2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware

Kai Cheng*¶, Qiang Li†, Lei Wang ‡, Qian Chen*¶, Yaowen Zheng*¶, Limin Sun*¶, Zhenkai Liang§
† School of Computer and Information Technology, Beijing Jiaotong University, China
* School of Cyber Security, University of Chinese Academy of Sciences, China
¶ Institute of Information Engineering, Chinese Academy of Sciences, China
‡ State Grid Corporation of China
§ National University of Singapore, Singapore

***Abstract*—A rising number of embedded devices are reachable in the cyberspace, such as routers, cameras, printers, etc. Those devices usually run firmware whose code is proprietary with few public documents. Furthermore, most of the firmware images cannot be analyzed in dynamic analysis due to various hardware-specific peripherals. As a result, it hinders traditional static analysis and dynamic analysis techniques. In this paper, we propose a static binary analysis approach, DTaint, to detect taint-style vulnerabilities in the firmware. The taint-style vulnerability is a typical class of weakness, where the input data reaches a sensitive sink through an unsafe path. Specifically, we generate data dependency in a bottom-up manner through traversing callees before callers. To reduce the influence of the binary firmware, DTaint identifies pointer aliasing, interprocedural data flow, and similarity of the data structure layout. We have implemented a prototype of DTaint and conducted experiments to evaluate its performance. Our results show that DTaint discovers more vulnerabilities in less time, compared with the existing techniques. Furthermore, we illustrate the effectiveness of DTaint through applying it over six firmware images from four manufacturers. We have found 21 vulnerabilities, where 13 of them are previously-unknown and zero-day vulnerabilities.**

I. Introduction

Over the last decade, Internet of Things (IoT) devices are under rapid development. More and more devices are connected into the cyberspace for easy remote access and efficient management, e.g., residential routers, IP-cameras or industrial control systems. These devices are indispensable to a wide range of industries, including healthcare, manufacturing, and household. Reports show that there are more than 5 million IoT devices connected to the network and the amount will reach nearly 20 billion by 2020 [1]. On the other side of the reach features, those devices are exposed to attackers and malicious users. Due to insufficient security protection, attackers may compromise IoT devices and cause serious security incidents. For instance, millions of infected IoT camera devices attacked the DNS infrastructure in 2016, disrupting Internet services across Europe and the USA.

The functionality of IoT devices is provided by their firmware, which is a type of software semi-permanently embedded in IoT devices. Like software on other platforms, there are many vulnerabilities about the firmware, such as buffer flow, improper access control, information disclosure, and credentials mismanagement. However, it is challenging to apply traditional static and dynamic techniques to analyze firmware. Many device manufacturers do not provide the source code of the firmware that is necessary for static analysis. Furthermore, Manufacturers usually customize the firmware with proprietary hardware components. As a result, it is difficult to use dynamic analysis to find vulnerabilities in firmware. In a study we carried out on more than 6,500 firmware images, most firmware (90%) do not have source code released, nor can be dynamically analyzed.

The taint-style vulnerability [2]–[5] is a common vulnerability in software. Due to weakness of data sanitization, attacker's malicious data can be used by programs in an undesired way. This vulnerability has an input source, a data propagation path, and a sensitive sink. As an example, "Heartbleed" [6] in the cryptographic library OpenSSL and "Devil's Ivy" [7] in gSOAP are two typical taint-style vulnerabilities. The data dependency path plays a vital role in the taint-style vulnerability discovery.

Prior work provides a wide range of analysis techniques for taint-style vulnerability. In dynamic analysis, when firmware cannot be emulated, fuzzing [8]–[10], dynamic taint tracking [11], [12] and concolic symbolic execution [13]–[15] cannot be used to detect the taint-style vulnerability. In the static analysis, Yamaguichi et al. [3], [16] proposed to design code property graph representation and path traversal patterns for various vulnerability types at source codes level. However, the code property graph cannot be directly applied to the firmware because of the binary file. The state-of-the-art tools (e.g. BAP [17], BinNavi, Bitblaze [18] and Angr [19]) provide static binary analysis for detecting vulnerability. When the firmware is the complex system, the static analysis will face several challenges in practice. First, it is hard to find accurate data flow across multiple functions, due to the indirect memory accesses. Second, pointer aliasing would block discovering the data paths for the taint-style vulnerability. Moreover, indirect calls are commonly seen in the firmware. Those obstacles bring about the poor performance of the vulnerability detection in the binary firmware.

In this paper, we propose a novel solution, called DTaint,

†Qiang Li is the corresponding author.

2158-3927/18/$31.00 Â©2018 IEEE
DOI 10.1109/DSN.2018.00052

based on static binary analysis to address the above challenges and detect taint-style vulnerability in embedded device firmware. The key technique is the analysis to comprehensively and accurately identify data flows in firmware binaries.

Specifically, we first transfer the binary executable file into an intermediate representation. For each function in the intermediate representation, DTaint identifies pointer aliasing for finding more accurate data dependency. To identify indirect calls, we calculate the similarity among data structures based on their layout among multiple functions. Furthermore, DTaint generates the intraprocedural and interprocedural data flow in a bottom-up manner, where callees are visited before callers. Based on the data flow graph, we track the sinks and perform backward depth-first traversal to generate paths from sinks to sources. Finally, we check the constraints of tainted data through sink-source paths for identifying whether the data has sanitization.

We have implemented a prototype of DTaint and conducted experiments to validate the DTaint performance. Given a set of firmware images containing more than 6,500 samples, our results show that DTaint can detect more vulnerabilities with less time, compared with the existing techniques. Furthermore, we use DTaint over six firmware images (across four device vendors) to illustrate the effect of DTaint. We have found 21 vulnerabilities, where 13 of them are previously-unknown and zero-day vulnerabilities.

In summary, our contributions are the following.

- To the best of our knowledge, our work is the first work to detect the taint-style vulnerability in embedded device firmware binaries, when accessing source code or emulating the firmware is not possible.
- We have implemented a prototype system of DTaint. The experimental results show that DTaint can discover more vulnerabilities with less time, compared with conventional tools.
- We apply DTaint to six firmware images for detecting the taint-style vulnerability. We have found 21 vulnerabilities, where 13 of them are previously-unknown/zero-day vulnerabilities.

The remainder of the paper is structured as follows: Section II illustrates the background of embedded device firmware. Section III presents the data flow identification of DTaint, and Section IV details the implementation. Section V presents real-world experiments. Section VI discusses related work, and finally, Section VII concludes this paper.

II. Firmware Analysis

In this section, we first present the background of embedded device firmware for detecting vulnerabilities. Then, we discuss the problem of taint-style vulnerability discovery when we can not emulate the firmware and have no source code.

A. Embedded Device Firmware

Firmware is the "software for hardware" running on the embedded devices, which provides the functionality of IoT devices. Usually, firmware shares similar architectures (e.g., ARM or MIPS CPUs) and provides multiple network modules (e.g., HTTP, RTSP, and UPnP) for embedded devices. In this work, we focus on firmware based on Linux, which is the most commonly used in today's embedded devices, including residential routers, camera, and printers. As our goal is to detect vulnerabilities in IoT firmware, we first carry out an empirical study to understand the limitation of traditional dynamic analysis and static analysis techniques when dealing with IoT firmware.

