

# Hunting the Ghost: Towards Automatic Mining of IoT Hidden Services

Shuaike Dong

The Chinese University of Hong Kong  
1155085891@link.cuhk.edu.hk

Siyu Shen

The Chinese University of Hong Kong  
ss019@ie.cuhk.edu.hk

Zhou Li

University of California, Irvine  
zhou.li@uci.edu

Kehuan Zhang

The Chinese University of Hong Kong  
khzhang@ie.cuhk.edu.hk

**Abstract**—The Internet of Things (IoT) has been prosperously used and profoundly changing people’s daily life. However, for normal users, most IoT devices are smart “black boxes” listening to their commands and providing feedbacks. Recent notorious IoT-based attacks have shown that even IoT devices themselves may behave normally, evil things can be happening under the hood. Therefore, it is crucial for both normal users and security analysts to quickly profile the behavior of an IoT device and dig out “open” and “half-open” doors in it. In this paper, we proposes IoT Bolt, an automatic firmware analysis tool targeting at finding hidden services that may be potentially harmful to the IoT devices. Our approach uses static analysis and symbolic execution to search and filter services that are transparent to normal users but explicit to experienced attackers. A prototype is built and evaluated against a dataset of IoT firmware, and The evaluation shows our tool can find the suspicious hidden services effectively.

## I. BACKGROUND

### A. Smart-Home IoT Architecture

A typical smart home consists of three parties: IoT device, service provider and user. The normal interaction scheme is shown in fig . Below we briefly introduce it.

### B. IoT Firmware

IoT firmware is the software programmed on IoT hardware. Since IoT products are usually embedded devices with limited computation resources and variant architectures. IoT firmware provides basic functionalities to interact with the hardware.

**Firmware layout.** There are two typical types of firmware, *firmware blobs* and *embedded-Linux based firmware*. Firmware blobs often come with bare-metal devices with very monotonous capabilities and restricted resources, like smart bulbs and smart plugs. Firmware blobs usually have various formats due to the proprietary development process of their vendors.

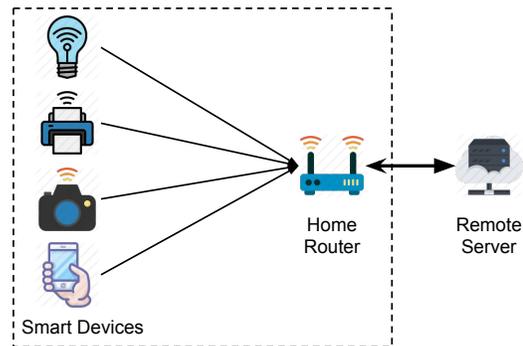


Fig. 1. A typical smart home

According to previous works [22], embedded-Linux based firmware takes up the largest portion of off-the-shelf firmware with a ratio of around 86%. Contents of an embedded-Linux based firmware vary among different vendors. In most cases, there is at least one embedded file system inside and can be extracted by firmware unpacking tools like *binwalk* [14].

**Firmware acquisition.** There are generally three ways to obtain a firmware.

1) Download from websites of the vendor. Several vendors release the firmware dataset on their websites, like *D-Link* [5] and *TP-Link* [11].

2) Extract firmware from hardware. Firmware often resides in the storage unit of an IoT device such as flash chip. With microcontrollers that supports flash communication protocols, one can dump the firmware without destroying the device [9]. Apart from flash chip dumping, other hardware debugging ports can be leveraged for the firmware extraction as well [8].

3) Network interception. Some vendors provide mobile Apps to facilitate the control of IoT devices. Most of these Apps can trigger the firmware upgrade by one or several simple operations. During the firmware upgrading process, either the meta-data (including download url, file size and checksum, etc.) or the firmware itself will be transmitted to the device. An experienced analyst can easily intercept the traffic he/she needs by setting up a proxy.

### C. Issues in IoT Development

As the development of a commercial IoT product is a long and lasting task during which the code may experience a lot of changes and updates from different developers. Many issues may arise during the process and thus produce some "hidden" services, which may be utilized by malicious attackers. We classify the factors to these risks as follows:

- 1) **Leftover Debugging code.** As the modern software development is usually a complicated process including multiple coupling components, developers widely use debugging to locate the errors and fix them. Different from codes released in production version, debugging code usually involves a wider capability and higher privilege, or some functionalities that are just for testing usage. Apart from that, the debugging codes are usually not directly visible to normal users but can be easily found by experienced and purposeful attackers.
- 2) **Unmatched interfaces and services.** To facilitate normal users in using IoT devices, vendors usually provide easy-to-use interfaces for them. For example, most domestic routers provide a series of web pages leading users to configure the product step by step. As those pages are just interactive entrances to back-end running programs, we call them *interfaces* and their targets as *services*. From the perspective of a normal user, interfaces and services should have a "surjection" relation which means each service should have at least one interface to trigger it. However, there are cases when multiple services running behind the scene without any companion interfaces. Apart from that, even those "already-matched" services may contain hidden functions that are not within the scope of their interfaces, which are exploitable by experienced attackers.
- 3) **Backdoors.** Previous work [25] has shown that backdoors are pervasive in IoT devices. From the perspective of vendors, backdoors are helpful for debugging usage and testing phase during the development. However, they can be taken advantage of by attackers and thus leading to the compromise of the whole device or privacy exposure.

