KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware

Nilo Redini*, Aravind Machiry*, Ruoyu Wang†, Chad Spensky*, Andrea Continella*,
Yan Shoshitaishvili†, Christopher Kruegel*, and Giovanni Vigna*
*UC Santa Barbara †Arizona State University
{nredini, machiry, cspensky, conand, chris, vigna}@cs.ucsb.edu
{fishw, yans}@asu.edu

***Abstract*—Low-power, single-purpose embedded devices (e.g., routers and IoT devices) have become ubiquitous. While they automate and simplify many aspects of users' lives, recent large-scale attacks have shown that their sheer number poses a severe threat to the Internet infrastructure. Unfortunately, the software on these systems is hardware-dependent, and typically executes in unique, minimal environments with non-standard configurations, making security analysis particularly challenging. Many of the existing devices implement their functionality through the use of multiple binaries. This multi-binary service implementation renders current static and dynamic analysis techniques either ineffective or inefficient, as they are unable to identify and adequately model the communication between the various executables. In this paper, we present KARONTE, a static analysis approach capable of analyzing embedded-device firmware by modeling and tracking multi-binary interactions. Our approach propagates taint information between binaries to detect insecure interactions and identify vulnerabilities. We first evaluated KARONTE on 53 firmware samples from various vendors, showing that our prototype tool can successfully track and constrain multi-binary interactions. This led to the discovery of 46 *zero-day* bugs. Then, we performed a large-scale experiment on 899 different samples, showing that KARONTE scales well with firmware samples of different size and complexity.**

I. INTRODUCTION

A radical increase in the connectivity of our world is being driven by the proliferation of small, *interconnected* embedded devices, which are taking the place of traditional door locks, light bulbs, and many other previously inconspicuous objects. Unfortunately, the software (or *firmware*) running on these Internet-of-Things (IoT) devices is vulnerable to attack [3], [9], [32], which led to the development of an IoT-specific cyber-crime underground [21]. For example, in 2016, the Mirai botnet compromised millions of devices (e.g., routers and cameras) and leveraged them in denial-of-service attacks to disrupt core Internet services and shut down websites [27], [30], [50].

In response, researchers have proposed techniques to automatically identify vulnerabilities in firmware distributions, generally by unpacking them into analyzable components [11], which are then analyzed in isolation [5], [42], [40]. Nonetheless, despite these advances in vulnerability discovery techniques, state-of-the-art approaches are insufficient, and vulnerabilities persist.

A key reason behind the insufficiency of current techniques is that *embedded devices are, themselves, made up of interconnected components*. These components are different binary executables, or different modules of a large embedded OS, which *interact* to accomplish various tasks. For example, embedded devices often expose web-based interfaces comprised by a web server and various back-end applications [6], [44]. In this architecture, any given piece of functionality often relies on the execution of multiple programs [12]: e.g., the web server that accepts an HTTP request, a local binary that is summoned by the web server (e.g., using sockets), and an external command that is executed by the local binary to accomplish the request.

Each interacting firmware component (the web server, the back-end applications, and other helper programs) can make different assumptions about the data being shared, and inconsistencies can manifest as security vulnerabilities. Precisely detecting these *insecure multi-binary interactions* among the different components of a firmware sample is challenging. Program analysis approaches that consider each component in isolation, without accounting for the internal flow of data, yield suboptimal results, as they (i) ignore meaningful constraints imposed by components in the course of inter-binary communication, (ii) cannot effectively differentiate between attacker-controlled and non-attacker-controlled sources of input, and (iii) might uncover only superficial bugs.

Consider a web server that accepts user credentials, restricts their lengths to 16 characters, and then passes them to a handler binary (e.g., through environment variables), which copies them into two 16-byte long buffers. If the latter binary is *dedicated* to only handle user credentials received (and vetted) by the web server, it may forego the implementation of a length check. In this example, analyzing the handler binary *in isolation* could result in identifying bugs that are impossible to trigger in practice, and a security analysis would likely produce a large number of false positives, because it would have to assume that *all* sources of input into the binary might produce unconstrained, attacker-controlled data. These false positives would need to be checked by a human analyst, representing a time cost. As the time required by an analyst to check, patch, and test a firmware sample is not negligible, controlled interactions between binaries that, in practice, do not impose security threats should be deprioritized. On the other hand, analyses that only consider the *network*-facing binaries (i.e., those directly accepting user requests) cannot identify deeper and more complex bugs within the firmware.

Thus, an effective firmware analysis must take into account multiple binaries, and reason about the data they share.

Unfortunately, most existing work in program analysis only focuses on a single program or module at a time [55], [37], [45]. While some work has attempted to emulate embedded devices, thus analyzing all components simultaneously, current approaches either impose strict assumptions on the firmware samples [11], or achieve a limited success rate (i.e., from 13% [12] to 21% [5]). Other approaches [6], [48], [54] attempt to analyze actual devices directly, but as they adopt purely dynamic techniques (e.g., fuzzing), they may be ineffective in discovering deeper and more complex bugs [34].

In this paper, we present KARONTE, a novel static analysis approach that tracks data flows across the binaries of a firmware sample to precisely uncover security vulnerabilities. KARONTE is based on the intuition that *binaries communicate using a finite set of Inter-Process Communication (IPC) paradigms*, and it leverages commonalities in these paradigms to detect where user input is introduced into the firmware sample, and to identify interactions between the various components. The identified interactions are then used to track data flows between components, and perform cross-binary taint analysis. Finally, the propagated taints and constraints are used to detect insecure uses of the user-controlled input, which can lead to vulnerabilities.

We implemented KARONTE and evaluated it using two datasets: 53 current-version firmware samples and 899 samples gathered from related work [5]. We leveraged the former dataset to study, in depth, each phase of our approach and evaluate its effectiveness to find bugs. In our experiments, we showed that our approach successfully identifies data flows across different firmware components, correctly propagating taint information. This allowed us to discover potentially vulnerable data flows, leading to the discovery of 46 *zero-day* software bugs, and the rediscovery of another 5 *n-days* bugs, demonstrating the effectiveness of our approach on complex firmware of varying designs (i.e., both monolithic embedded OS and embedded Linux distributions). Indubitably, a sound single-binary static analysis technique could also find these vulnerabilities, but it would do so with a significant amount of false positives, making the analysis untenable in the real world. In our comparison between KARONTE's multi-binary analysis approach and the same analysis run in *single-binary* mode (i.e., with inter-binary data flow tracking disabled), the number of produced alerts increased from an average of 2 to an average of 722 per sample: KARONTE provided an **alert reduction of two orders of magnitude** and a resulting low false-positive rate. As shown in our evaluation, we estimate that the verification of all the alerts produced by a single-binary analysis might require a security analyst around four months of work. On the other hand, the verification of the alerts generated by our prototype took a cumulative time of roughly 10 hours.

Finally, we leveraged the second, bigger dataset to study the performance of our tool, showing its ability to scale well on firmware samples of different size and complexity.

In summary, we make the following contributions:

- We introduce novel combinations of static analysis techniques to perform multi-binary taint analysis. To do so, we design a novel technique to precisely apply and propagate taint information across multiple binaries.
- We propose KARONTE, a novel static analysis approach to identify insecure interactions between binaries. KARONTE radically reduces the number of false positives, making real-world firmware analysis practical.
- We implement and evaluate our prototype of KARONTE on 53 real-world firmware samples, showing that our tool can successfully propagate taint information across multiple binaries, resulting in the discovery of 46 unknown (zero-day) bugs, and producing few false positives. Then, we leverage a bigger dataset of 899 firmware samples to assess the performance of our tool.
- The results obtained by our tool were thoroughly verified by an independent researcher at another university.

In the spirit of open science, we release the implementation of our prototype and a docker image to replicate our working environment[1].

II. BACKGROUND

This section provides the background information to understand the goals of our approach and inherent challenges thereto.

A. IoT Attacker Model

IoT devices exchange data over the network. This data can come directly from the user (e.g., through a web interface), or indirectly from a trusted remote service (e.g., cloud backends). Many devices, especially routers, smart meters, and a host of low-power devices, such as smart light bulbs and locks, use the former paradigm. Moreover, recent attacks have shown that such devices can be exploited by clever remote attackers, even when their communication is restricted to a closed local network [23]. In this work, we consider network-based attackers who communicate directly with the device, either through a local network or the Internet. However, as shown in Section X, KARONTE can be easily extended to other scenarios.