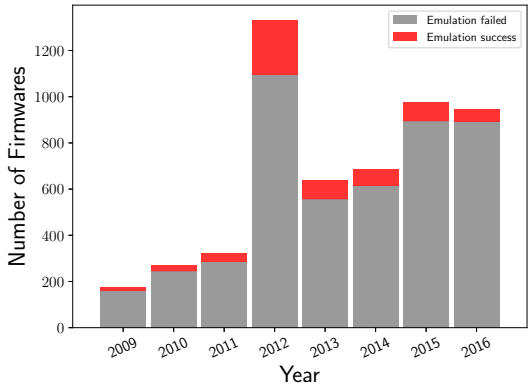

Fig. 1: The number of firmware can be successfully emulated.

Firmware collection. Prior work [20] proposed to collect firmware images for generating their online fingerprints. Here, we developed a web crawler to download firmware images from vendors' websites. Device manufacturers often distribute the firmware images for their products on their official websites. One challenges in firmware collection is to deal with specific web page formats of manufacturers' websites, where we need to parse firmware information, such as the vendor, product name, release date, version number, change log, etc. We used specific web crawler scripts for each website to parse the URL of firmware images. After downloading these binary images, we used the metadata on the official websites to label the firmware image with detailed description information. When vendor websites disallow web crawling, we manually downloaded these images using the browser. In total, we have collected 6,529 firmware images from 12 manufacturers.

Effectiveness of dynamic analysis. For dynamic analysis over the firmware, we use the FIRMADYNE [21] full system emulator to run the bootstrap of firmware images. FIRMADYNE is the first automated dynamic analysis system for embedded firmware. Figure 1 shows the summary of dynamic analysis in firmware images released from 2009 to 2016. The red portion on top of the histogram is the number of the firmware that we can successfully run dynamical analysis in the emulator tool. The gray color histogram represents the firmware that the emulator failed to execute its functionality. We can see that only a small faction of the firmware can be simulated for discovering the vulnerability, less than 670. Most of the firmware images (5,859) cannot be directly used in dynamic analysis, mainly because the firmware failed to

```
1  /* ssl/t1_lib.c */
2  // [...]
3  int tls1_process_heartbeat(SSL *s) {
4    unsigned char *p = &s->s3->rrec.data[0], *p1;
5    unsigned short hbtype;
6    unsigned int payload;
7    unsigned int padding = 16; /* Use minimum padding */
8    /* Read type and payload length first */
9    hbtype = *p++;
10   n2s(p, payload);
11   p1 = p;
12   // [...]
13   if (hbtype == TLS1_HB_REQUEST) {
14     unsigned char *buffer, *bp;
15     int r;
16     buffer = OPENSSL_malloc(1 + 2 + payload + padding);
17     bp = buffer;
18     /* Enter response type, length and copy payload */
19     *bp++ = TLS1_HB_RESPONSE;
20     s2n(payload, bp);
21     memcpy(bp, p1, payload);
22     bp += payload;
23     /* Random padding */
24     RAND_pseudo_bytes(bp, padding);
25     r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
26                          3 + payload + padding);
27     // [...]
28   }
29   // [...]
30   return 0;
31 }
```

Fig. 2: The *Heartbleed* vulnerability in OpenSSL.

```
< ssl3_read_bytes() >
0x7C3C: MOV   R11,  R0
...
0x8530: LDR   R4,   [R11, #0x58]
...
0x7F14: MOV   R0,   R11
0x7F18: BL    ssl3_read_n
...
0x8130: LDR   R3,   [R11, #0x4C]
0x8134: ADD   R3,   R3, #5
...
0x8148: STR   R3,   [R4, #0x118]
...
0x83f8: MOV   R0,   R11
0x83FC: BL    tls1_process_heartbeat

< tls1_process_heartbeat() >
0xFBA0: LDR   R3,   [R0, #0x58]
0xFBA4: LDR   R2,   [R3, #0x118]
0xFBAC: LDRB  R2,   [R2, #1]
0xFBB0: LDRB  R3,   [R2, #2]
0xFBB4: ORR   R3,   R3, R2, LSL#8 ;n2s
...
0xFC38: MOV   R2,   R3      ; n
0xFC3C: ADD   R1,   R2, #3  ; src
0xFC40: MOV   R0,   R9      ; dest
0xFC44: BL    memcpy        ; taint

< ssl3_read_n() >
0xFFE0: MOV   R5,   R0
0xFFE4: LDR   R8,   [R0, #0x58]
...
0x700C: AND   R10,  R3, #7
...
0x7090: LDR   R9,   [R8, #0xEC]
0x7094: ADD   R9,   R9, R10
0x7098: LDR   R1,   [R5, #0x4C]
0x709C: CMP   R9,   R1
0x70A0: BEQ   loc_0x70BC
0x70A4: ADD   R2,   R4, R11
0x70A8: MOV   R0,   R9
0x70AC: BL    memmove
0x70B0: STR   R9,   [R5, #0x4C]
...
0x7234: ADD   R1,   R4, R11
0x7238: SUB   R2,   R7, R4      ; read_n
0x723C: ADD   R1,   R9, R1      ; read_buf
0x7240: BL    BIO_read          ; source
```

Fig. 3: Associated assembly code snippet for *Heartbleed*.

access custom and proprietary hardware components or failed to initialize the network configuration in the boot process.

Effectiveness of static analysis. For static analysis over the firmware, we further identify whether manufacturers provide the source code of the firmware image. Many manufacturers often customize firmware based on an open source code for their products and systems. For example, OpenWrt [22] is a Linux distribution for embedded devices and open source for manufacturers to build their own firmware. Even the firmware based on OpenWrt, the customized parts are close-source to protect business secrets. For 5,023 firmware images, we cannot find their code or documents corresponding to their firmware.

In summary, the firmware images often lack source code for the static analysis, and 90% firmware images cannot be emulated successfully by the state-of-the-art tools for the dynamic analysis. Therefore, in our solution, we based our approach on static binary analysis.

B. Taint-style Vulnerability

The taint-style vulnerability [3] is one common weakness in software. This vulnerability is caused by the lack of security sanitization on data propagation path. As a result, inputs provided by attackers can pass the tainted data to an unsafe sensitive point, causing system security breaches. For example, buffer overflow and command injection belong to taint-style vulnerabilities. There are three general steps for discovering taint-style vulnerabilities, as following:

1) *Identifying attacker-controlled sources*. We can mark the data input as the tainted point, indicating where the data comes from. Several places are insecure and easily controlled by attackers, such as a file, and the network (functions recv() and recvfrom()).
2) *Identifying security-sensitive sinks*. We mark the unsafe library functions (e.g., strcpy(), memcpy(), and system()) or code patterns (loop buffer copies) as sinks, denoting where the data maybe trigger a vulnerability.
3) *Finding vulnerability as unsafe data paths*. We can obtain a data propagation flow between an input point and a sink. If the path lacks security sanitization, we can mark it as the unsafe path.

We use a typical example to illustrate the taint-style vulnerability. The *Heartbleed* bug in OpenSSL discovered in 2014 [6] is a taint-style vulnerability. Figure 2 shows the vulnerable source code: The tainted variable *payload* is defined by the macro *n2s*, which reads a 16-bit integer from network data (line 10). In line 21, this tainted integer is passed to the third argument of function *memcpy* without security sanitization. In particular, there is no guarantee that the *payload* is less than or equal to the size of the source buffer *p1*. Then, uninitialized heap memory may be copied to the buffer *bp*, and sent out to the network via a call to *ssl3_write_bytes* on line 25. At the source-code level, a code property graph from the sink *payload* to the source *n2s* can be built to generate a path for detecting this vulnerability.