### D. IoT interfaces and services

As the medium between normal users and the environment, IoT devices provide easy-to-use interfaces for consumers. We categorize common interfaces as follows:

- 1) **Web-based interfaces.** Some IoT devices embed web pages inside the firmware to help users do the configuration. For example, home-use routers often host the domain "http://192.168.0.1" by a web server. People access that domain and set up the router step by step according to its wizard. According to our study, some IoT cameras, also support such kind of interface.
- 2) **Mobile-based interfaces.** More and more IoT developers are releasing their companion mobile Apps with devices. A user yields commands by operating the UI

controllers of the mobile App, the commands will be transferred to the cloud or directly sent to the device in the form of network packets.

- 3) **Physical contacts.** Most IoT devices provide physical interfaces to make them consistent with previous non-IoT devices, like traditional plugs and sockets. Some physical actions like long-pressing a button also act as the critical role in the binding process, like [21] shows.
- 4) **Others.** There are additional interfaces utilizing different kinds of signals. For example, Google home [15] responds when it hears legal voice commands. Some smart thermometers commits certain behaviors when the temperature has reached a threshold.

After a user request is well received by the device, it is processed by the software components of the IoT firmware. Due to the variety of the information involved inside requests, different handlers are chosen to react. We refer to reactions taken by the IoT devices as *services*, which are IoT functionalities that can be aroused through IoT interfaces.

To conclude, IoT services hold the following attributes:

- 1) can be triggered through IoT interfaces or channels with the equal semantics. For instance, a service of a home router can be activated either by interacting with the web interface or by a manually-crafted network packet.
- 2) can reveal information to the sender of a request.
- 3) can change the device status.

**Hidden Service.** Due to the "black box" feature of an IoT device, we differentiate a hidden service from a normal one by *whether it is transparent to normal users*. To simplify, we assume normal users have no enough knowledge of network security and can only operate IoT devices following specifications. For example, when given a home-use router, a normal user only performs the following steps:

- 1) Plug in necessary wires and turn on the router.
- 2) Open the browser and access "http://192.168.0.1", input default credentials on the specification to authorize.
- 3) Follow the Setup Wizard and configure the router.
- 4) Connect his/her electronic devices to the WiFi and surf the Internet.

He or she is unaware of more advanced techniques like penetration tests. According to our definition, all the functionalities provided outside the scope of "http://192.168.0.1" are hidden from the user, and therefore are hidden services.

Note that hidden services do not certainly mean vulnerabilities. However, the existence of hidden services brings unnecessary redundancy to the IoT system and has negative influence on its robustness and security.

## II. SYSTEM OVERVIEW

*IoTbolt* aims at finding hidden services in IoT devices. In this section, we introduce the high-level idea of our tool and how it works in an automatic way.

Figure 2 shows the overall structure of *IoTbolt*. From it we know, *IoTbolt* takes 4 steps analyzing each input firmware sample.

- 1) **Firmware Unpacking.** `IoTbolt` works on Linux-based firmware. We use the popular firmware extraction tool `binwalk` to unpack the firmware. After a successful unpacking, all files in the original file system (`squash-fs`, etc.) of the firmware are restored.
- 2) **Files Filtering.** Step 1 will recover most of the files inside a file system, including executables, libraries, configuration files, etc.. However, not all of them are needed by `IoTbolt`. `IoTbolt` will recognize all web-based interface files, including those with extensions like `.html`, `.js`, `.php`, `.asp`. `IoTbolt` will also identify executables handling user requests and provide services, which we call *service binaries*. The detailed algorithm we use is illustrated in section III.
- 3) **IoT Services Recognition.** After locating *service binaries*, `IoTbolt` analyzes them in a static way to find and summarize the provided services. At this step, `IoTbolt` also prepares for the symbolic execution by identifying the start and end points.
- 4) **Normal User View Emulation.**
- 5) **Hidden Services Detection.**

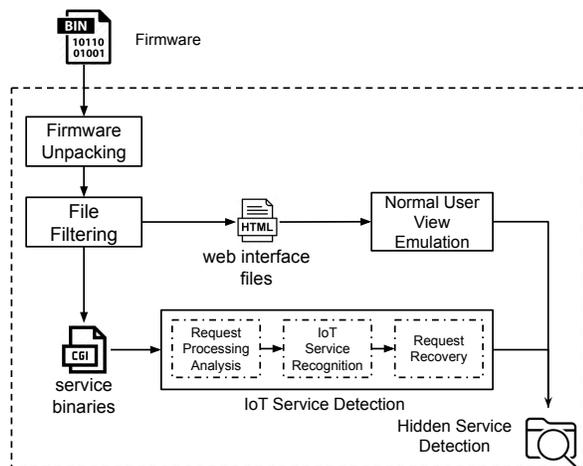


Fig. 2. The workflow of `IoTbolt`

### III. FILES FILTERING

`IoTbolt` takes a straightforward name-based approach to filtering the files needed.

**Document Root Location.** Current web servers usually hold most of their web interface files in a certain directory, called *Document Root*. Files under the Document Root directory are what a user sees when he/she is visiting a website.