B. Firmware Complexity

The firmware of modern IoT devices is complex and made of multiple components. These components can take the form of either different binaries, packaged in an embedded Linux distribution, or different modules, compiled into a large, single-binary embedded OS ("blob firmware"). The former type of firmware is, by far, the most ubiquitous: a large-scale experiment analyzed tens of thousands of firmware samples, and found that 86% of them were Linux-based [11]. Similar to other Linux-based systems, Linux-based firmware includes a large number of interdependent binaries.

The different binaries (or components) of the firmware on embedded devices share data to carry out the device's tasks. Under our attacker model, this interaction is critical, as we focus on bugs that can be triggered by attacker input from "outside" of the device (i.e., over the network), but may affect binaries other than those directly facing the network. Any

[1]https://github.com/ucsb-seclab/karonte

```c
char* parse_URI(Req* req) {
  char* p  = req[1];
  if (!strncmp(p, "<soap:AddRule", 13))
   return p; // unconstrained data
  // ...
  if (strlen(p) > 127)
   p[128] = 0;
  return p; //   constrained data
}
int serve_request(Req *req) {
  char *data = parse_URI(req);
  setenv("QUERY_STRING", data, 1);
  execve(get_handler(req));
}
```

Listing 1: Decompiled code of a network-facing program of a real firmware sample.

```c
int process_request(char *query, char *log_path) {
  char *q, arg[128];
  char log_dir[128];
  if (!(q=strchr(query, "op=")))
   return;
  strcpy(arg, q); // query string argument
  strcpy(log_dir, dirname(log_path));
  // ...
  return 0;
}
int main(int argc, char *argv[], char *envp[]) {
  char *query = getenv("QUERY_STRING");
  char *log_path = getenv("LOG_PATH");
  process_request(query, log_path);
}
```

Listing 2: Decompiled code of a handler binary that contains two bugs. However, only one bug is reachable by an attacker.

analysis that focuses only on these *network-facing* binaries would miss bugs contained in other components [6]. On the other hand, an analysis that focuses on all the binaries in isolation would produce an unacceptable amount of false alerts.

We demonstrate this in the following example service, based on a real-world firmware sample. This service is composed of a network-facing web server (Listing 1) that executes a CGI handler binary (Listing 2). When the web server receives a user request, it invokes the function `serve_request`. Then, after parsing the request (`parse_URI`), the web server executes the handler program, passing data via the `QUERY_STRING` environment variable. The handler binary retrieves the data and passes it to `process_request`. This function contains a bug: if the value of the field `op` in the user request is longer than 128 bytes, a buffer overflow occurs. This overflow is attacker-controlled and represents a significant vulnerability.

While this specific overflow would be detected by an analysis that only focuses on the handler binary, any single-binary analysis would detect *two* vulnerabilities in this program. The second one is the overflow of the `log_dir` buffer caused by the `LOG_PATH` environment variable. Though this is a legitimate bug, its classification as a vulnerability depends on the provenance of the data in `LOG_PATH`. If an attacker cannot control this data, the bug is not a vulnerability, and the real vulnerability should be prioritized. Ideally, every alert would be examined, and every bug fixed. Unfortunately, this goal is not feasible in practice. While this simple example has two alerts that reveal one vulnerability, our evaluation shows that static analysis on individual binaries in real-world firmware can produce *thousands* of alerts per device, requiring months of analyst time to process.

For static analyses to be feasible on binaries, an approach to filter out bugs that cannot be triggered by an attacker is critical. KARONTE is such an approach. It identifies data dependencies across binaries, such as the one in this example, by using static analyses to connect functions that produce (or *set*) data to functions in other binaries that consume (or *get*) it.

Throughout this paper, we refer to the program interactions shown in the above example as *multi-binary* interactions. Similarly, we refer to vulnerabilities that involve data flows across multiple binaries as *multi-binary vulnerabilities*. Finally, we refer to the binary producing data (e.g., the web server in Listing 1) as a *setter* binary, and the binary consuming data (e.g., the handler binary in Listing 2) as a *getter* binary.

C. IPC in IoT Firmware

Automatically determining how user input is introduced into and propagates through an embedded device is an open problem [36], [51], [55], and prone to a discouraging rate of false positives [22]. However, we observed that, in practice, processes communicate through a finite set of *communication paradigms*, known as Inter-Process Communication (or IPC) paradigms.

An instance of an IPC is identified through a unique *key* (which we term a *data key*) that is known by every process involved in the communication. As this information has to be available to all the involved programs before their execution, it is usually hard-coded in the binaries themselves. For example, two binaries exchanging data through a file have to know the filename (i.e., the data key) prior to transferring the data.

Data keys associated with common IPC paradigms can be used to *statically* track the flow of attacker-controlled information between binaries. Below, we describe the most common IPC paradigms employed in firmware[2].

Files. Processes can share data using files. A process writes data on a given file, and another process reads and consumes such data. The data key is the name of the file itself.

Shared Memory. Processes can share memory regions. Shared memory can be either backed by a file on the filesystem, or be anonymous (if two processes are in a parent-child relationship). In the former case, the data key is represented by the backing file name, whereas in the latter case by the virtual address of the shared memory page.[3]

Environment Variables. Processes can share data via environment variables. In this case, the data key is the environment variable name (e.g., `QUERY_STRING`).

Sockets. Processes can use sockets to share data with processes that reside on the same host (Unix domain sockets with a file path) or on a different host (network

[2] We focus on IPC mechanisms that enable rich data exchange. IPCs that do not transport data (e.g., signals) are not included, as they are out of our scope. Additionally, we reference UNIX-based concepts for user-space IPC. Other systems (e.g., iOS) have analogous concepts.

[3] Note that, components in a "blob" can use a statically mapped region to exchange data. By using the addresses of these regions as data keys, we can reason about data flows without analyzing the prohibitively large amount of control flow that separates the components themselves in a real-world firmware.

sockets). The socket's endpoint (e.g., IP address and port, or file path of a Unix domain socket) represents the data key.

Command Line Arguments. A process can spawn another process and pass data through command line arguments. The data key is the name of the invoked program.

We represent shared data as a tuple $(data\_key,\ data)$.

III. Approach Overview

KARONTE is an approach that performs *inter-binary* data-flow tracking to automatically detect insecure interactions among binaries of a firmware sample, ultimately discovering security vulnerabilities. Although our system focuses on detecting memory-corruption and DoS vulnerabilities, it can be easily extended, as discussed in Section IX. KARONTE analyzes firmware samples through the following five steps (Figure 1):

Firmware Pre-processing. KARONTE's input is comprised of a firmware sample (i.e., the entire firmware image). As a first step, KARONTE unpacks the firmware image using the off-the-shelf firmware unpacking utility *binwalk* [20].

Border Binaries Discovery. The Border Binaries Discovery module analyzes the unpacked firmware sample, and automatically retrieves the set of binaries that export the device functionality to the outside world. These *border* binaries incorporate the logic necessary to accept user requests received from external sources (e.g., the network). As such, they represent the point where attacker-controlled data is introduced within the firmware itself. For each border binary, this module identifies the program points that reference attacker-controlled data (Section IV).

Binary Dependency Graph (BDG) Recovery. Given a set of border binaries, KARONTE builds a *Binary Dependency Graph* (*BDG*), which is a directed graph [49] that models communications among those binaries processing attacker-controlled data. The BDG is iteratively recovered by leveraging a collection of *Communication Paradigm Finder* (*CPF*) modules, which are able to reason about the different inter-process communication paradigms (Section V).

Multi-binary Data-flow Analysis. Given a binary b in the BDG, we leverage our static taint engine (see Section VI) to track how the data is propagated through the binary and collect the constraints that are applied to such data. We then propagate the data with its constraints to the other binaries in the BDG that have inbound edges from b (Section VII).

Insecure Interactions Detection. Finally, KARONTE identifies security issues caused by insecure attacker-controlled data flows, which are reported for further inspection (Section VIII).

KARONTE's novelty lies in the creation of its Binary Dependency Graph and its ability to accurately propagate taint information *across* binary boundaries, enabling the detection of complex, multi-binary vulnerabilities in an efficient manner, and drastically decreasing the number of false positives that would be otherwise generated. While KARONTE focuses on inter-binary software bugs, it also performs single-binary analysis.

Furthermore, though KARONTE detects data-flows across binaries of a firmware sample, its generic design allows KARONTE to also reason about interactions of different modules of a monolithic embedded OS, as long as a separation among these modules is present (e.g., they represent different processes at runtime), as shown in Section X. Finally, given our attacker model (Section II-A), we assume that border binaries are represented by network-facing binaries (i.e., binaries implementing network services). For this reason, we interchangeably use the terms border binaries and network-facing binaries.