However, at the binary level, there are many obstacles hindering the discovery of data paths, such as the loss of structures and compiler optimizations (e.g., inline or macro expansion). We illustrate the challenge of detecting the taint-style vulnerability using the assembly code of the *Heartbleed* vulnerability. Figure 3 shows the associated assembly code that caused this weakness. Specifically, we cannot find source *n2s* in the assembly code, because the compiler optimization directly insert the body of the macro *n2s* inside the callsite in function *tls1_process_heartbeat* (at 0x6FBB4). By manual analysis, the third argument of a call to *memcpy* at 0x6FC44 in *tls1_process_heartbeat* comes from the second argument of a call to *BIO_read* at 0x67240 in *ssl3_read_n*. In binary code level, trace the data propagation in the memory is the challenging for detecting taint-style vulnerabilities. As far as we know, the state-of-the-art static taint analysis cannot detect *Heartbleed* weakness at the binary code level.

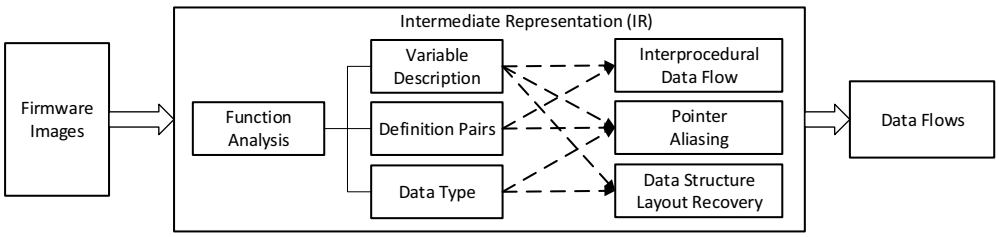

Fig. 4: The architectural diagram of DTaint, generating data flows in a firmware image.

III. DTaint: Data Flow Identification

In this section, we present the data flow identification technique, which is the core of DTaint.

A. Overview

With the increasing complexity of the firmware, comprehensive and accurate data flows are in dire need to be extracted to detect the taint-style vulnerability. There are three challenges for generating such data flows, which are context-sensitive and interprocedural. First, it is hard to generate data flow through the memory when the pointer aliases exist. Second, there are many indirect calls in the firmware, leading to the loss of the data propagation. Third, traversing the function call graphs from top to bottom would bring numerous repetitive analysis. For example, the same callee may be analyzed multiple times. Particularly, the large-size firmware contains massive context-sensitive information passing between callee and callsite.

To address these three challenges, we propose DTaint as shown in Figure 4. It takes firmware images as inputs and extracts data flows for taint-style vulnerability detection. There are four modules to generate data flows, including function analysis, pointer aliasing, data structure and interprocedural data flow components. In the function analysis component, DTaint separately analyzes every function by static symbolic analysis. Specifically, this component proposes three techniques: variable description through the memory, data type inference, and definition pairs of every function. In pointer aliasing component, we identify the pointer alias by combining functional definition pairs and the variable description. In data structure component, DTaint establishes the data flow between indirect call functions through similarity of the data structure. We propose to use similarity of the layout to the data structure to determine whether multiple functions share the same data structure. In interprocedural data flow component, we use the "bottom-up" approach to traverse the call graph to build intraprocedural and interprocedural data flows. DTaint updates the definitions of callees to all callers and only analyzes every function just one time.

Next, we explicate every module of DTaint in details.

B. Function Analysis

We first prepare the binary for static analysis by transforming the binary code into an intermediate representation (IR). Our analysis is based on the VEX IR [23]. DTaint first creates a control flow graph (CFG) for the firmware. DTaint performs a static analysis on the firmware to generate the CFG for each function separately. The node in a CFG represents a basic block, and the directed edge represents the transition in the control flow. Every basic block generates a symbolic state in the current path. DTaint explores both directions of each conditional branch. When loops in the function, DTaint uses a simple heuristic rule: *blocks in the same loop are only analyzed once*. Therefore, a basic block may contain several distinct symbolic states in the different path.

When DTaint analyzes every function, we need to handle the calling convention of functions carefully. DTaint uses unique symbolic values to initialize the corresponding calling convention. For example, in the 32-bit ARM architecture, the first four parameters are passed through the registers $R0$-$R3$. When the number of parameters exceeds four, the excess parameters are passed through the stack. The return value of the function is saved in the register $R0$. DTaint initializes the registers $R0$-$R3$ with symbolic variables $arg0$-$arg3$ and initializes other arguments by pushing the symbolic variables $arg4$-$arg9$ to the stack. For the function callee, we save a symbolic value $ret_{callsite}$ to the register $R0$ and return to the callsite by hooking the callee. Then, we can judge the number and type of parameters used by the function, and the return value from callsite according to the unique symbolic value.

Variable Description. In the IR, it is difficult to trace the data propagation across multiple functions in the memory. DTaint uses the address expression of the memory to describe the variable. The rationale is that when memory is stored or loaded, the variable is allocated by either the memory or the register. In the absolute memory address, DTaint directly uses the memory to present variables, such as 0x670B0. In the indirect memory address, DTaint uses the "base + offset" form to represent variables, where the base is the register, and the offset is the register or integer constant. DTaint uses *deref* as a reference to memory access. We utilize a specific example to illustrate the variable description. For *R1* variable in the IR, DTaint uses *R1 = deref(R5 + 0x4C)* to replace the instruction *LDR R1, [R5, 0x4C]* as shown in Figure 5. Then, DTaint could generates the expressions of memory-access (Figure 6) variables for tracing data propagation between function *foo* and *woo*.

Data Type. In the IR, DTaint uses the data layout to represent the data structure for discovering data flow among multiple call graphs. The layout of data structure depends on the primitive types. There are several primitive types,

```
< foo(R0, R1) >
C3C: SUB SP, SP, 0x118
C40: MOV R5, R0
C44: MOV R4, R1
C48: BL  woo
C4C: MOV R2, R0
C50: LDR R1, [R5, 0x4C]
C54: ADD R0, SP, 0x118-0x100
C58: BL  memcpy          ;sink

< woo(R0, R1) >
C60: LDR R0, [R1, 0x24]
C64: STR R0, [R0, 0x4C]  ;alias
..     ...
C70: MOV R2, 0x200
C74: MOV R1, R0
C78: BL  recv            ;source
```

Fig. 5: Assembly code snippet of function *foo* and *woo*.

```
< foo(arg0, arg1) >
C3C: SP = SP-0x118
C40: R5 = arg0
C44: R4 = arg1
C48: call woo,
  R0 = ret_{c3c-cf0}
C4C: R2 = ret_{c3c-cf0}
C50: R1 = deref(arg0+0x4C)
C54: R0 = SP-0x100
C58: call memcpy,
  n = ret_{c3c-cf0}
  deref(sp-0x100)=deref(deref(arg0+0x4C))

< woo(arg0, arg1) >
C60: R0 = deref(arg1+0x24)
C64: deref(arg0+0x4C)=deref(arg1+0x24)
..     ...
C70: R2 = 0x200
C74: R1 = deref(arg1+0x24)
C78: call recv,
  deref(deref(arg1+0x24))=taint
  R0 = ret_recv
  deref(deref(arg0+0x4C))=taint   ;new def
```

Fig. 6: Static symbolic analysis for function *foo* and *woo*.