To recognize the document root directory, we count the number of files with extensions like `.html`, `.js`, `.php`, `.asp`, `.css`. The directory holding the largest quantity of web interface files is regarded as the document root.

**Service Binary Recognition.** We categorize the *service binaries* into two folds: *server-like binaries* and *module-like binaries*. These two types mainly differentiate at *whether it can receive/send network requests/responses individually*. Except

from web servers, there are some daemon programs belong to server-like binaries. They provide other interfaces, like another open port rather than 80 and 443, to interact with users.

Module-like binaries mainly refer to Common Gateway Interface (CGI) files written in C programming language. These files can interact with users but only with CGI-modules enabled by web servers.

We recognize service binaries by counting the occurrences of a set of network-related strings. We also perform a scanning in `/etc` folder, if those service binaries are found in those initialization files, like `/init.d/rcS`, we assume they have larger chances to be service binaries and prioritize them in the analyzing queue.

### IV. REQUEST PROCESSING ANALYSIS

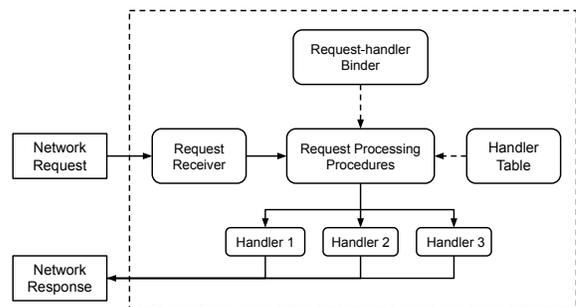


Fig. 3. The procedure of processing a network request

As section I-D says, services can be triggered only when requests received at the firmware side can be successfully processed. Figure 3 shows a normal process of handling a network request. At first, the request is captured by a *request receiver*, which is often realized by standard C functions like `recv` and `recvfrom`. After that, the request is parsed and checked by *request processing procedures*.

A request processing procedure examines network messages by traversing a tree-like structure. At each internal node of the tree, a certain field inside the message is checked and different values will lead to different paths until a corresponding service is invoked. Figure 4 shows the control flow graph of a typical request processing procedure.

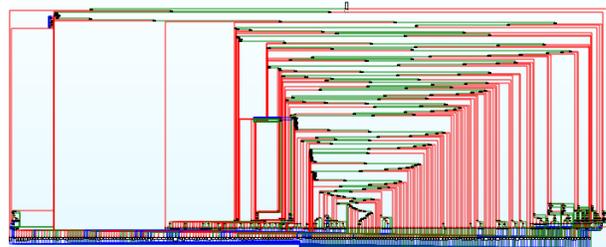


Fig. 4. The overview graph of a typical request processing procedure (by IDA)

Previous works have already proposed several approaches to identifying request processing procedures. For example,

Karonte [24] leveraged five features to evaluate whether a function implements a request processing procedure or not, including (1) the number of basic blocks, (2) the number of branches, (3) the number of conditional statements used in conjunction with memory comparisons, (4) *network mark*, the number of network-related strings that are used by memory comparisons and (5) *connection mark*, whether the data received from a network socket is used in memory comparisons.

Karonte’s approach has moved a big step forward to the automatic recognition of network-related parsing functions. However, we find there are still some limitations that are not fixed, which are listed as below:

- 1) Some embedded web servers provide APIs to associate a URL with a specific handler. For example, *GoAhead* [7] provides an API called *websUrlHandlerDefine* whose first parameter is a string representing the target URL and the second parameter is a processing function object, whenever the target URL is detected, the corresponding handler will be invoked to process the request. We refer to such functions as *request-handler binders* since server itself will initiative deliver requests to their handlers without querying a tree-like structure.
- 2) The heavy-weight approach to calculate the *connection mark* can easily fail. According to the details of Karonte, we find it applies symbolic-execution-based taint analysis to trace the data flow from standard socket APIs, like *recv* and *recvfrom* to memory comparisons like *memcmp* and *strcmp*. This approach can work when the program being analyzed is small-sized. However, for a program with a large size and many *indirect control flow transitions*, the approach can easily get lost and stops in the half way. The following code snippet shows how *boa* hinders the taint analysis with the usage of a message queue.

To fix the limitations above, we propose the following improvements to Karonte’s approach based on our observations, which are summarized as below:

- **Observation 1.** A request-handler binder usually contains a network-related string and a function object as parameters. Apart from that, request-handler binders are usually invoked for multiple times to bind different URLs with their handlers.
- **Strategy 1.** We propose another two features, of which the first recording whether a network-related string and a function object appear together as the arguments of a subroutine, the second being the number of invocations to this subroutine.
- **Observation 2.** For a request processing subroutine with a tree-like structure, the request or the object representing the request is usually stored in function arguments or global variables.
- **Strategy 2.** Instead of tracking the data flow from standard socket APIs, we determine to apply intra-procedure taint analysis on the subroutine. *IoTBoIt* checks if the parameters and the global variables appearing inside the

address	name	handler
0x58c560	“SetMultipleActions” (0x4a6620)	sub_433768
0x58c568	“GetDeviceSettings” (0x4a6634)	sub_423d28
0x58c570	“GetOperationMode” (0x4a6648)	sub_433f70
0x58c578	...	...
0x58c888	0x00	0x00

TABLE I  
A TYPICAL FUNCTION TABLE IN MEMORY.

subroutine are finally used by memory comparisons.

