapidpatch: Firmware hotpatching for real-time embedded devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2225–2242, 2022.

[26] Hex-rays. Ida pro. https://www.hex-rays.com/ida-pro. Accessed 2022-9-10.

[27] Itnews. https://www.itnews.com.au/XXXXX, 2022. Accessed 2022-10-10.

[28] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1645, 2022.

[29] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Annual Computer Security Applications Conference*, pages 733–745, 2020.

[30] Eduard Kovacs. 70 percent of iot devices vulnerable to cyberattacks. https://www.securityweek.com/70-iot-devices-vulnerable-cyberattacks-hp, 2014. Accessed 2022-9-8.

[31] Malwarebytes Labsi. 150,000 verkada security cameras hacked—to make a point. https://blog.malwarebytes.com/iot/2021/03/150000-verkada-security-cameras-hacked-to-make-a-point, 2021. Accessed 2022-9-10.

[32] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security Privacy*, 9(3):49–51, 2011.

[33] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Annual Network and Distributed System Security Symposium, NDSS*, 2011.

[34] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678, 2018.

[35] Puzhuo Liu, Yaowen Zheng, Zhanwei Song, Dongliang Fang, Shichao Lv, and Limin Sun. Fuzzing proprietary protocols of programmable controllers to find vulnerabilities that affect physical control. *Journal of Systems Architecture*, 127:102483, 2022.

[36] Puzhuo Liu, Yaowen Zheng, Chengnian Sun, Chuan Qin, Dongliang Fang, Mingdong Liu, and Limin Sun. Fits. https://zenodo.org/record/8376901, 2023.

[37] Xiaozhu Meng and Weijie Liu. Incremental cfg patching for binary rewriting. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1020–1033, 2021.

[38] Palo Alto Networks. 2020 unit 42 iot threat report. https://iotbusinessnews.com/download/white-papers/UNIT42-IoT-Threat-Report.pdf, 2020. Accessed 2022-9-10.

[39] ONEKEY. How-to: Extracting decryption keys for d-link. https://onekey.com/blog/extracting-decryption-keys-dlink/, 2022. Accessed 2022-10-10.

[40] Pyvex. https://github.com/angr/pyvex, 2022. Accessed 2022-10-10.

[41] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., August 2015. USENIX Association.

[42] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Annual Network and Distributed System Security Symposium, NDSS*, volume 14, page 1125, 2014.

[43] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. BootStomp: On the security of bootloaders in mobile devices. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 781–798, Vancouver, BC, August 2017. USENIX Association.

[44] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561. IEEE, 2020.

[45] Dimensionality reduction. https://scikit-learn.org/stable/modules/decomposition.html, 2022. Accessed 2022-10-10.

[46] ReFirmLabs. Binwalk. https://github.com/ReFirmLabs/binwalk, 2022. Accessed 2022-9-10.

[47] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise mmio modeling for effective firmware fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1239–1256, 2022.

[48] scikit-learn developers. scikit-learn. https://scikit-learn.org/stable/, 2022. Accessed 2022-9-10.

[49] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, 44(3):1–46, 2012.

[50] Statista. Internet of things (iot). https://www.statista.com/topics/2637/internet-of-things, 2021. Accessed 2022-9-10.

[51] SwatiKhandelwal. Thousands of mikrotik routers hacked to eavesdrop on network traffic. https://thehackernews.com/2018/09/mikrotik-router-hacking.html, 2018. Accessed 2022-9-10.

[52] Global Offset Table. https://en.wikipedia.org/wiki/Global_Offset_Table, 2022. Accessed 2022-10-10.

[53] Boxiang Wang, Rui Chen, Chao Li, Tingting Yu, Dongdong Gao, and Mengfei Yang. Specchecker-isa: a data sharing analyzer for interrupt-driven embedded software. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 801–804, 2022.

[54] wikipedia. https://en.wikipedia.org/wiki/DBSCAN, 2022. Accessed 2022-10-10.

[55] wikipedia. https://en.wikipedia.org/wiki/Euclidean_distance, 2022. Accessed 2022-10-10.

[56] wikipedia. https://en.wiktionary.org/wiki/Manhattan_distance, 2022. Accessed 2022-10-10.

[57] wikipedia. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient, 2022. Accessed 2022-10-10.

[58] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[59] Audrey Dutcher Yan Shoshitaishvili, Ruoyu (Fish) Wang. Angr. https://angr.io/, 2022. Accessed 2022-9-10.

[60] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 224–236. IEEE, 2021.

[61] Yao Yao, Wei Zhou, Yan Jia, Lipeng Zhu, Peng Liu, and Yuqing Zhang. Identifying privilege separation vulnerabilities in iot firmware with symbolic execution. In *European Symposium on Research in Computer Security*, pages 638–657. Springer, 2019.

[62] Ruotong Yu, Francesca Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. Building embedded systems like it's 1996. In *Annual Network and Distributed System Security Symposium (NDSS 22)*, 2022.

[63] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676. IEEE, 2021.

[64] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–31, 2019.

[65] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl:high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[66] zynamics. Bindiff. https://www.zynamics.com/bindiff.html, 2022. Accessed 2023-2-14.