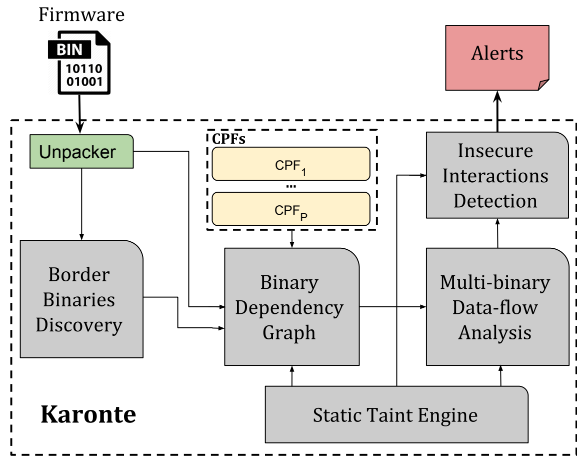


Fig. 1: After unpacking a firmware sample, KARONTE extracts the binaries handling user requests, identifies their data dependencies to build the Binary Dependency Graph (BDG), and uses its inter-binary taint analysis engine to find insecure data flows.

IV. Border Binaries Discovery

KARONTE is designed to detect vulnerabilities that may be exploited by attackers over the network. To do so, KARONTE first identifies the set of binaries that export network services (i.e., network-facing binaries) in a firmware sample. We leverage the observation that network-facing binaries are the components of a firmware sample that *receive* and *parse* user-provided data. Therefore, we identify those binaries within a firmware sample that parse data read from a network socket.

Following Cojocar et al. [8] work, we utilize three features to identify functions in embedded systems that implement parsers: (i) the number of basic blocks ($\#bb$), (ii) the number of branches (e.g., if-then-else, loops) ($\#br$), and (iii) the number of conditional statements used in conjunction with memory comparisons ($\#cmp$). Since we want to specifically identify input-affected network parsers, we consider two additional features: (iv) a metric we call *network mark* ($\#net$), and (v) a flag we call *connection mark* ($\#conn$).

The network mark feature encodes the probability that a parsing function handles network messages, and it is calculated by identifying every memory comparison in the code of the function, and comparing the referenced memory locations against a preset list of *network-encoding* strings (e.g., `soap` or `HTTP`). We initialize $\#net$ to 0 and increment it for every comparison against network-encoding strings present in the code.

The connection mark flag, instead, indicates if any data read from a network socket is used in a memory comparison. We initialize $\#conn$ to 0 and set it to 1 if there exists a data-flow between a socket read and a memory comparison operation.

We combine the aforementioned five features to compute the *parsing score* ps_b of a binary b as follows:

$$ps_b = max(\{ps_j \mid \forall j \in get\_functions(b)\}),$$
$$ps_j = (\sum_{i \in \{bb,br,cmp\}} k_i * \#i_j) * (1 + k_n * \#net_j) * (1 + k_c * \#conn_j) \quad (1)$$

where each constant k_i is set to maximize the parsing detection capabilities ($k_{bb} = 0.5$, $k_{br} = 0.4$, $k_{cmp} = 0.7$ [8]), whereas k_n and k_c promote functions that refer to network-encoding keywords and binaries that parse network data, respectively. The optimal values for the last two constants are found empirically in Section X-B. Finally, ps_j is the parsing score of the *j*-th function of b. Note that, we introduce our two features as multipliers in order to highlight input-affected network parsers.

Since all binaries are likely to have a score greater than zero, we need to distinguish and separate the "most significant" scores. To this end, we leverage the DBSCAN density-based clustering algorithm [15], which groups binaries whose scores are closely packed together. Then, we select the cluster that contains the binary having the highest parsing score in the firmware sample, and consider all the binaries belonging to the cluster as the initial set of network-facing binaries.

Finally, the algorithm implemented by this module returns the unpacked firmware sample, the set of identified network-facing binaries, and the program locations containing memory comparisons against network-encoding keywords. These memory comparisons represent the program locations where attacker-controlled data is *more likely* to be referenced.

V. Binary Dependency Graph

The Binary Dependency Graph module detects data dependencies among a set of binaries or components belonging to a firmware sample. Furthermore, it establishes how data is propagated from a setter binary to a getter binary. Data propagation across different processes differs from data transfer during subroutine calls/returns and program-library dependency analyses, as both of these are guided by control flow information. For inter-process interactions, there is no control flow transfer to rely on, because after making the data available (e.g., through environment variables), processes proceed with their execution. Since processes do not normally access other processes' memory regions, traditional points-to analyses are also futile.

Karonte tackles these problems by modeling the various inter-process communication paradigms through the use of a set of modules that we call *Communication Paradigm Finder*s (or *CPF*es). Karonte uses them to build a graph, called *Binary Dependency Graph* (or *BDG*), which encodes the data flow information among binaries within a firmware sample.

A. Communication Paradigm Finders

A CPF provides the necessary logic to detect and describe instances of a communication paradigm (e.g., socket-based communication) used by a binary to share data. To achieve this goal, a CPF considers a binary and a program path (i.e., a sequence of basic blocks), and checks whether the path contains the necessary code to share data through the communication paradigm that the CPF represents. If so, it gathers the details of the communication paradigm through the following paradigm-specific functionality:

Data Key Recovery. The CPF recovers data keys that reference data being set or retrieved by the binary under the associated communication paradigm.

Flow Direction Determination. The CPF identifies all the program points where data represented by the collected data keys is accessed. If such program points exist, it determines the *role* of each program point in the communication flow (i.e., setter or getter).

Binary Set Magnification. The CPF identifies other binaries in the firmware sample that refer to any of the data keys previously identified. These binaries are likely to share data with the binary currently under consideration, and are thus scheduled for further analysis.

We then combine the information gathered by the different CPFes to create edges in the Binary Dependency Graph, recovering the data flow across different binaries.

The specifics of each CPF depend on the OS that the firmware sample runs on (e.g., Linux). Therefore, to maintain OS-independence and to reason about inter-process communication paradigms when some information is missing (e.g., a firmware blob), Karonte uses a generic OS-independent CPF, which we call the *Semantic CPF*. This CPF leverages the intuition that any communication among processes must rely on data keys, which are often hard-coded in binaries (e.g., hard-coded addresses). To this end, the Semantic CPF detects if a hard-coded value is used to *index* a memory location to access some data of interest (e.g., attacker-controlled data). Our prototype of Karonte implements the Environment, File, Socket and Semantic CPFes (details in Appendix A).

B. Building the BDG

Karonte models data dependencies among binaries through a disconnected cyclic digraph [49], called the *Binary Dependency Graph* (or *BDG*). A BDG, G, of the set of binaries B is denoted as $G = (B,E)$, where, E is the set of directed edges. Each directed edge $e \in E$ from $b_1 \in B$ to $b_2 \in B$ is represented by a triplet $e = ([b_1,loc_1,cp_1],[b_2,loc_2,cp_2],k)$, which indicates that the information associated with the data key k (e.g., an environment variable name) can flow from binary b_1 at location loc_1 (e.g., a program point containing a call to the `setenv` function) via the communication paradigm cp_1 (e.g., the OS environment), to the binary b_2 at location loc_2 (e.g., a call to the `getenv` function) via the communication paradigm cp_2.

The algorithm to recover the Binary Dependency Graph (Algorithm 1) begins by considering the information gathered by the Border Binaries Discovery module: (i) the unpacked firmware sample in analysis (fw), (ii) the border binaries (B), and (iii) a set of program locations (int_locs) performing memory comparisons. Then, for each binary b in B, we consider each location loc in int_locs belonging to b (function get_locs), and we leverage our taint analysis engine

Algorithm 1 Binary Dependency Graph Algorithm

```
function BDG(int_locs, B, fw)
    comm_info ← {}
    E ← {}
    for each b ∈ B do
        locs ← get_locs(int_locs,b)
        for each loc ∈ locs do
            f_addr ← get_faddr(loc)
            for each block ∈ explore_paths(f_addr) do
                if (address(block) == loc) then
                    buf ← get_buf(loc)
                    apply_taint(buf)
                end if
                if matches_CPF(block) then
                    CPF_p = get_CPF(block)
                    k ← find_data_key_and_role(block,CPF_p)
                    B_new, int_locs_new ← get_new_binaries(fw,k,CPF_p)
                    update_binaries(B,int_locs,B_new,int_locs_new)
                    comm_info ← comm_info ∪ {b,block,CPF_p,k}
                end if
            end for
        end for
    end for
    for each {b,block,CPF_p,k} ∈ comm_info do
        if is_setter(block,k) then
            getters ← get_getters(comm_info,k,CPF_p)
            E ← E ∪ create_edges(b,getters)
        end if
    end for
    return (B,E)
end function
```

(Section VI) to bootstrap a symbolic path exploration starting from the beginning of the function containing loc (function $explore\_paths$). When the analysis reaches loc, we taint the memory location buf being referenced, i.e., the memory location being compared against the network-encoding keyword (functions get_buf and $apply\_taint$).