**Function table recognition.** Apart from the tree-like structure, *function tables* are often used to find corresponding handlers. A function table is often composed of multiple compactly-stacked items. Each item has a key(name) and a value(handler address). Once a key matches that of a request, the corresponding handler will be invoked. Table I shows a real function table we extract from our firmware database and Listing 1 illustrates how function table is usually used inside a function.

---

**Algorithm 1** Abstract of function table usage

---

**Require:** *url* ← the url to access

- 1: *pointer* ← &*func\_table*
- 2: **while** *\*pointer* ≠ zero\_byte **do**
- 3:   **if** *\*pointer* == *url* **then**
- 4:     *handler* ← *\*(pointer + 4)*
- 5:     CALL(*handler*)
- 6:   **else**
- 7:     *pointer* = *pointer + 8*
- 8: **end while**

---

Locating function tables can significantly enrich the services we can find. Based on our observations, function tables are often accessed inside loop structures. To this end, we take the following steps to locate them.

- 1) Identify *loop* structures inside a function body. This can be done by *angr*’s built-in analysis modules.
- 2) Partially symbolic-execute the loop until an ending (usually byte zero) is encountered. Collect all memory addresses accessed during the execution and store them into a list [*addr*<sub>1</sub>, *addr*<sub>2</sub>, ..., *addr*<sub>*n*</sub>].
- 3) Generally, the collected memory addresses are those pointing to name strings, therefore, we calculate the *stride* between *addr*<sub>*i*</sub> and *addr*<sub>*i*+1</sub> and get a second address list [*addr*<sub>1</sub> + *stride*/2, *addr*<sub>2</sub> + *stride*/2, ..., *addr*<sub>*n*</sub> + *stride*/2]
- 4) For each address inside the second address list, we match the disassembly code at that address against the built-in function prologues of *angr*. A successful match implies an identification of a function table item, and we finally obtain a function table containing multiple jumping targets.

## V. IOT SERVICES RECOGNITION

Services are the most critical part of an IoT device due to their usability. In this section, we introduce how *IoTBoIt*

recognize the services within a firmware. Note that the “service” in our paper emphasizes on its availability, not the concrete behaviors that can benefit people’s daily lives, like “playing a song” and “turning on the light”.

### A. Service Categorization

Referring to attributes of services listed in section IV, we summarize the services into four categories:

**persistent output.** One of the main capabilities of IoT devices is to feed back users with information they have asked for. After manually analyzing multiple firmwares from different vendors, we conclude three strategies that are commonly used in network-related binaries.

- 1) For Common Gateway Interface (CGI) files, information is directly output by standard output functions, like *printf* and *puts* in C programming language, *echo* in *Linux bash*.
- 2) For non-CGI files, especially web server binaries, information is usually yielded to files using stream output functions, like *fprintf* and *fpputs*. Whether the user can access the information is determined by the authentication and authorization settings of that binary.
- 3) Some developers leverage third-party libraries to deliver data of different formats from plain HTML, like Extensible Markup Language (XML) and JavaScript Object Notation (JSON). For example, the web server file *goahead* in a D-Link firmware<sup>1</sup> widely uses functions from library *libxml.so* to get/set values inside a XML object.

**system command execution.** Since a firmware contains multiple binaries with different capabilities, there are often cases when a user’s request can only be fulfilled by executing another binary. Apart from that, Linux provides a lot of easy-to-use but powerful commands, which are frequently-used in IoT firmwares, like *reboot*, *sleep* and *echo*, etc. For example, when a user uploads a new version of firmware to the home router, the router will first store the new firmware in a temporary area and changes the bootloader settings. After that, a *system("reboot")* will be executed to make the device reboot and load the newer firmware.

According to our observation, the most commonly-used system command execution APIs are *system* and *exec* function family, including *execl*, *execv*, etc.

**non-volatile storage operation.** Non-volatile storage is pervasively used in IoT devices due to its small size and low power consumption [13]. As the physical carrier of firmware itself, non-volatile storage is often used for device configuration and user personalization [1]. Developers can also control the non-volatile storage with the help of non-volatile memory libraries(*libnvram.so*, etc.) like operating a normal key-value storage. Different from a normal key-value storage, the updates is stored globally and can be shared among different binaries.

**network activities.** We conclude two scenarios when IoT devices create network activities. (1) **Network activities be-**

service category	APIs	library needed
persistent output	<i>printf</i> , <i>printf_s</i> , <i>puts</i>	C standard library
	<i>fprintf</i> , <i>fprintf_s</i> , <i>fpputs</i>	
	<i>fopen</i> , <i>fopen64</i>	
	<i>open</i> , <i>open64</i>	
system command execution	<i>system</i> , <i>execl</i> , <i>execl</i> , <i>execlp</i> , <i>execv</i> , <i>execve</i> , <i>execvp</i> , <i>popen</i>	
nonvolatile storage operation	<i>nvram_set</i> , <i>nvram_update</i> , <i>apmib_set</i> , <i>apmib_update</i>	<i>libnvram.so/apmib.so</i>
network activities	<i>bind</i> , <i>connect</i> , <i>SSL_connect</i> , <i>SSL_read</i> , <i>SSL_write</i>	C standard library; <i>libssl.so</i>

TABLE II  
TYPICAL APIs OF FOUR IOT SERVICES

**tween IoT devices and remote servers.** Due to the restricted resources of an IoT device, some of its functionalities can only be accomplished with the help of remote servers. For example, a smart assistant like Google Home and Amazon Alexa have to interact with remote servers to answer users’ various questions. (2) **Network activities between IoT devices and terminal users.** One typical case is network cameras with real time streaming protocol(RTSP) supported. When the user accesses the RTSP channel of the camera, the camera will start sending RTSP packets continuously.