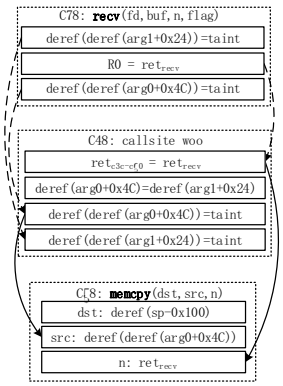

Fig. 7: Data flow between the source *recv* and sink *memcpy*.

such as *int*, *char*, *int** and *char**. DTaint infers data type through two ways: (1) standard C/C++ library function calls, and (2) a machine instruction defining the data type. In the most standard library calls, the parameters are specified data types. For example, the two arguments of the *strcpy* function are all *char** types. DTaint marks the types of registers and memory variables from the arguments and returns of the library functions. In machine instructions, there are specific types of operands. In 32-bit ARM architecture, DTaint infers data type from the instruction as follows: given *LDR/STR* instructions, they have an indirect memory access operand (e.g., *LDR R0, [R4, 4]*), the value held in the register *R4* must be a pointer. *CMP* instruction (e.g, *CMP R0, 8*), the value held in the register *R0* must be an integer.

Definition Pairs. In every function, DTaint would generate the definition pairs *(d, u)*, where the variable in definition pair is the concrete value or symbolic expression in memory and register. DTaint uses those definition pairs of functions to find pointer alias, and to generate intra- and inter-procedural data flow.

C. Pointer Aliasing

Pointer aliasing refers to the situation when the same memory location is "pointed to" by different names. When pointer aliasing exists, many data flows cannot be found in the firmware. In the design of DTaint, we focus on firmware that is written in the C/C++ languages, which is the most common case. In binary code, we mainly consider two kinds of pointer aliases: the pointer assignment statement by "move" instruction and the pointer being written to memory by "store" instruction. In the first pointer alias, DTaint addresses it through static analysis. As an example, *int *p = x; q = p*, **p* and **q* are alias, and they would have same name by symbolic analysis.

In the second case, the pointer is saved into the memory. For example, *int *p = x; *(q+4) = p*, **(*(q+4))* and **p* are alias. If *q* is a symbolic value, the generic symbolic analysis cannot directly identify **p* and **(*(q+4))* as aliases. The second pointer aliasing can be described as the following formula:

$$deref(base1 + offset1) = base2 + offset2 \qquad (1)$$

where $base1$ and $base2$ are both addresses, $offset1$ and $offset2$ are integers.

Algorithm 1 shows how DTaint recognizes the pointer aliasing in Formula (1). Lines 4-7 are used to find all pointer aliases in each function. Every function has a definition pairs (DP), a set $ALIAS$ for storing pointer alias, and a set DOP for storing definitions of variable points-to. If the operation of d is equal to $deref$ and u is a pointer type, we use *GetBasePtr* function to obtain the base address and offset of u. For a pointer '*base + offset*', where base and offset are both symbolic, we need to identify which symbolic value is a pointer. We merge the $(d, base, offset)$ with the ALIAS set. In the firmware, it is commonly seen that a variable expression has multi-base pointers. For example, $deref(deref(arg0 + 0x58) + 0xEC)$ has two base pointers, including $arg0$ and $deref(arg0 + 0x58)$. We store them in the set DOP. We obtain all the base pointers contained in variable *d* through the function $GetPtrInVar$ in Lines 8-11. We replace the base pointer of the variable *d* with an alias and get a new variable *new_d*, where variable *new_d* and *d* point to same object. Then we put the new definition pair *(new_d, u)* to the definition pairs (DP) in Line 13-22.

D. The Similarity of Data Structure

Typically, in a C or C++ program, the indirect call takes the address of the callee from the memory or the register. There are multiple call graphs when the indirect call occurs, and we cannot find the data flows between them. DTaint identifies those indirect call relations through calculating the similarity of data structure. The key insight is that the object pointing to the data structure may be used in multiple functions and they may share the same data structure.

Data Structure Layout. DTaint uses data structure layout as the similarity metric of the data structure in the IR. At the binary level, some instructions reveal the actual offsets

Algorithm 1 Pointer Aliasing Recognition.

Input: Definition Pairs: DP

```
1: function AliasReplace(DP)
2:     ALIAS ⇐ φ; DOP ⇐ φ
3:     for each d, u ∈ DP do
4:         if (d.op == deref) ∧ u is a pointer then
5:             base, offset = GetBasePtr(u)
6:             ALIAS = ALIAS ∪ {(d, base, offset)}
7:         end if
8:         if d.op == deref then
9:             ptrs = GetPtrInVar(d)
10:            DOP = DOP ∪ {(d, u, ptrs)}
11:        end if
12:    end for
13:    for each d, u, ptrs ∈ DOP do
14:        for each ptr ∈ ptrs do
15:            for each alias, p, o ∈ ALIAS do
16:                if p == ptr then
17:                    new_d = d.Replace(p, alias − o)
18:                    DP = DP ∪ {(new_d, u)}
19:                end if
20:            end for
21:        end for
22:    end for
23: end function
```

for the specified data structure fields. These instructions can be described as the form '*base + offset*'. DTaint infers the data types and collects the '*base + offset*' expressions to construct the layout of data structures. The characteristic of taint-style vulnerability determines that data structures usually contain pointers. DTaint infers data structure layout consisting of pointers. If a stack pointer is passed as the argument to the callee, we will use it as the base pointer and construct the layout of the data structure on the stack.

Based on primitive types, DTaint represents the data structure through a 3-tuple (b, o, t), where b is base address, o is the constant offset of the field, and t is the field type. A multi-layer structure can be described as $S = (S_1, S_2, ..., S_n)$, where S represents a collection of fields with the same root pointer, S_i represents a set of fields with the same base address, and n is the number of different base address in S. Given the structure A and the structure B, DTaint determines whether they are similar through two rules:

- $base(A) \subseteq base(B)$ or $base(B) \subseteq base(A)$, where $base(A)$ is the structure A's base address set.
- fields with same offset at same base address in A and B must have the same type.

If data structures A and B satisfy both two conditions, DTaint calculates their similarity as following:

$$\sigma(A, B) = \sum_{(i,j)} \frac{|A_i \bigcap B_j|}{|A_i \bigcup B_j|}, \quad where\ (i, j) \in pair(A, B) \quad (2)$$

The function $pair(A, B)$ is used to align the base address of two data structure. For each (i, j) in $pair(A, B)$, the A_i and B_j have the same base address. For different indirect call graphs, we compute the data structure similarity between them, and establish data dependencies of two data structures with the highest similarity σ.

Listing 1: Example that illustrates the return value of function.

```
1 void A() {              int* B() {
2     int *x = B();           return malloc(sizeof(int);
3     int *y = B();     }
4     *x = 8;
5 }
```

E. Interprocedural Data Flow.

As we mentioned before, DTaint generates the definition pairs of every function separately. In this case, the modifications made by callees to their pointer arguments and return values, or the definitions from arguments of callsites could hind interprocedural data flow between functions. Conventional techniques usually generate data flows traversing the function call graph from top to bottom. The problem here is that the different context-sensitive information needs to be passed to callee through callsite chains, which causes the same callee to be analyzed multiple times.