In each step of the path exploration (i.e., for each visited basic block), we invoke each of our CPF modules, which analyze the current path and use the taint information (propagated by the taint engine during the path exploration) to detect if the binary b is sharing some tainted data d. If a CPF_p *matches*, i.e., it detects that the analyzed binary relies on the communication paradigm p to share some data, we leverage CPF_p to recover all of the *details* of the communication paradigm instance in use. More precisely, the CPF_p recovers the data key k used to share data through p and infer the role (i.e., setter or getter) of the binary for k (function $find\_data\_key\_and\_role$) and finds other binaries within the firmware sample that might communicate through this channel (function $get\_new\_binaries$). Newly discovered binaries are then added to the overall set of binaries to analyze. Note that, when any of these new binaries B_{new} is scheduled to be analyzed, the analysis has to know where to apply the taint initially. In other words, we have to detect where the shared data is initially *introduced* in these new binaries. Therefore, for each newly added binary b_a, the CPF_p also retrieves the program points int_locs_{new} where the data key k is referenced, and add them to int_locs. These last two operations are performed by the function $update\_binaries$. Finally, for each analyzed binary b, we consider each CPF (*cp*) that matched for b over some key k, and use *cp* to retrieve the role of b for k (e.g., setter). Then, we create an edge between b and any other binaries that have the opposite role of b for k (e.g., getter).

To demonstrate the BDG algorithm, we again refer to Listing 1. The BDG algorithm starts by considering the memory comparison against a network-encoding keyword (Line 3). After inferring that the variable `p` is used in the memory comparison, we taint the memory location it points to, and bootstrap the intra-procedural taint analysis exploration, starting from the function `parse_URI` (Line 1), and propagating the taint by following the control flow of the program. When the taint exploration reaches the `execve` function call (Line 13), the Environment CPF detects that another binary is being executed, and that the `setenv` function is used to set the data key `QUERY_STRING`. Therefore, the Environment CPF establishes that the binary in analysis is a setter for `QUERY_STRING`. Then, the Environment CPF scans the firmware sample and finds other binaries relying upon the same data key, and adds them to the set of binaries to analyze. Finally, for each newly added binary, the Environment CPF retrieves the code locations where the data key `QUERY_STRING` is referenced (e.g., a call to the function `getenv("QUERY_STRING")`).

VI. Static Taint Analysis

KARONTE uses taint propagation to detect multi-binary vulnerabilities. This section describes the operation of the underlying taint engine, and the next section discusses how KARONTE combines the taint engine with the BDG, described previously, to achieve such detection.

KARONTE's taint engine is based on BootStomp [40]. Given a source of taint s (e.g., a function returning untrusted data) and a program point p, our taint engine performs a symbolic path exploration starting from p, and, every time s is encountered, the taint engine assigns a new taint ID (or tag) to the memory location receiving data from s. KARONTE's taint engine propagates taint information following the program data flow, and it untaints a memory location (i.e., by removing its taint tag) when the memory location gets overwritten by untainted data, or when its possible values are constrained (e.g., due to semantically equivalent `strlen` and `memcmp` functions). Our taint engine presents two improvements compared to related work: (i) it includes a path prioritization strategy, and (ii) it introduces the concept of *taint tag dependencies*.

The path prioritization strategy tackles the undertaint problem, which affects taint engines based on path exploration when dealing with implicit control flows [18], by prioritizing more *interesting* paths. In the scope of a taint analysis, a path p_1 is considered to be more interesting than a path p_2 if a variable of interest is tainted in p_1, and untainted in p_2.

Consider the example in Listing 3, and assume that the variable `user_input` (Line 14) points to tainted data. When the function `parse` is invoked, the variable `start` (Line 1) aliases `user_input` (i.e., they point to the same memory location), and, therefore, it points to tainted data. The function `parse` contains, potentially, an infinite number of paths: If the variable `start` is represented by an unconstrained symbolic expression, there is always a possible path passing through the `default` statement (Line 9) to the head of the while loop (Line 3). Among these paths, only those passing through the first case statement (Line 5) would propagate

```
1  char* parse(char *start) {
2    char* end = start + strlen(start) - 1;
3    while ( start < end )
4      switch ( *start[0] ) {
5        case '=':
6          return start + 1;
7        case ';':
8          return 0;
9        default:
10         start ++;
11     }
12 }
13 void foo() {
14   char dst[512], *user_input = get_user_input();
15   char *cmd = parse(user_input); // undertaint
16   int n = strlen(cmd);
17   if (n >= 512)
18     return -1;
19   strcpy(dst, cmd);
20 }
```

Listing 3: Path prioritization and taint dependencies use case.

the taint outside the function. Therefore, an analysis that does not explore these paths would mistakenly establish that `user_input` cannot affect the variable `cmd` (Line 15).

Our path prioritization strategy aims to valorize those paths within a function that *potentially* propagate the taint also outside the function (as the paths passing through the first `case` statement in Listing 3). As expected, we noticed that network-facing binaries contain various sanitization functions that can cause the issue just discussed. In Appendix A, we describe the implementation details of our path prioritization feature.

Finally, in our taint engine, an analyst can create dependencies among tainted variables having different tags (*taint tag dependencies*). Tracking these dependencies plays an important role in having an effective untaint policy in a multi-tag taint tracking system, thus alleviating the overtainting problem [41].

To demonstrate this, consider again the example in Listing 3, and assume that there exists an untaint policy to remove a taint tag when a variable is explicitly constrained within a range of values. First, as `get_user_input` generates untrusted data (Line 14), a new taint tag t_1 is created and assigned to `user_input`. If the function `strlen` is not analyzed (e.g., its code is not available or the call is not followed to keep the overall analysis tractable), following the semantics of a multi-tag taint tracking [40], the variable `n` gets tainted using a different tag t_2. When the taint execution engine reaches the `if` statement (Line 17), following the untaint policy in use, the variable n is automatically untainted by removing the tag t_2. Given that the taint tag of `user_input` (t_1) is different than `n`'s tag (t_2), `user_input` is not untainted, and the call to the unsafe `strcpy` (Line 19) could cause a false positive to be generated. This behavior emerges because some functions that semantically constrains tainted data might not be analyzed (due to lack of code, or limits of the employed analysis). The solution we propose is to maintain the information that the taint tag of `user_input` (i.e., t_1) *depends* on the taint tag of `n` (i.e., t_2), and, to untaint `user_input` when `n` is untainted. We say that a taint tag t_1 *depends* on a taint tag t_2, if removing t_2 (i.e., untainting the variable with taint tag t_2) provokes t_1 to be removed. Of course, the taint tag $t1$ might depend on multiple taint tags. In this case, if all the tags that t_1 depends on are removed, t_1 is removed too. Our prototype automatically finds semantically equivalent `memcmp` and `strlen` functions, and applies taint tag dependencies (see Appendix A).

VII. Multi-binary Data-flow Analysis

To discover insecure interactions among binaries and find vulnerabilities, we need to recover the data-flow details of the binaries in a BDG. Enumerating all the possible *inter-binary* paths in a BDG leads, in general, to the path explosion problem [4].

Our key insight is that the inter-binary paths *more likely* to lead to bugs are those that apply *less strict* constraints on the user-provided data d (i.e., the set of values that d can assume has a higher cardinality). To retrieve such paths, we collect the sets of constraints that a binary applies to d across different program paths, and propagate to other binaries only the least restrictive set of constraints.

To do so, we create a graph that we called the *Binary Flow Graph* (or *BFG*), which extends the BDG with the least strict set of constraints applied to the data shared among multiple binaries. In the BFG, an edge ([b_1, loc_1, cp_1, c_1], [b_2, loc_2, cp_2, c_2], k) indicates that the data associated with the data key k can flow from the binary b_1 at location loc_1 via the communication paradigm cp_1 with the set of constraints c_1 to the binary b_2 at location loc_2 via the communication paradigm cp_2 with the set of constraints c_2. The BFG building algorithm is based on the notion of chaotic iteration [1], and is composed of two phases.