### B. Service Recognition

We locate services by searching for APIs with correct contexts, which we call **end points**. The selected APIs are listed in table II. Note the API list can be extended when some third party libraries are used in the development.

**API context.** Even for the same set of APIs, there is no guarantee that the consequences of their usage are the same. For example, although *printf* can output information as HTML codes directly in CGI files, non-CGI files do not have such feature, where *printf* is frequently used to yield messages to the console, for the aim of logging and debugging. So is stream output function *fprintf* [6]. Given its prototype – *int fprintf(FILE \* stream, const char \* format, ...)*; we know the first parameter plays a big role in determining the behavior of *fprintf*. The following piece of code gives an example. From it we see, although *fprintf* writes “something” to the file *\_\_stream*, however, *\_\_stream* refers to */dev/console* that does not deliver any information to the user’s screen.

*IoTbolt* deals with this by considering and recovering the context of API call sites. For *printf*-like functions, *IoTbolt* determines whether it is an end point by the type of the file, in our case, CGI or non-CGI. For *fprintf*-like functions, *IoTbolt* first searches for corresponding file handling functions, like *open* and *fopen*, in the vicinity of *fprintf* call sites. After that, *IoTbolt* recovers its arguments, only when the argument *filename* is not the console device and the argument *access mode* includes “writing” capability, will the invocation be regarded as a legal end point.

```

1 __stream = fopen("/dev/console", "w");
2 fprintf(__stream, 'SCRIPT NAME{%s}\n', __s1);

```

Listing 1. *fprintf* output to console

<sup>1</sup>DLink DIR823G\_V1.0.2B05\_20181207

**API encapsulation.** Another issue we consider is the encapsulation of APIs. According to our observations, dynamically linked shared object libraries may introduce functions with the same capabilities but different names from the APIs listed in table II. For example, we found *libputil.so* of the firmware *D-Link DIR-505 v1.09B02* defines a function called *\_system*, which encapsulates the API *system* by adding a call to *vsprintf* in the front. Two functions behave in the almost same way but are invoked interchangeably in other binaries, as shown in the listing 2. To fix this, `IoTbolt` scans the whole file system(in the firmware) looking for dynamically linked shared libraries used by the binary being analyzed, if there are any function found to be the encapsulation of the APIs in table II and are invoked inside the current binary, `IoTbolt` add them to the end points list.

```

1 int _system(char* param_1, char* param_2)
2 {
3     char arr[520];
4     int var = vsprintf(arr, param_1, param_2);
5     system(arr);
6     return var;
7 }

```

Listing 2. *printf* output to console

## VI. REQUESTS RECOVERY BY SYMBOLIC EXECUTION

After identifying services inside a binary, `IoTbolt` tries to recover the network requests that can trigger them. In this section, we describe the challenges we met in recovering the requests and how we overcome them.

### **Challenge 1. Determine start points of symbolic execution.**

As a popular software testing approach, symbolic execution engine needs to specify an entry point to start the simulation. By default for a C-language program, the *main* function will be set as the entry point. However, for a lot of complex binaries like web servers, *main* function is not the optimal solution, since there are a lot of initialization and pre-processing tasks within the *main* function which can be the burden to the symbolic execution.

**Solution 1.** The intuition to solve challenge 1. is that *request processing subroutine* is the most important in recovering the request. If a request can trigger the service along a certain path, the *request processing subroutine* must be on that path as well.

Following this, we first generate a call graph for the whole binary. For each end point collected as the section V shows, we perform a backward tracing from it until a function with no predecessors is found. Among all functions in the trace, we select the one holding the highest probability of being the *request processing subroutine*, and take it as the start point for that end point.

### **Challenge 2. State explosion during symbolic execution.**

State explosion has been a long-term challenge when applying symbolic execution to practical projects [17]. For network-related binaries, the issue becomes obvious due to the extensive existence of loop-like structures and deep function call traces.

*%dskto* think about why our work is more difficult

**Solution 2.** Our high-level idea of solving challenge 2. is to leverage static analysis to guide the symbolic execution. To be specific, we statically generate a set of *chopped control flow graphs* for each pair of start point and end point. The graphs are used to guide the system in avoiding unnecessary and unrelated paths during the symbolic execution. The detailed algorithm is introduced in section VI. Apart from that, we take the strategy of *high-level constraints recording*. On one hand, this strategy helps the symbolic execution engine bypass functions that have no or little impacts on request recovery, on the other hand, provide security analysts with clearer information about the request contents rather than byte- or bit-level memory constraints.