DTaint uses the "bottom-up" approach to traverse the call graph to build intraprocedural and interprocedural data flows. In the DTaint, each function is analyzed only once. Specifically, DTaint uses the definition pairs to construct *use-def* and *def-use* chains [24] to generate data flows by traversing the call graph in post-order. During the graph traversal, once the definition *(d,u)* meets the calling convention, DTaint would update the definition information of callee and forward the new definition to the definition pairs of callers. Also, DTaint pushes the undefined variable *use* to all callers.

For interprocedural data flow, there are two types of indirect definitions that affect the propagation of data flow information: (1) the modification of the callee to the pointer parameter and the return value, and (2) the definitions of the actual arguments of the caller.

In the first case, algorithm 2 shows how DTaint updates definition information of every function to identify these modifications of callees. For return values, DTaint returns a unique symbolic value $ret_{callsite}$ to the callsite for every callee during the symbolic analysis. Line 3-6 shows updating the pair (d, u) which contains a return symbolic. DTaint replaces the symbolic value $ret_{callsite}$ with the actual return value of the callee by function $ReplaceRetVarialbe$. When the return value is a heap pointer, for example, Listing 1 shows the *x* and *y* are the heap pointer from function $B()$. DTaint uniquely identifies the heap pointer by calculating the hash value of the callsite chain. The callsite chain is a sequence of callsite from the use of heap pointer to the allocation of the heap.

When the parameter is a pointer, and the callee changes the object that parameter points-to. DTaint updates definitions to the definition pairs of callsites in Line 7-13. If the definition pair *(d, u)* can reach the exit statement and the root pointer of *d* is formal argument *arg0-arg9*, DTaint would replace the

Algorithm 2 Updating Definition Pairs Information

```
1: function UpdatDefPairs((d, u), CS)
2:     if (d, u) contains ret symbol then
3:         d = ReplaceRetVarialbe(d)
4:         u = ReplaceRetVarialbe(u)
5:     end if
6:     if (d, u) could reach the exit statement then
7:         if d.rootPtr is argument or return pointer then
8:             new_d = ReplaceFormalArgs(d, CS)
9:             new_u = ReplaceFormalArgs(u, CS)
10:        end if
11:        PushToCallSite(CS, (new_d, new_u))
12:    end if
13: end function
14:
15: function ForwardUndefinedUse(u, DP, CS)
16:    if u cannot find a reaching definition d in DU then
17:        new_u = ReplaceFormalArgs(u, CS)
18:        PushToCallSite(CS, (none, new_u))
19:    end if
20: end function
21:
22: function ReplaceRetVarialbe(v)
23:    return v.replace(ret_callsite, getRetValue(callsite))
24: end function
25:
26: function ReplaceFormalArgs(v, CS)
27:    return v.replace(formalArgs(v), actualArgs(CS))
28: end function
```

formal argument *arg_i* in the definition pair *(d, u)* with the *i th* actual argument of callsite by function $ReplaceFormalArgs$. Then, it pushes the new definition pair *(new_d, new_u)* to callsites. When the return value is a pointer, and the callee changes the object that the return variable points-to. In this case, DTaint also updates the functional definition pairs in Line 7-13.

In the second case, DTaint identifies whether every variable in use is defined in the definition pairs of local functions. The algorithm 2 (Line 16-22) shows that DTaint updates the undefined variable u to callsites. If variable u is defined in the local function, we can directly generate a data dependency. Otherwise, DTaint replaces the formal argument *arg_i* in the variable *u* with the *i th* actual argument of callsite by function $ReplaceFormalArgs$. Then, we send the new use variable to all callsites. For instance, Figure 7 shows an interprocedural data flow generation for the function *foo* and *woo*, where the dashed lines indicate the definition updating, and the solid lines indicate the data flow.

IV. Implementation

This section provides an overview of the DTaint implementation for discovering the taint-style vulnerability in the firmware. First, we need to extract the binary file from the firmware. We use a custom-written extraction utility built around the Binwalk API [25] to extract the root file system. Then we choose the binary file of interest and load it into static symbolic analysis module for generating IR for DTaint. In the IR, DTaint generates data flow, details in Section III. DTaint selects several sinks and sources for constructing the tuple *(source, path, sink)*. We use two kinds of constraint expressions to check whether the tuple *(source, path, sink)* lacks security sanitization. If so, we have discovered a taint-style vulnerability.

TABLE I: Sources and Sinks

	Library functions
Sensitive sinks	strcpy, strncpy, sprintf, memcpy strcat, sscanf, system, popen, loop
Input sources	read, recv, recvfrom, recvmsg getenv, fgets, websGetVar, find_var

In the static symbolic analysis module, DTaint uses the symbol execution engine module *SIMUVEX* from Angr [19], which is popular and a combination of techniques in Mayhem [26] and S2E [27]. DTaint converts the binary into an intermediate language VEX, the IR of the Valgrind project [23]. VEX intermediate representation (VEX IR) is an architecture-neutral intermediate representation, abstracting binary code into a representation designed to make program analysis easier. DTaint performs static symbolic analysis over every function for generating the concrete and symbolic variables, constraint expressions and the data type of every variable. DTaint generates paths in the tuple *(source, path, sink)* for identifying the taint-style vulnerability. We have extracted common library functions, as shown in Table I. DTaint identifies loop copy as the sink by identifying the copy statements in the loop. The source contains several injecting input nodes, e.g., *recv()*, *recvfrom()* and *fgets()*.

For every data path, we have implemented two kinds of constraint expressions to check whether tainted data is sanitized from source to sink, as follows:

- Buffer overflow vulnerability. The weakness is the insufficient validation of length fields passed to copy operations. We have generated the constraint expressions for every tainted variable, such as the third argument of *memcpy*, the length of copy size of *strcpy*. For each path, if there is no constraint (i.e., $n < 64$ or $n < y$, y is a symbolic value) on the tainted variable *n* in all constraint expressions, the path is identified as the vulnerability.
- Command injection vulnerability. It is caused by the library functions (i.e., *system*, or *popen*) in embedded Linux systems. We have used the constraint expression for the first argument *cmd* of function *system*. For each path, if there is no constraint (i.e., *deref(cmd + offset) != ';'*) on the tainted variable *cmd* in all constraint expressions, this path is a vulnerability.

In summary, we have implemented a prototype of DTaint including three modules: a static symbolic analysis, a data flow generation module and a vulnerability detection module. The

TABLE II: The summary information of firmware analysis using DTaint.

Index	Manufacturer	Firmware Version	Architecture	Binary	Size (KB)	Functions	Blocks	Call graph edges
1	D-Link	DIR-645_1.03	MIPS	cgibin	156	237	3,414	1,087
2	D-Link	DIR-890L_1.03	ARM	cgibin	151	358	3,913	1,418
3	Netgear	DGN1000-V1.1.00.46	MIPS	setup.cgi	331	732	4,943	2,457
4	Netgear	DGN2200-V1.0.0.50	MIPS	httpd	994	796	11,183	4,497
5	Uniview	IPC_6201	ARM	mwareserver	4,813	6,714	99,958	32,495
6	Hikvision	DS-2CD6233F	ARM	centaurus	13,199	14,035	219,945	68,974

TABLE III: The summary of the taint-style vulnerabilities that DTaint found.