Initialization. We consider every edge in the BDG and create a new edge setting $c_1 = c_2 = \perp$ ($\perp$ means "uninitialized"). Next, we consider every edge e whose setter (i.e., b_1) is a border binary, and retrieve the variable var_1 that contains the data being shared at location loc_1. Then, we use our taint engine to explore the paths between the entry point of the function containing loc_1 and loc_1 itself, and collect, for each path, the set of constraints applied to var_1. For instance, if var_1 maximum length is checked (e.g., through a `strlen`) against a constant value, we collect such constraint. Then, we select the least strict set of constraints l_1, and set $c_1 = l_1$. Finally, we add e to a set $wset$, which is used during the second phase.

Constraint Propagation. We consider every edge $e_w \in wset$, and set $c_2 = c_1$, thus propagating the constraints from the setter binary to the getter binary. We then retrieve the variable var_2 used by b_2 to receive the data at loc_2 and find the least restrictive set of constraints l_2 that the binary applies to var_2 (relying on the same approach used to find l_1), and set $c_2 = c_2 \cup l_2$.

As b_2 might further share the data, we also determine the additional constraints that b_2 applies to such data before re-sharing it. To do this, we collect every edge e_r where the binary b_2 is the setter. Then, we run our taint engine to find a path between the program point where the binary previously received the data (i.e., loc_2 of edge e_w) and the location where it shares it further (i.e., loc_1 of edge e_r) and find the least strict set of constraints l_r applied to var_2 along these paths. If we cannot find a path between these two program points (e.g., due to limits of the underlying analyses), we determine l_r using the same approach used to find l_1 (i.e., starting from the entry point of the function containing loc_1 of e_r). Finally, we consider the constraints $c^* = l_r \cup c_2$ and the constraints for the setter of e_r. If the latter set is uninitialized (i.e., $c_1 = \perp$ for e_r) or more restrictive than c^*, we substitute it with c^* and add e_r to $wset$—thus keeping the least restrictive constraints. We iterate this phase until $wset$ is empty.

VIII. Insecure Interactions Detection

The Insecure Interactions Detection module leverages the BFG to find dangerous data flows and detect subsets of two classes of vulnerabilities: (i) memory-corruption bugs (e.g., buffer overflows) and (ii) denial of service (DoS) vulnerabilities (e.g., attacker-controlled loops). To detect the former class, we first find *memcpy-like* functions within a binary, that is, every function that is semantically equivalent to a `memcpy` (Appendix A). Then, if attacker-controlled data unsafely reaches a memcpy-like function (e.g., without being sanitized), we raise an alert. To detect the latter class of vulnerabilities, we retrieve the conditions that control (guard) the iterations of a loop. Then, we check whether their truthfulness completely depends on attacker-controlled data, and, if so, we raise an alert. We refer to both `memcpy-like` functions and attacker-controlled loops with the general term *sinks*.

The Insecure Interactions Detection phase works as follows. First, we consider every edge e_f in a BFG, and for each node $(b,loc,cp,c) \in e_f$, we leverage the static taint engine to bootstrap a symbolic path exploration from the function f containing loc. Then, when we encounter the location loc, we rely on the provided CPF cp to retrieve the address of the buffer buf that references attacker-controlled data at location loc (e.g., the memory location returned by `getenv`), and apply the taint to it. Furthermore, at each step of the path exploration, we collect any constraints on buf (in a similar way as explained in Section VII) and add them to c.

If a sink is encountered during the path exploration, we check whether it contains tainted data. If the sink is a loop, and one of its conditions completely relies on tainted variables, we raise an alert (for a possible DoS vulnerability). On the other hand, if the sink is a memcpy-like function, we retrieve the address of the destination buffer $bdst$. Then, we retrieve the allocation point of $bdst$ (e.g., its position in the function's stack) and estimate its boundaries (e.g., the offset of the surrounding variables in the stack) to recover its size. If the size of buf (given by its constraints c) is greater than the size of $bdst$, we raise an alert, as it means that the copy operation might produce a buffer overflow.

Finally, we consider every disconnected node in the BFG, and perform a single-binary static analysis.

IX. Discussion

In this section, we discuss some key points of our system.

As with any other path-based exploration analyses, Karonte suffers from the path explosion problem. In our prototype, we limit path explosion, while increasing precision, by: (i) providing precise taint propagation policies (e.g., function calls with no tainted arguments are not always followed, depending on call-stack depth), (ii) using timeouts (each symbolic path exploration is performed up to a certain time limit), (iii) limiting loop iterations, and (iv) automatically creating function summaries (as explained in Appendix A).

Our prototype may generate both false positives and false negatives. They are due to the fact that taint information might not be correctly propagated to unfollowed paths (e.g., due to time, call-stack depth, or loop constraints), or imprecisions of the underlying static analysis tool (i.e., angr), as shown in Section X. This might result in incomplete BDGs, and, therefore, some security vulnerabilities might be left undiscovered. However, Karonte alleviates this problem by generating taint tag dependencies (see Section VI).

Though by default, Karonte finds buffer overflows and denial-of-service vulnerabilities, its design allows an analyst to support different types of vulnerabilities. The Insecure Interactions Detection algorithm (Section VIII) relies on a set of detection modules designed to use taint information to recognize specific classes of vulnerability. For instance, an analyst can extend our system to find use-after-free bugs by providing a new detection module, such as [16].

X. Evaluation

In this section, we first evaluate each phase of Karonte's algorithm on several of the latest firmware samples available at the time of writing. Then, we evaluate Karonte's performance using a dataset from related work [5]. We implemented a prototype of Karonte on top of angr [43], and, in particular, our taint engine on top of BootStomp [40].

A. Datasets

We evaluated our prototype of Karonte on both Linux-based firmware samples and firmware blobs.

Recent Linux-based Firmware. We selected four major IoT vendors that make the firmware of their devices available for download: NETGEAR, TP-Link, D-Link, and Tenda. Then, we scraped their official websites to collect the available firmware, for a total of 112 different products. Unfortunately, several firmware samples were not available for download or packaged with proprietary algorithms. We eventually successfully collected 49 different firmware samples.

Firmware Blobs. We retrieved the BootStomp [40] dataset, which provides us with the ground truth for our approach. BootStomp's dataset is composed of 5 firmware samples. In particular, it contains two versions of Qualcomm's Little Kernel (or LK): the most recent at the time of publication, and a version (not specified) that was released before 2016-07-05 that contains a known vulnerability. Throughout this work, we refer to the latter with a *. Also, as these firmware blobs receive data from persistent storage (rather than from the network), we modified our Border Binaries Discovery module to accommodate BootStomp's approach to identifying procedures that read from or write to the hard drive. Finally, we did not consider the Mediatek bootloader because angr fails to analyze it [40].

Table I shows our dataset of 53 firmware images (the combination of the Linux-based and firmware blobs datasets).

TABLE I: Results on our dataset of current-version firmware samples. For each vendor we report the device series, the number of firmware samples, and those samples whose network services are handled by one and multiple binaries, respectively, the total number of binaries, the average number of border binaries, the number of alerts our prototype generated, the average execution time, the number of true positives, and the number of bugs retrieved by tracking the data-flow through one or more binaries.

Vendor	Device Series	# Firmware Samples	# Single Binary	# Multi Binaries	# Binaries	Avg # Border Binaries	#Alerts	Avg Time [hh:mm:ss]	# Bugs	# Single Binary Vulnerabilities	# Multi-binary Vulnerabilities
NETGEAR	R/XR/WNR	17	12	5	4,773	7	36	17:13:45	23	10	13
D-Link	DIR/DWR/DCS	9	4	5	1,290	5	24	14:09:12	15	0	15
TP-Link	TD/WA/WR/TX/KC	16	16	0	1,769	5	2	1:30:16	2	2	0
Tenda	AC/WH/FH	7	4	3	734	5	12	1:01:22	6	0	6
Huawei	ALE-L23	1	1	0	1	0	6	4:04:37	4	4	0
Nvidia	Nexus 9	1	1	0	1	0	0	0:25:01	0	0	0
Qualcomm	-	1	0	1†	1	0	0	2:28:27	0	0	0
Qualcomm*	-	1	0	1†	1	0	7	5:03:32	1	1	0
Total	-	**53**	**38**	**15**	**8,565**	**279**	**87**	**49:09**	**51**	**17**	**34**

†: The firmware sample was manually separated into distinct components.

Large-scale Dataset. To measure the scalability of KARONTE, we obtained Firmadyne's dataset [5], and considered the firmware samples whose architecture is supported by BootStomp [40] (i.e., ARM, AARCH64, and PowerPC). We did not consider firmware samples for MIPS architectures, as angr only partially supports MIPS binaries, and some of its analyses might yield imprecise results in these cases (as explained in Section X-C). This limitation is introduced by the employed tool, and not by our approach, which is architecture-independent. Overall, this dataset consists of 899 firmware samples from 21 different vendors (Table III).