### A. Multi-stage Chopped Control Flow Graphs

As the aim of symbolic execution engine is to rapidly explore the paths from a start point to an end point and recover the critical information inside the request body that leads to it, strategies should be taken to guide the symbolic engine to avoid unrelated paths. To achieve this, we leverages static analysis to generate multi-stage chopped control flow graphs before running the symbolic execution.

We referenced the concept of “chopped CFG” proposed by Brumley et al. [18]. To reduce the overhead of chopped CFG generation, we take advantage of function-level backward trace mentioned in Solution 1. To be specific, given one pair of start point and end point, instead of calculating chopped CFGs from the whole binary, we decompose the task into multiple sub-tasks. That is to say, for each caller and callee along the backward trace, we set the caller’s address as the start point and the callee’s call site as the end point, then calculate the sub-chopped-CFG of them. After iterating the whole trace, we compose all sub-chopped-CFGs to get the *multi-stage chopped CFG*. Taking advantage of the backward trace significantly reduce the time spending on getting chopped CFGs.

During the symbolic execution, the current symbolic state will be checked against the multi-stage chopped CFG at each step. If the basic block being accessed by the current state is inside its basic block set, we assume the state is moving towards the target end point in the right direction, otherwise, the state will be dropped for the sake of appropriate memory cost. Listing 2 shows the detailed algorithm.

### B. High-level Constraints Recording

Due to the exploration scheme mentioned in section VI-A, `IoTbolt` limits its symbolic execution to codes that directly deals with request parsing and service triggering. This can not fully simulates the running of a real-world network-related binary. For instance, a web server usually does some initialization work during its start up, like reading configurations from local file system and setting up necessary environment variables. Due to such “tedious” jobs are usually not included inside the selected code paths for `IoTbolt`, the symbolic solver is prone to fail in determining the possible values when the constraints contain those unknown values.



if two basic blocks are connected in the control flow graph, the constraint of the successor node will be dependent on that of the predecessor node.

```

1 char* __nptr = (char*)websGetVar(param_1, "
  start_time");
2 char* __nptr_00 = (char*)websGetVar(param_1,
  "end_time");
3 int iVar5 = strcmp(__nptr, "0");
4 if (iVar5 == 0){
5     iVar5 = strcmp(__nptr_00, "86400");
6     if (iVar5 != 0) goto LAB_0046e540;
7     else{
8         //do something at 0x46e534
9     }
10 }

```

Listing 4. An example of high-level constraints recording

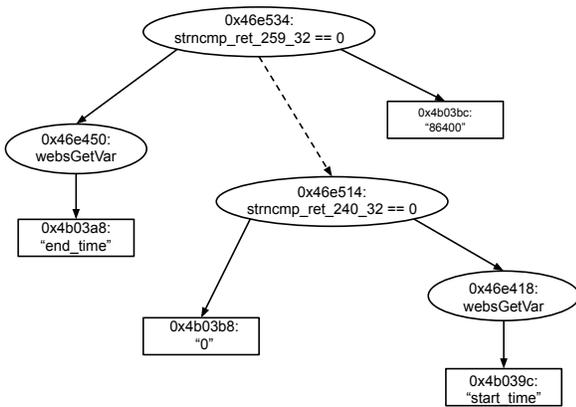


Fig. 5. The high-level constraints collected by IoTbolt

## VII. NORMAL USER VIEW EMULATION

For the sake of finding services that are transparent to normal users. It is essential to realize what a normal user sees at the client side. In this section, we illustrate the hybrid approach of IoTbolt to do normal user view emulation.

Previous work [23] has summarized several approaches of running web interfaces. Different from their scenario of penetrating the firmware by web interfaces, we only care about thoroughly collecting network requests that can be triggered by normal users, and leave the CGI-like binaries to the symbolic execution module of IoTbolt, and therefore, we propose the following strategies:

**Strategy 1. Local Web Server Hosting.** We referenced the idea of how to run web interfaces in [23]. After locating the document root in the file filtering phase, IoTbolt launches a simple web server under the document root directory using the *Express* module provided by *node-js* [10]. With the web server, all *html* and *js* files can be accessed. IoTbolt then takes a DFS-based iteration approach to traverse all these interface files and collect generated network requests.

**Strategy 2. Qemu Emulation.** Strategy 1 can only solve the issue of *html* and *js* file emulation. For web interface files that need the support of certain modules, like *asp* and *php*, we

decide to take advantage of tool *Firmadyne* proposed by Chen et al. [19]. *Firmadyne* provides a system-level emulation given a firmware. Using this approach, *asp* and *php* files can both run due to the successful recovery of its web server environment. **Strategy 3. Static Analysis.** *Firmadyne* has a limitation of run-time successful rate due to various layouts of firmware samples. When *Firmadyne* fails to emulate a firmware, we leverage the signature-based static analysis to uncover user requests.

## VIII. EVALUATION

We evaluate IoTbolt on 3 firmware from two different vendors, D-Link and Netgear. Two of them have been reported as vulnerable. Our goal is to test whether we could find security-critical issues with the help of IoTbolt.

### A. D-Link DIR-619L wireless N300 Cloud Router

D-Link has a long history in home-use router development and manufacturing. DIR-619L is a popular brand released by D-Link in 2012. According to statistics provided by [4], there are in total three vulnerabilities presented by previous researchers from 2015 to 2018, whose vulnerability types are all *Exec Code*.