Firmware	Analysis functions	Sinks count	Execution time (minutes)	Vulnerable paths	Vulnerability
D-Link DIR-645_1.03	237	176	1.18	7	4
D-Link DIR-890L_1.03	358	276	1.48	5	2
Netgear DGN1000-V1.1.00.46	732	958	3.19	19	6
Netgear DGN2200-V1.0.0.50	796	1264	6.62	14	2
Uniview IPC_6201	430	447	3.97	10	1
Hikvision DS-2CD6233F	3,233	2,052	31.89	30	6

static symbolic analysis contains 900 lines of Python code, extending from the open source symbolic execution engine *SIMUVEX*. The data flow generation module has nearly 2,200 lines of Python code, and the vulnerability detection module takes approximately 650 lines of Python code.

V. Evaluation

In this section, we first use six firmware images to illustrate the effectiveness of DTaint. We have analyzed vulnerabilities detected by DTaint, including eight previously reported weaknesses and 13 unknown/zero-day vulnerabilities. Then, we evaluate the performance of DTaint. We conducted the experiments on an Ubuntu 16.04 LTS system equipped with a 64-bit 4-core Intel CPU and 128 GB RAM. The firmware image is the input of DTaint, and the paths between sources and sinks are output for the vulnerability discovery.

A. Vulnerability Detection Effectiveness

To illustrating DTaint, we use six firmware images for discovering their taint-style vulnerabilities. Table II shows the summary information of six firmware images. They come from four different device manufacturers, including Hikvision, Uniview, Netgear, and D-Link. Those manufacturers have been prevalent in the market with a large number of global deployments. It is important to note that those images are proprietary and we cannot simulate all. Table III shows the summary information of the taint-style vulnerability when we use DTaint to analyze six firmware images. Totally, we have found 21 taint-style vulnerabilities over firmware images. For more details, Table IV shows the previously known vulnerabilities, which from the CVE database [28] and exploit database [29]. Table V shows zero-day vulnerabilities detected by DTaint. We use real devices for verifying these vulnerabilities in the firmware. Below we would explicate and analyze these taint-style vulnerabilities in the firmware.

1) D-Link: D-Link is a well-known manufacturer of embedded equipment, mainly providing routers, network cameras, switches and other products around the world. Its products are primarily divided into domestic and commercial use, such equipment in the confidentiality and security plays a vital role. D-Link firmware is the lightest system with the smallest size. We use DTaint to analyze two version: DIR-645_1.03 and DIR-890L_1.03. Those two firmware versions utilize the same binary file *cgibin*, less than 160KB.

Analysis. For the binary *cgibin*, DTaint takes nearly 1.18 minutes to analyze 237 functions, which have 3,414 blocks and 1,087 call edges. Among all 452 sinks, DTaint found 12 vulnerable paths. For the version DIR-645 1.03, DTaint discovers four vulnerabilities with the taint-style, as shown Table III. For the version DIR-890L 1.03, there are two previous reported vulnerabilities found by DTaint.

Discovery. DTaint found six taint-style vulnerabilities, where five of them are the previous known, and one weakness is unknown. We now describe them in detail.

- CVE-2013-7389. This CVE item has two taint-style vulnerabilities. The first one is a stack-based buffer overflow. The HTTP POST parameter named "password" from the source *read* is copied into a local stack buffer using the function *strncpy* without checking its length. The second one is a stack-based buffer overflow. An overly-long cookie value from the function *getenv* is copied into a local stack buffer using the function *sprintf* without check its length.
- CVE-2015-2051. It is a code injection vulnerability. The variable field "SOAPAction" is read from the function *getenv* and passed to the sink *system* without checking whether it contains special characters (e.g., semicolon).

- CVE-2016-5681. It is a stack-based buffer overflow. A long session cookie value from the function *getenv* is copied into a local stack buffer of max size 152 using the function *strcpy* without checking its length.
- Unknown. This unknown vulnerability is a code injection vulnerability. The tainted data is not strictly checked when it is propagated from a function *getenv* to the sink *system*. We have reported it to the CVE website [28] and D-Link vendor. So far, we do not have its CVE label.

As we mentioned, *cgibin* is the simple system. Most of the vulnerabilities have been previously reported in the CVE. Only one is unknown.

2) Netgear: Netgear is a leading corporate network solutions company in the world, primarily producing products such as routers, switches, and networked storage. Netgear firmware is more complicated than D-Link firmware. We use DTaint to analyze two version: DGN1000-V1.1.00.46 and DGN2200-V1.0.0.50. The first one uses the binary file *setup.cgi* with 331KB, and the latter uses the binary file *httpd* with 994KB.

Analysis. In the version DGN1000-V1.1.00.46, DTaint analyzes 732 functions, 4,943 blocks and generates 2,457 call graph edges. DTaint detected 19 unsafe paths within 3.19 minutes. In the version DGN2200-V1.0.0.50, DTaint detected 14 unsafe paths within 6.62 minutes. Compared with D-Link firmware, Netgear contains more functions and features, arising the higher time cost of analysis. DTaint found two vulnerabilities in the version DGN2200-V1.0.0.50, and six vulnerabilities in the version DGN1000-V1.1.00.46.

Discovery. Among of eight vulnerabilities detected by DTaint, where five vulnerabilities are unknown, and three vulnerabilities are previously reported. We now briefly describe them in detail.

- CVE-2017-6334. It is a code injection vulnerability. The variable field *host_name* is read from the attacker-controlled source *websGetVar* and passed to the sink *system* without checking whether it contains a semicolon, causing a code injection.
- CVE-2017-6077. It is a code injection vulnerability. The source *websGetVar* reads the variable *ping_IPAddr* and passes it to the sink *system* without checking whether it contains a semicolon.
- EDB-ID:43055. It is a code injection vulnerability. The source *find_val* reads the variable *cmd* and passes it to the sink *popen* without checking whether it contains a semicolon.
- Unknown. Four vulnerabilities are belonging to the code injection vulnerability.
- Unknown. One vulnerability is belonging to the stack-based buffer overflow.

The binary files (*setup.cgi* and *httpd*) have larger size than the file *cgibin* in D-Link. DTaint has found more unknown vulnerabilities.

3) Uniview: Uniview is a professional video surveillance products and solutions provider. Uniview firmware has the larger size (4,813KB) than D-Link and Netgear. The firmware uses the binary file *mwareserver*, which is deployed as a CCTV camera with networking functionality. We used DTaint to analyze the latest firmware version of Uniview IPC_6201.

TABLE IV: The previous reported vulnerabilities with the taint style using DTaint.

Vulnerability	Sink	Sources	Security check
CVE-2013-7389	strncpy	read	N
	sprintf	getenv	N
CVE-2015-2051	system	getenv	N
CVE-2016-5681	strcpy	getenv	N
EDB-ID:43055	popen	find_val	N
CVE-2017-6334	system	websGetVar	N
CVE-2017-6077	system	websGetVar	N

TABLE V: Zero-day vulnerabilities discovered using our tool. The last column is the number of different vulnerabilities in the firmware.

Firmware	types	bug status	bugs
Hikvision DS-2CD6233F	Buffer Overflow	repaired	6
Uniview IPC 6201	Buffer Overflow	reviewing	1
DIR-645	Command Injection	repaired	1
Netgear DGN1000	Command Injection	-	3
	Command Injection	reviewing	1
	Buffer Overflow	-	1

Analysis. In the version IPC_6201, *mwareserver* has 6,714 functions (99,958 blocks) and 32,495 call graph edges. It is important noted that the functional logic would increase enormous complexity as long as the program size, and Angr tool cannot analyze it in such large size. We manually extract 430 functions that are used to process RTSP and HTTP module. DTaint found 10 unsafe paths within 3.97 minutes. We found 1 zero-day vulnerability in the Uniview firmware.