B. Border Binaries Discovery

First, we established the optimal values for k_n and k_c. We randomly selected one firmware sample and manually investigated its border binaries. We identified three binaries. Then, we ran the Binary Border Discovery module against the firmware sample using different values for k_n and k_c (ranging from 1 and 10). For $k_n \geq 5$ and $k_c \geq 1$ we correctly identified the three binaries as border binaries. Therefore, we set k_n and k_c to 5 and 1 respectively.

Next, we measured the effectiveness of the Border Binaries Discovery module to identify network parsers. We randomly picked 10 firmware samples, investigated their network-facing binaries and randomly selected 150 more binaries. Then, we ran the Border Binaries Discovery module against all of these binaries three times: (i) considering only the features described in [8], (ii) considering also the $\#net$ feature, and, (iii) considering also the $\#conn$ feature. In the first case (i), this module identified 50 binaries containing parsers. However, after manual investigation, we concluded that only 16 of them handled data received from the network. In the second case (ii), our tool identified 51 binaries, and we found that 26 of them contained network parsers that are affected by user input. Finally, in the third experiment (iii), this module identified 50 binaries, and we verified that 26 of them contained network parsers affected by user input. One of the 51 binaries identified during experiment two (ii) was not detected as a network parser in experiment three (iii). We found that, indeed, it does not implement any network functionality. Finally, we found that our Border Binaries Discovery module's algorithm missed a real network parser. This false negative was due to the fact that angr failed to identify any strings, as the binary retrieved them by computing their addresses at runtime as offsets from the Global Offset Table (GOT), thus affecting the binary parsing score.

C. Binary Dependency Graph

We manually checked the soundness and completeness of the recovered BDGs. In all of the 53 cases, to the best of our knowledge, the BDGs were sound: every edge in the BDG corresponded to an existing data dependency between the involved binaries. Then, we checked if any edge was missing. Out of 53 BDGs, we found that, for the three Tenda firmware samples, the BDG algorithm failed to connect an edge between two binaries, as a valid network-facing binary was missing (as explained in Section X-B). However, our Semantic CPF correctly identified the binaries receiving data from the missing network-facing binary as getters. Furthermore, the BDG of 14 TP-Link firmware samples did not contain any edges, as angr failed to resolve several data attributes referenced within these firmware samples during the Border Binaries Discovery phase. We discovered that these firmware samples ran on a MIPS architecture, which is unfortunately poorly supported by angr.

We manually investigated all the matching CPFs, and we found that the Semantic and the Environment CPFes matched 11 and 32 times respectively, whereas the remaining CPFes did not identify any active IPC communication. After manual investigation, we concluded that these results were indeed correct.

D. Insecure Interactions Detection

Each alert produced by our prototype consists of an insecure data flow (e.g., a flow reaching an unsafe memcpy-like function), and we distinguish true positives from false positives according to the type of data reaching the sinks of the data flows. If the data is provided by the user (e.g., HTTP headers), we consider the alert a true positive bug (if the bug can be exploited, it is denoted as a security vulnerability). On the other hand, if the data is not user-provided (e.g., the data is represented by filesystem file names), we consider the alert a false positive.

Our prototype produced 87 alerts, among which 51 were true positives (34 multi-binary bugs and 17 single-binary bugs), for

TABLE II: Comparative Evaluation. Number of alerts generated for each step of KARONTE. For each vendor, we report the average values.

Vendor	ALL†			PARSERS			BDG			KARONTE		
	Ana. Bin‡	No. Alerts	Time	No. Bin	No. Alerts	Time	No. Bin	No. Alerts	Time	No. Bin	No. Alerts	Time
NETGEAR	71	729	7 days	7	312	13:53 h	8	443	25:31 h	8	2	17:13 h
D-Link	80	811	7 days	5	205	12:00 h	6	294	14:33 h	6	3	14:09 h
TP-Link	181	819	7 days	5	71	7:44 h	5	86	6:37 h	5	0	1:30 h
Tenda	41	474	7 days	5	154	10:41 h	6	175	11:07 h	6	2	1:01 h
Total	**2,424**	**20,931**	**28 days**	**279**	**9,363**	**44:18 h**	**312**	**12,778**	**48:57 h**	**312**	**74**	**33:57 h**

†: Experiment conducted up to 7 days. ‡: Average number of binaries analyzed within 7 days.

a total of 8,565 considered binaries (Table I). We manually verified each alert by reverse-engineering the involved binaries and inspecting the highlighted data flows. We reported all our findings to the appropriate manufacturers (responsible disclosure).

We also verified how many of these 51 bugs were security vulnerabilities. We acquired two of the devices and successfully crafted PoCs for three of the vulnerabilities, and obtained one CVE and one PSV[4]. Two other alerts were non-exploitable bugs: though user data reached a sensible program point, we were not able to achieve control-flow redirection. Five more vulnerabilities were confirmed by related work [40]. For the remaining vulnerabilities, we relied on manufacturers' collaboration, since we could not obtain all of the devices for the firmware in our dataset without incurring in excessive expenses, and confirmed nine more vulnerabilities. Sadly, some of the manufacturers were uncooperative and refused to consider reports without a proof-of-crash (PoC) on the physical device. Therefore, we assessed the remaining vulnerabilities by reverse engineering the firmware. By using vendor-confirmed vulnerabilities and checking whether other firmware using the same codebase (information gathered from vendors) had similar bugs, we were able to confirm another 20. The remaining 12 were statically investigated for exploitability, and we believe that all of them are exploitable. Overall, we verified every alert, and 46 of the detected bugs, to the best of our knowledge, were not publicly known before KARONTE. The 12 confirmed vulnerabilities are being fixed as well as those bugs affecting samples sharing similar codebases (at least an additional 20).

To evaluate the false negative rate of our prototype, we searched for CVEs involving our dataset, and collected information for 30 different bugs. Since 21 of these bugs belonged to the binary that angr failed to analyze (Section X-B), we manually added this binary to the BDG and annotate the functions referencing network-encoding keywords, and re-ran our analysis. KARONTE re-discovered all of these bugs. Overall, our prototype generated two false negatives belonging to the Nvidia and Huawei firmware, respectively. In these cases, we failed to introduce the initial taint, as angr failed to resolve two indirect control-flow transfers.

E. Comparative Evaluation

To evaluate the importance of every step of KARONTE, we compared the effort required by an analyst to verify the results generated by different approaches. To do this, we considered the

[4]CVE-2017-14948, PSV-2017-3121

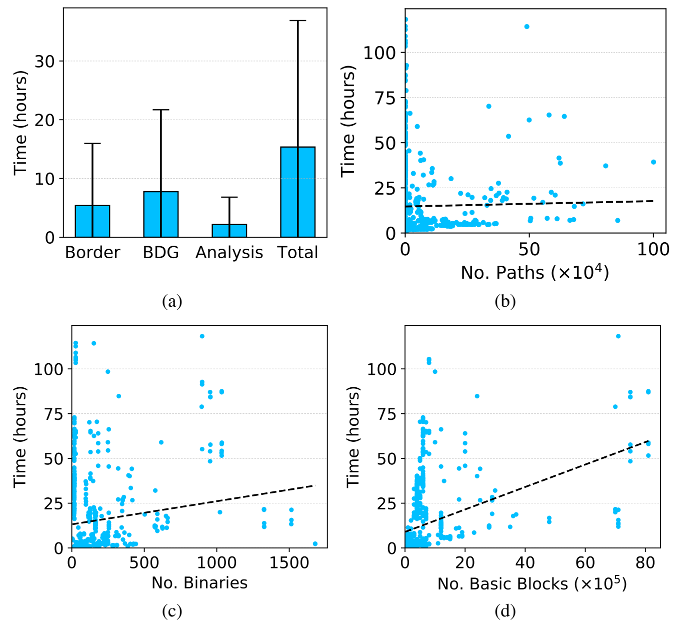
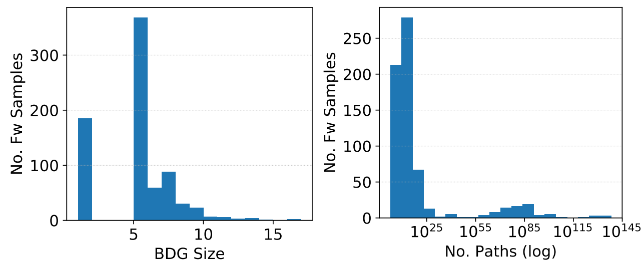


Fig. 2: (a) Average and standard deviation of the execution time of each step of KARONTE. Analysis time includes BFG Recovery and Insecure Interaction Detection. (b) Dependency between execution time and the number of explored paths. (c) Dependency between execution time and the number of binaries in the firmware samples. (d) Dependency between execution time and the number of basic blocks in the firmware samples. The dashed lines represent the linear regressions.