We use the firmware of version 2.06.B01 to conduct the evaluation. After unpacking the firmware, IoTbolt first locates the document root path as */web* with 168 *asp* files, 3 *html* files and one *cgi* file. Unfortunately, both the strategy 1 and strategy 2 fail to emulate the normal user view, IoTbolt then takes the third approach – static analysis. After statically analyzing the web interface files, IoTbolt recovers 758 requests in total, including 662 *asp* requests and 96 *html* requests (html form request). In the table III, we show the top 5 most-frequently invoked *asp* handlers and *html* urls by this firmware.

ASP handlers	times	HTML URLs	times
getInfo	170	/goform/formEasySetupWizard	8
getIndexInfo	157	/goform/formdumpeasysetup	3
firewallRule_row	50	/goform/formSetEasy_Wizard	2
getWizardInformation	35	/goform/formWISiteSurvey	2
staticRouteList	32	/goform/form_mydlink_sign	2

TABLE III  
TOP-5 MOST-WIDELY INVOKED ASP HANDLERS AND HTML URLs IN DIR-619L

IoTbolt then runs its service recognition module and symbolic execution engine on the filtered service binary, which is *boa* in this case. The results show that all *asp* handlers and *HTML* urls are captured by IoTbolt due to the function used to define them *websAspDefine* and *websFormDefine* have been recognized as *request-handler binders*.

```

1 void websAspInit(void){...
2     websFormDefine("formSetHNAP11",
3         formSetHNAP11);
4     websAspDefine("getInfo",getInfo);
5     websAspDefine("getIndexInfo",getIndexInfo);
6     ... }

```

```

6 void formSetHNAP11(char* param1){...}
7 void getInfo(char* param1){...}
8 void getIndexInfo(char* param1,char* param2,
  char* param3){...}

```

Listing 5. The usage of *websAspDefine* and *websFormDefine*

We then analyze the results given by *IoTbolt* to see if there are “hidden” services inside the firmware. To our surprise, *IoTbolt* emits two alerts:

1) Accessing “/common/info.cgi”, “/version.txt” and “Dev-Info.txt” can lead to information disclosure, some of which are sensitive, like the WPS password of the current WiFi.

After checking the function invocation trace we find that this hidden service is of the type “persistent output” since a call to *open* with write capability(0x302) is captured by *IoTbolt*.

2) A function table structure is recognized at address 0x4f9014. The symbolic execution module of *IoTbolt* in total recovers 21 handlers from the function table. We then observe that this function table is looked up inside the form handler of url “formSetHNAP11”, which is also not used by web interface files. Our manual code analysis reveals that such services can be utilized to set new passwords or DoS the device.

#### B. D-Link DIR-823G Dual Band Smart Wireless Router

The firmware we use to evaluate *IoTbolt* is version A1 FW102B03, *IoTbolt* first recognizes a request-handler binder *sub\_40b1f4*. Tracking the usage of that function, we find handlers of four types of urls – “/HNAP1/\*”, “/goform/\*”, “/cgi-bin/\*”, “/EXCU\_SHELL”.

Different from DIR-619L, DIR-823G firmware can be successfully emulated by *Firmadyne*. Using the automatic triggering scripts, *IoTbolt* collects network requests generated by a normal user when accessing the web interface.

After comparing against the services found by *IoTbolt*, we find “/EXCU\_SHELL” is a hidden service which can be utilized to remotely execute arbitrary Linux commands. After searching recent CVE reports, we found the exploitation has been released in 2018 [3] but tested with different brands, which also reveals the code reuse issue among D-Link devices.

#### C. D-Link 8010LH Wi-Fi Camera

8010LH is a recent Wi-Fi camera released by D-Link. Different from other two samples, *IoTbolt* does not find any web servers inside the firmware, however, there are three service binaries found – “Rtk\_MainProc”, “StreamProxy” and “TW\_Monitor”. All of them are recorded inside the system initialization configuration file “/init.d/rcS” to launch during the starting process.

*IoTbolt* found one hidden service inside “Rtk\_MainProc” belonging to system command execution. Tracking the traces output by *IoTbolt*, we find that inside the “main” function of “Rtk\_MainProc”, when the binary detects that file *.tw\_enable\_telnet* exists, telnet service will be enabled by *system*(“telnetd &”). We further try cracking the credentials of *root* user and find the password can be cracked within two seconds.

*IoTbolt* finds another hidden service inside the service binary “StreamProxy” that creates new network activities. By tracing back from the call to “SSL\_connect”, *IoTbolt* finally recovers a HTTP packet that can establish an unauthenticated RTSP channel with a remote server. The recovered HTTP request is as the listing 6.

```

1 [LAN ip]:7000/command=rtsp_start_viewing
2 &rtsp_host=[remote IP]
3 &rtsp_port=[remote port]
4 &code=0

```

Listing 6. Recovered HTTP request

## IX. RELATED WORK

Firmware analysis has become a hot topic in recent years. Costin et al. [22] conducted a large-scale security assessment on a dataset with more than 320,000 firmware samples. Their results show vulnerabilities are pervasive in firmware and should be attached with enough notice. Shoshitaishvili et al. [25] proposed an automatic tool called *Firmalice* to detect authentication bypass vulnerabilities in a firmware. Their work shed a light on symbolically-executing firmware codes. Furthermore, their group presented a binary code analysis framework called *anqr* in [18], which is also the back-end of our framework. D. Chen et al. [19] presented a qemu-based emulation tool *Firmadyne*. The tool can globally simulate a firmware and well capture the behaviors of that firmware. However, it still suffers from the success rate, due to the variety in firmware layout. Redini et al. [24] proposed a taint-based static analysis tool capable of finding insecure multi-binary interactions. However, they get rid of all firmware sample of MIPS architecture, which takes up more than 79.4% in a recent released data set [19]. Different from their work, our framework can well handle MIPS firmware samples.