Discover. For this vulnerability, we have reported it to Uniview vendor.

- Unknown. It is a buffer overflow vulnerability. When input use from session field of RTSP methods reaches the program, there is a data path propagating it to *sscanf()*. The function *sscanf()* can read 254 characters and copy the string to the local stack buffer which has the max size of 180 bytes.

4) Hikvision: Hikvision is a manufacturer for providing surveillance devices for traffic, police, office, and household. There are a large number of surveillance devices are visible and accessible through IP addresses. We use the camera for protecting physical world security. However, the vulnerability of devices would lead to a serious concern. For instance, people can directly know those surrounding scenes by watching the video streams while you are working or falling asleep at home. Hikvision firmware is the most complex system (13,199KB). We use DTaint to analyze the version DS-2CD6233F. Specifically, we focus on the binary file *centaurus*

TABLE VI: CPU, memory and time usage of prototype software.

	CPU usage	Memory
Static symbolic analysis	25%	15.3GB
Data flow generation	10%	208.9MB

in Hikvision firmware.

Analysis The binary has 14,035 functions, 219,945 blocks and 68,974 call graph edges. These functions are used to handle the initialization, file system updates, and a large number of network protocol data processing. Because of the complicated system, we still focus on four modules including RTSP, HTTP, ONVIF, and ISAPI in the *centaurus*. DTaint analyzes 3,233 functions within 31.89 minutes, and 30 sinks may be vulnerable. By validating, we found that six of them were vulnerabilities.

Discovery We have reported these vulnerabilities to Hikvision vendor. Below we simply explicate those six zero-day vulnerabilities.

1) Unknown. It is a stack-based buffer overflow. The attacker-controlled string reads from the source *read* is copied into a local stack buffer of max size 48 bytes using the function *memcpy* without checking its length
2) Unknown. There are two taint-style vulnerabilities. They belong to the stack-based buffer overflow. The source function *read* reads 2048 bytes and copies them to a local stack buffer using loop copy operation. However, the string length is larger than stack buffer size.
3) Unknown. There are three taint-style vulnerabilities. They belong to the stack-based buffer overflow. The program parses the parameters of an HTTP URL without security checking. Then the URL parameters are directly copied to the local stack buffer. These three vulnerabilities are associated with pointer alias and the similarity of data structure.

Hikvision firmware is the most complicated system, compared with other firmware. The behind rationale is that manufacturers will improve performance and add features for distributing new device products to the market. We think that this is a market trend that the firmware will become more and more complex.

B. Performance

We have used four programs for evaluating DTaint: OpenSSL, cgibin, setup.cgi, and httpd. We have conducted experiments to evaluate DTaint on its performance. We also used Angr [19] on the same set of applications as a reference.

Resource Overhead. We have measured the overhead of DTaint. Table VI shows the average utilization of memory and CPU resources. The static symbolic analysis module takes the most resources, closing to 15GB memory and 25% CPU usage. It is important to note that the DTaint overhead would grow up along with the increasing complexity of the firmware system. The overhead of DTaint is affordable in practice.

TABLE VII: Time cost between Angr and DTaint.

Program	Time cost (second)			
	Angr		**DTaint**	
	SSA	**DDG**	**SSA**	**DDG**
cgibin	134.49	16463.32	62.34	10.48
setup.cgi	39.17	539.68	33.85	1.205
httpd	106.92	22195.45	60.92	8.87
openssl	102.94	7345.56	47.33	3.09

Time Cost. We have conducted experiments for validating the time cost of DTaint. We use the same functions to calculate the time cost. Table VII shows the time cost of DTaint, with Angr as a reference. In the static symbolic analysis (SSA), DTaint takes less time than Angr. In the data dependency graph (DDG) generation, DTaint spends significantly less time. The reason is that Angr leverages a worklist-based and iterative approach to generate interprocedural data flows [30], which can support data-flow operations such as slicing. As a result, it builds data dependence on every variable (in the register and memory). When the binary complexity is high, it needs to repeatedly build the data flows for the same block and function with different context. In contrast, focusing to support taint analysis, DTaint generates interprocedural data flows from the bottom to top, through updating definitions of callees to callers. Each function would be analyzed once, and the iteration would quickly converge in the function.

VI. Related Work

Our work is to detect the taint-style vulnerability in the proprietary firmware. It is related to both firmware analysis and vulnerability discovery.

Firmware Analysis. With the increasing prevalence of embedded devices, several related works have performed the security analysis of firmware. Prior work analyzed the specific weakness associated with the firmware. Li et al. [20] leveraged the difference of file systems to generate the fingerprint of firmware. Specific, they collected 9.7K firmware images and extracted their file systems for online discovering firmware of embedded devices. Maskiewicz et al. [31] found the inject malicious codes in the firmware image about the USB input of embedded devices. PIE [32] proposed machine learning algorithms to find parsers with external input for discovering vulnerabilities in the dynamic analysis. Our work focuses on discovering the taint-style vulnerability, which is one common weakness lacking security sanitization in the data propagation. Chen et al. [21] presented FIRMADYNE tool, which can automatically crawl firmware images from the vendor websites and analyzed the filesystem. Both works focused on simple vulnerabilities in the firmware, such as weak passwords. DTaint leverages the sophisticated static analysis to detect the taint-style vulnerabilities in the firmware.

There are several work about firmware simulation for the dynamic analysis. Decaf [33] proposed to use multi-target in the virtual machine for speeding up the dynamic analysis.

BINWALK [25] is a tool to unpack the firmware image to obtain the kernel filesystem. However, incomplete, encryption or unrecognized firmware can lead to its failure, and only a small fraction of firmware images supports to unpacking successfully. In our research, we have found more than 65% firmware images can not be successfully unpacked. After unpacking, QEMU [34] is a favorite tool for simulate firmware images, which runs the virtual host with supporting the architecture (i.e., ARM or MIPS) and endianness (i.e., big-endian or little-endian format). For dynamic analysis, AVATAR [35] can simulate firmware images through the interaction of real hardware. However, AVATAR needs to interact with real devices through JTAG or UART. In practice, many embedded devices do not open those interfaces, impeding the large-scale firmware analysis. In contrast, our work is orthogonal to those prior works. DTaint can detect the vulnerabilities in the proprietary firmware without the simulation ability.

Vulnerability Discovery. Vulnerability detection is a long-standing topic in security research across a wide range of approaches and techniques. There are two general categories for discovering vulnerabilities, including static analysis and dynamic analysis. We only review static analysis approaches related to DTaint.

There are several prior works that discover vulnerabilities at source code analysis level. Pixy [36] and phpSAFE [37] proposed to match vulnerability patterns through analyzing program structure and syntax. Jovanovic et al. [38] presented a precise alias analysis to detect cross-site scripting vulnerabilities in PHP scripts. Julia L et al. [39] proposed an approach to bug finding in Linux OS code using a control-flow based program search engine. Hossain et al. [40] classified static analysis-based BOF vulnerability detection approaches based on six features. Machine learning algorithms are usually used to analyze the complex source codes for identifying the vulnerability [41] [42]. Yamaguichi et al. [16] proposed the code property graph based on the syntax, control flow, and data dependence relationships. The vulnerability can be identified by matching template through traversing code property graph. Furthermore, Yamaguichi et al. [3] combined sources, paths, and sinks as regular expressions as automatically inferred various vulnerability types through searching. However, many vendors do not provide source codes for their firmware, leading to the failure of the static analysis. In contrast, DTaint has applied the static analysis to the binary firmware without any source codes.