49 firmware samples containing multiple binaries, and selected those 29 samples whose architecture is fully supported by angr.

We then compared four different approaches. First, we performed a static single-binary bug search using our static taint engine on every binary contained in each firmware sample (dataset **ALL**). Second, we ran our static taint engine on the border binaries of the firmware sample (dataset **PARSERS**). Third, we run the BDG algorithm on each firmware sample, and we applied our static taint engine to the binaries that handle user-provided data *without* propagating the data constraints (dataset **BDG**). Finally, we considered our full approach (dataset **KARONTE**). During this evaluation, we made the realistic assumption that without propagating user input from network-facing binaries, the security analyst has no prior knowledge of where, or if, the user input is introduced in a given binary. Therefore, we considered every IPC channel as a possible source of input.

As clearly shown by our results depicted in Table II, the number of generated alerts decreased to a manageable number



(a) (b)

Fig. 3: (a) Distribution of the sizes of the BDGs of our firmware samples. (b) Distribution (in log scale) of the estimated total number of paths in an average binary in the BDG. For graphical reasons, this figure shows 95% of our data.

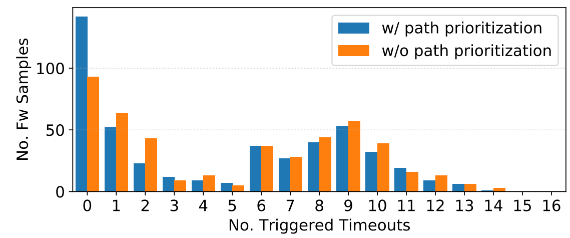


Fig. 4: Distribution of the number of timeouts triggered during the symbolic exploration with and without our path prioritization.

(i.e., 20,931 to 74) only when applying the full KARONTE approach. We manually investigated 50 randomly picked alerts selected from those generated by the single-binary analysis experiments, which were effectively filtered out by the full KARONTE approach. All of them were false positives. In fact, in all of these cases, the binaries causing the alerts were spawned (e.g., through the `system` function call) only using hard-coded arguments and parameters, thus not being affected by the user input. On average, KARONTE uncovered 2 vulnerabilities per sample not discovered when only network-facing binaries were considered (**PARSERS**), which highlights the importance of considering all the binaries handling attacker-controlled data.

Throughout our experiment, our expert program analyst averaged 7 minutes per investigation of alert. Based on this, we estimate that the investigation of alerts stemming from a single-binary analysis of a NETGEAR firmware sample, for instance, would require approximately 138 hours. KARONTE decreases this time to 14 minutes.

F. Large-scale Scalability Assessment

We assessed KARONTE's performance and scalability by analyzing 899 firmware samples from Firmadyne dataset (all samples using architectures supported by KARONTE). We ran this evaluation on a cluster of machines equipped with Intel Xeon E5 CPU, 16 to 32 GB of RAM, and running Ubuntu 18.04.

Firmware Complexity. We investigated the complexity of the firmware samples in our dataset using three metrics: number of binaries, number of basic blocks, and number of paths present in the binaries handling user input (i.e., those in the BDG). In particular, we leveraged Bang et al.'s work [2] to calculate an upper bound on the number of paths of a program. To do this, Bang's approach requires us to retrieve the program's longest path, which is an NP-hard problem [46]. To overcome this issue, we approximated the longest path of a binary by performing a symbolic exploration for 10 minutes (while limiting the maximum number of iterations of a loop to five), and recording the longest visited path.

Table III shows that, on average, a firmware sample contains around 157 binaries, for a total of $7.85 * 10^5$ basic blocks. Furthermore, 82% of the binaries in the BDGs contain less than 10^{25} paths, as shown in Figure 3.b. Interestingly, our dataset includes some far more complex firmware samples. Around 2% of them contain more than 1000 binaries (for a total of more than $7.15*10^6$ basic blocks), and those handling user input can reach a number of paths on the order of 10^{306}.

Overall, our dataset is composed of a collection of firmware samples with a wide range of complexity, thus making it suitable for studying the performance of our tool.

BDG. We investigated the BDGs of our dataset, and found that 38.7% of the firmware samples implement network-related services through the use of multiple binaries (#Multi-Binary column in Table III). Their BDGs contain, on average, 5 binaries, among which 3 are border binaries. Most of BDGs are comprised of 5 or 6 binaries, though some samples have BDGs composed of more than 10 binaries, and one BDG contains 16 binaries (Figure 3.a). For 6 vendors our tool did not identify any firmware sample sharing user data among multiple binaries. We randomly picked 5 of these 18 firmware samples for manual investigation. In three cases, the network functionality was indeed performed by single binaries, not communicating with each other. In two cases, the Border Binary Discovery phase failed to find one border binary, as we could not statically resolve its strings (Section X-B). However, the firmware samples were relying on a single program to implement the network functionality of the device.

On average, a BDG connected subgraph contains 4 nodes (i.e., four binaries communicating), and has a depth of 1 (i.e., a binary shares data with other 3 binaries). However, our dataset presented more complex cases. For instance, the BDG composed of 16 different binaries had 4 different connected subgraphs, and the biggest subgraph had a depth of 2 and contained 7 binaries. In this case, we found that a border binary exchanged data with 6 other binaries, and one of them modified the data and shared it further. Finally, there were a few cases where both the cardinality of a BDG connected subgraph and its depth were 1 (e.g., Belkin). In these cases, we found that a border binary was using IPC to exchange data with itself.

Overall, the results are in line with those discussed in Section X-C, and show that firmware samples are made of highly interconnected components, whose interactions can be fairly complex, highlighting the importance of approaches like KARONTE.

Performance. We measured the time required by each phase of KARONTE, and the total analysis time. Our prototype fully analyzed 80% of the firmware samples within a day, and, on average, it completed each phase within 8 hours (Figure 2.a).

TABLE III: Dataset for large-scale evaluation. In order: vendor's name, number of firmware samples, number of firmware samples whose network services are handled by multiple binaries (percentage), number of binaries in the firmware samples, number of border binaries, number of binaries in the BDG, cardinality of a subgraph in the BDG, maximum depth of a subgraph in the BDG, number of basic blocks in the firmware sample, number of paths in binaries handling user input, execution time, and generated alerts.

Vendor	# Firmware Samples	# Multi Binary (%)	# Binaries†	# Border Binaries†	BDG Size†	Subgraph Cadinality‡	Subgraph Depth‡	# Basic Blocks†	# Paths†	Explored Paths	Time† [hh:mm:ss]†	# Alerts†
Airlink101	1	1 (100.0%)	94	5	8	4	1	9×10^{04}	1×10^{05}	68.58K	3:55:44	13
Belkin	6	1 (16.7%)	184	5	5	1	1	2×10^{05}	3×10^{81}	4.12K	0:49:46	1
Buffalo	3	0 (0.0%)	301	5	5	0	0	2×10^{06}	3×10^{14}	43.00	0:17:01	0
Cisco	21	6 (28.6%)	142	5	5	3	1	4×10^{05}	2×10^{22}	173.27K	5:36:15	4
D-Link	306	196 (64.1%)	103	3	3	1	1	7×10^{05}	3×10^{30}	41.64K	21:51:27	1
Foscam	5	5 (100.0%)	115	5	6	4	2	4×10^{05}	5×10^{15}	52.20K	18:01:00	7
Inmarsat	2	0 (0.0%)	640	5	5	0	0	2×10^{06}	9×10^{03}	3.10K	11:05:06	0
Linksys	12	1 (8.3%)	404	5	6	11	1	8×10^{05}	2×10^{305}	23.20K	3:32:36	1
NETGEAR	304	52 (17.1%)	115	5	5	3	1	5×10^{05}	4×10^{107}	82.83K	3:54:00	1
OpenWrt	12	1 (8.3%)	14	1	1	4	2	3×10^{04}	4×10^{15}	24.41K	1:06:16	0
Polycom	7	0 (0.0%)	130	4	4	0	0	1×10^{06}	2×10^{12}	1.01M	31:49:22	8
Supermicro	26	3 (11.5%)	209	5	5	2	1	4×10^{05}	2×10^{148}	12.16K	1:54:03	5
Synology	44	28 (63.6%)	679	3	3	1	1	5×10^{06}	1×10^{14}	4.55K	33:12:01	1
TP-Link	3	0 (0.0%)	200	5	5	0	0	7×10^{05}	1×10^{12}	2.00K	2:53:15	1
TRENDnet	55	26 (47.3%)	156	3	4	2	1	6×10^{05}	2×10^{118}	14.52K	22:59:12	1
Tenda	4	1 (25.0%)	332	5	5	1	1	6×10^{05}	2×10^{13}	13.04K	5:39:25	1
Tomato	51	11 (21.6%)	223	5	5	4	1	7×10^{05}	1×10^{26}	90.36K	9:40:55	6
Ubiquiti	15	7 (46.7%)	68	3	4	1	1	1×10^{05}	3×10^{08}	11.61K	3:06:21	2
Verizon	1	0 (0.0%)	10	5	5	0	0	1×10^{05}	5×10^{20}	2.49K	0:19:02	1
Zyxel	19	9 (47.4%)	153	5	6	3	1	3×10^{05}	4×10^{16}	260.87K	4:46:38	3
forceWare	2	0 (0.0%)	173	5	5	0	0	2×10^{05}	2×10^{03}	3.00	0:30:18	0
Total	**899**	**348 (38.7%)**	**140.82K**	**3.60K**	**-**	**-**	**-**	**16.43M**	**-**	**60.68M**	**11830:28:37**	**1.03K**