Apart from direct analysis on firmware binaries, researchers propose other approaches targeting at different interfaces. J. Chen et al. [20] presented a mobile-based fuzzing tool *IoTfuzzer*, which can trigger vulnerabilities of IoT devices by mutating mobile-side messages. Similarly, Wang et al. [26] proposed a novel approach of finding vulnerabilities inside firmwares by analyzing companion mobile apps.

## REFERENCES

- [1] “Advantages of Embedded MTP Non-Volatile Memory for IoT SoC Designs,” <https://www.synopsys.com/designware-ip/technical-bulletin/advantages-of-mtv.html>.
- [2] “Claripy,” <https://docs.angr.io/advanced-topics/claripy>.
- [3] “CVE-2018-19300: Remote Command Execution Vulnerability in D-Link DWR and DAP Routers,” <https://community.greenbone.net/t/cve-2018-19300-remote-command-execution-vulnerability-in-d-link-dwr-and-dap-routers/1772>.
- [4] “Dlink Dir619l Firmware: List of security vulnerabilities,” [https://www.cvedetails.com/vulnerability-list/vendor\\_id-899/product\\_id-31627/D-link-Dir-619l-Firmware.html](https://www.cvedetails.com/vulnerability-list/vendor_id-899/product_id-31627/D-link-Dir-619l-Firmware.html).
- [5] “DLink firmware repository,” <ftp://ftp2.dlink.com/>.
- [6] “fprintf-C++ Reference,” <http://www.cplusplus.com/reference/cstdio/printf/>.
- [7] “GoAhead Web Server,” <https://www.embedthis.com/goahead/>.
- [8] “How to extract firmware from a device,” <https://ianhowson.com/iot/extracting-firmware/>.

- [9] “Intro to Hardward Hacking - Dumping your First Firmware,” <https://nvisium.com/blog/2019/08/07/extracting-firmware-from-iot-devices.html>.
- [10] “Run JavaScript Everywhere,” <https://nodejs.dev/>.
- [11] “TP-Link Product Support - Download Center,” <https://www.tp-link.com/us/support/download/>.
- [12] “Z3Prover/z3,” <https://github.com/Z3Prover/z3>.
- [13] “Memory Options for the IoT,” <https://www.synopsys.com/designware-ip/technical-bulletin/memory-options.html>, Accessed: May 2020.
- [14] “ReFirmLabs/binwalk,” <https://github.com/ReFirmLabs/binwalk>, Accessed: May 2020.
- [15] “Google Home,” <https://home.google.com/welcome/>, Accessed: May 2025.
- [16] T. Amon and T. Loffredo, “Creating human readable path constraints from symbolic execution,” 01 2020.
- [17] R. Baldoni, E. Coppa, D. C. D’Elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, 2018.
- [18] D. Brumley, H. Wang, S. Jha, and D. X. Song, “Creating vulnerability signatures using weakest preconditions,” in *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*. IEEE Computer Society, 2007, pp. 311–325. [Online]. Available: <https://doi.org/10.1109/CSF.2007.17>
- [19] D. D. Chen, M. Woo, D. Brumley, and M. Egele, “Towards automated dynamic analysis for linux-based embedded firmware,” in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: <https://www.ndss-symposium.org/wp-content/uploads/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf>
- [20] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, “Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing,” in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: [http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018\\_01A-1\\_Chen\\_paper.pdf](http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_01A-1_Chen_paper.pdf)
- [21] J. Chen, C. Zuo, W. Diao, S. Dong, Q. Zhao, M. Sun, Z. Lin, Y. Zhang, and K. Zhang, “Your iots are (not) mine: On the remote binding between iot devices and users,” in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 2019, pp. 222–233. [Online]. Available: <https://doi.org/10.1109/DSN.2019.00034>
- [22] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, “A large-scale analysis of the security of embedded firmwares,” in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 95–110. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin>
- [23] A. Costin, A. Zarras, and A. Francillon, “Automated dynamic firmware analysis at scale: A case study on embedded web interfaces,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi’an, China, May 30 - June 3, 2016*, X. Chen, X. Wang, and X. Huang, Eds. ACM, 2016, pp. 437–448. [Online]. Available: <https://doi.org/10.1145/2897845.2897900>
- [24] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Karonte: Detecting insecure multi-binary interactions in embedded firmware,” in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 431–448.
- [25] 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, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: <https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware>
- [26] X. Wang, Y. Sun, S. Nanda, and X. Wang, “Looking from the mirror: Evaluating iot device security through mobile companion apps,” in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 1151–1167. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity19/presentation/wang-xueqiang>