Several prior works focused on identifying vulnerabilities at the binary code level rather than in the source code. CodeSurfer tool [43] provided pointer analysis, taint analysis capabilities and indirect function call checkers, which is similar to DTaint. CodeSurfer is for x86 machine code, while most firmware images are based on ARM or MIPS architecture. We cannot conduct comparison evaluation for DTaint, because CodeSurfer is not an open source tool. Rawat et al. [44] [45] proposed to detect taint-style vulnerability through combining call graph slicing and summary-based interprocedural data-flow analysis. The indirect call among functions is neglected for those work. Egele et al. [46] proposed to re-construct the CFG of binary of iOS applications, and detected privacy leaks through a standard data flow analysis. The method focuses on objective-C binary. Binary firmware on the ARM/MIPS architecture contains lots of pointer alias and indirect calls. In contrast, DTaint utilizes the similarity of data structure to find those indirect calls and solves the pointer alias for embedded firmware.

Similarly, Firmalice tool [47] analyzed the binary files of the firmware for discovering the authentication bypass vulnerability. DTaint targets a different goal, compared with Firmalice. We cannot apply Firmalice to obtain the taint-style vulnerability in the binary firmware. Angr [19] can analyze the binary file, through integrating symbolic execution, VEX IR, and program dependency graph. DTaint uses Angr as a building component. DTaint generates improved data flows in the binary file with the help of pointer alias, similarity of data structure, and interprocedural data flows.

VII. Conclusion

As online embedded devices play a crucial role in our daily lives, discovering unknown vulnerabilities in these devices is a prerequisite for providing security protection for the cyberspace. Unfortunately, the most firmware of embedded devices is closed source, and the state-of-the-art tools can simulate them successfully. In this paper, we propose DTaint for detecting the taint-style vulnerabilities in the proprietary firmware. DTaint eliminates the influence of the complex binary file for discovering data dependency in the program. DTaint generates the dependency between variable symbols in the memory through identifying interprocedural data flows, pointer alias and similarity of the data structure layout. We have implemented a prototype system of DTaint and conducted real-world experiments. Results show that DTaint can discover more taint-style vulnerabilities within less time cost, compared with the conventional DDG. Furthermore, we use DTaint to analyze six firmware images across four device vendors and found 21 vulnerabilities, where 13 of them are previously-unknown and zero-day vulnerabilities.

VIII. Acknowledgement

This work was supported by the National Key R&D Program of China (No. 2016YFB0800202), the Key Program of National Natural Science Foundation of China (No. U1766215), National Natural Science Foundation of China (No. 61602029), the Science and Technology project of State Grid Corporation of China (No. 52110417001B), and the Research Supported by IIE, CAS International Cooperation Project (No. Y7Z0461104).

References

[1] Gartner. (2017) Trend prediction of iot devices. [Online]. Available: https://www.gartner.com/newsroom/id/3598917

[2] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[3] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[4] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[5] N. Jovanovic, C. Kruegel, and E. Kirda, "Static analysis for detecting taint-style vulnerabilities in web applications," *Journal of Computer Security*, vol. 18, pp. 861–907, 2010.

[6] Heartbleed. (2014) The vulnerability in the cryptographic library openssl. [Online]. Available: http://heartbleed.com/

[7] Senrio. (2017) The vulnerability in gsoap. [Online]. Available: http://blog.senr.io/devilsivy.html

[8] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, Mar. 2012.

[9] H. Shahriar and M. Zulkernine, "A fuzzy logic-based buffer overflow vulnerability auditor," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, Dec 2011, pp. 137–144.

[10] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22Nd USENIX Conference on Security (SEC)*, 2013.

[11] A. Slowinska and H. Bos, "Pointless tainting?: Evaluating the practicality of pointer tainting," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, 2009.

[12] I. Doudalis, J. Clause, G. Venkataramani, M. Prvulovic, and A. Orso, "Effective and efficient memory protection using dynamic tainting," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 87–100, Jan 2012.

[13] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[14] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the Sixth Conference on Computer Systems (EuroSys)*, 2011.

[15] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "Concolic execution on small-size binaries: Challenges and empirical study," in *Ieee/ifip International Conference on Dependable Systems and Networks (DSN)*, 2017.

[16] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*, 2014.

[17] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011.

[18] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, 2008.

[19] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016.

[20] Q. Li, X. Feng, H. Wang, Z. Li, and L. Sun, "Towards fine-grained fingerprinting of firmware in online embedded devices," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2018.

[21] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[22] OpenWrt. (2010) a linux distribution for embedded devices. [Online]. Available: https://openwrt.org/

[23] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[24] S. S. Muchnick, "Advanced compiler design and implementation," in *Morgan Kaufmann Publishers Inc*, 1997.

[25] Binwalk. (2010) The tool for analyzing, reverse engineering, and extracting firmware images. [Online]. Available: http://binwalk.org/

[26] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 380–394.

[27] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *SIGPLAN Not.*, vol. 47, no. 4, pp. 265–278, Mar. 2011.

[28] CVE. (2016) Common vulnerabilities and exposures of the website. [Online]. Available: http://cve.mitre.org/

[29] exploit db. (2016) Exploit database of the website. [Online]. Available: https://www.exploit-db.com/

[30] T. B. Tok, S. Z. Guyer, and C. Lin, "Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers," in *Proceedings of the 15th International Conference on Compiler Construction (CC)*, 2006.

[31] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, "Mouse trap: Exploiting firmware updates in usb peripherals," in *Proceedings of the 8th USENIX Conference on Offensive Technologies (WOOT)*, 2014.

[32] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "Pie: Parser identification in embedded systems," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[33] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, 2014.

[34] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, 2005.

[35] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[36] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (SP)*, May 2006.

[37] P. J. C. Nunes, J. Fonseca, and M. Vieira, "phpsafe: A security analysis tool for oop web application plugins," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2015.

[38] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006.

[39] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller, "Wysiwib: A declarative approach to finding api protocols and bugs in linux code," in *2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2009.

[40] H. Shahriar and M. Zulkernine, "Classification of static analysis-based buffer overflow detectors," in *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, June 2010, pp. 94–101.

[41] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "Autoises: Automatically inferring security specifications and detecting violations," in *Proceedings of the 17th Conference on Security Symposium (SS)*, 2008.

[42] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, March 2016.

[43] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86: A platform for analyzing x86 executables," in *Compiler Construction, International Conference, Cc 2005, Held As*, 2005, pp. 250–254.

[44] S. Rawat, L. Mounier, and M. L. Potet, "Listt: An investigation into unsound-incomplete yet practical result yielding static taintflow analysis," in *2014 Ninth International Conference on Availability, Reliability and Security*, Sept 2014, pp. 498–505.

[45] P. M. L. Rawat S, Mounier L, "Static taint-analysis on binary executables," 2011.

[46] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *Network and Distributed System Security Symposium (NDSS)*, 2011.

[47] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.