†: Averages considering all of the vendor's firmware samples.
‡: Averages considering the firmware samples whose network services are handled by multiple binaries (multi-binary samples).

As we can see, the Border Binaries Discovery and BDG Recovery phases presented a great variance. We discovered that the time increase in the Border Binary Discovery phase was caused by the Z3 theorem solver, which sometimes required several minutes to solve a single symbolic expression and is heavily utilized by angr (some CFGs took 8 hours to be built). Time increases in the Binary Dependency Graph phase were also due to slow z3 solves, and, in a few cases, to an unusually high number of data keys. The time spent to build a BDG depends on the number of analyzed paths, which, in turn, depends on the number of data keys found in a binary. Some border binaries (around 7%) contain more than 50 data keys, which we analyzed to detect whether the binary is a setter or a getter. Since we perform each of these analyses up to a certain time limit (10 minutes in our experiments), the BDG phase might take several hours to analyze a single binary (around 8 hours for 50 data keys).

Figure 2.b depicts how the number of analyzed paths influences the total analysis time. Most samples that took longer to be fully analyzed are those for which we explored a small number of paths. These samples are those that caused angr to take a long time to generate the CFGs.

Finally, we found that the number of binaries and their size (in terms of the number of basic blocks) in a firmware sample do not significantly impact on the performance of our tool. In fact, 67% of the firmware samples that we analyzed for more than a day contained a number of binaries less than or equal to 27 (for a total number of basic blocks less than or equal to $7.64*10^5$), whereas far more complex firmware samples were analyzed faster, as shown in Figure 2.c and Figure 2.d.

Overall, KARONTE scales well with the firmware complexity, in terms of the number of binaries, basic blocks, and paths.

Symbolic Exploration. We studied the impact of our path prioritization strategy and untaint policies on our results. First, we ran our prototype on the KARONTE dataset with and without the path prioritization strategy and compared the number of times that a timeout (set to 10 minutes) triggered during the analysis (note that no timeout means all paths carrying tainted data have been exhausted). Figure 4 depicts the distribution of the number of timeouts triggered during the analysis of the samples in our dataset. Indeed, the number of firmware samples fully analyzed without any timeout is higher when the path prioritization is enabled. Specifically, considering the total number of times we ran our taint engine, we explored every tainted path 84% of the times when the path prioritization was enabled, and 75% of the times when it was disabled. This corresponded to around $2*10^6$ paths being pruned away. On average, KARONTE explored around $15*10^3$ paths per firmware sample (Table III). Though the average number of estimated paths is significantly higher, it is important to remind that KARONTE aims to find and analyze only those paths affected by user input.

Then, we ran our tool with and without untaint policies and compared the number of generated alerts. Overall, the number of alerts generated when the untaint policies were applied decreased by 2.5%. We manually inspected all of them and found them to be, indeed, false positives. In these cases, a buffer was safely copied using unsafe functions (e.g., using `strcpy` after checking their size through `strlen`).

Alerts & Vulnerabilities. On average, KARONTE generated 2 alerts per sample, for a total of 1,037 alerts. We sampled 100 alerts for inspection and found 44 to be true positive (i.e., user-provided data reached a sink), and 30 of them to be multi-binary vulnerabilities. This means that, in almost one case out of two, KARONTE is able to detect critical data flows that require immediate attention, and that often involve multiple binaries. We reported our findings to the respective vendors.

Firmadyne raised *zero* alerts for the large-scale dataset. Though we cannot be certain about *why* Firmadyne did not find bugs, we speculate that this emphasizes one of the advantages of a static approach over a dynamic one: though KARONTE makes certain trade-offs, it analyzes complex firmware without emulating it or tackling the dynamic coverage problem.

G. Verifiability

To promote reproducible research, we asked an independent researcher from Northeastern University to replicate our results shown in Table I (excluding the columns bugs and vulnerabilities, as they would have needed to contact the manufacturers, but including generated alerts). The large-scale evaluation and Table II were not replicated, due to the prohibitive cost of the required computational power.

We created a Docker container with our tool and running environment (e.g., KARONTE's dataset). Along with this container, we provided the researcher with the source code of our tool, a copy of this paper, the necessary documentation explaining the purpose of each component in our tool, and our expected results. Finally, we instructed them on how to run our tool. The independent researcher was successfully able to obtain all of the results presented in Table I.

XI. Related Work

Dynamic Taint Tracking & Emulation. Dynamic taint analysis [41] (DTA) is a well-known technique for vulnerability detection. However, reduction in performance is one of the main reasons for not integrating DTA into production devices. Techniques based on function summaries [55], instruction coalescing [37], storage optimization [24], and multi-threading [33] were developed to improve the performance of DTA techniques. However, resource constraints on embedded devices render traditional DTA techniques infeasible [53]. Although techniques such as FirmaDyne [11], SURROGATES [26], and Avatar [52] address this by emulation, custom hardware, and hardware proxying, they either pose strict assumption on the firmware, or rely on the presence of debugging ports (e.g., JTAG), which are usually disabled.

Fuzzing. Driller [45] uses bounded symbolic execution to generate deep inputs. Dowser [19] and offset-aware fuzzing [39] use a combination of taint analysis and symbolic execution to generate overflow-inducing inputs. However, gray-box fuzzing techniques [25], [31], [38] require access to the runtime state of the target program making them unsuitable for embedded devices. DIFUZE [10] uses the interface information extracted using static analysis for fuzzing mobile kernel drivers. However, their techniques are customized to kernel drivers and are not applicable to binary programs. RPFuzzer [48] provides a fuzzing framework for routers. However, it requires monitoring of the running process, which is not always possible for proprietary routers. IoTFuzzer [6] performs black-box fuzz testing of various IoT devices through the corresponding mobile app. However, it obeys to the app's code constraints on the user input to generate fuzzing inputs (user's data sanitization). FIRM-AFL [54] and FirmFuzz [44] fuzz programs on IoT devices by emulating the corresponding firmware. However, a faithful emulation of firmware is a hard problem. Furthermore, similar to the other fuzzing techniques they suffer from effective input generation.

Static Analysis. Most of the static analysis-based techniques focus on specific vulnerability types, such as buffer overflows [28], [35], integer overflows [47], [7], use-after-free [17], authentication bypass [42] and v-table escapes [14]. Few techniques exist to detect general taint style vulnerabilities [13], [40]. However, they suffer from scalability. Unlike KARONTE, none of these techniques handle vulnerabilities that require modelling interaction between multiple binaries. Costin et al. [12] provide a framework that mixes static analysis and emulation to analyze embedded web interfaces. However, their technique is not generic, does not detect previously-unknown memory-corruption vulnerabilities, and relies on various heuristics for emulation.

XII. Conclusion

We presented KARONTE, an approach to detect insecure interactions among components of embedded firmware. KARONTE leverages novel static analysis techniques to drastically reduce the false positives that traditional binary analysis techniques produce when analyzing real-world firmware. We extensively evaluated KARONTE on the latest firmware of 53 IoT products, showing its effectiveness. Our prototype produced 87 alerts (two orders of magnitude reduction over an approach *not* considering inter-component interactions), among which we identified 46 previously unknown *zero-day* bugs. Finally, we showed that KARONTE scales well using a collection of 899 firmware samples of different size and complexity.

Acknowledgements

We would like to thank our reviewers for their valuable comments and inputs to improve our paper. We also thank Ph.D. Sajjad Arshad and Prof. Engin Kirda to help us validate our findings, and Prof. Manuel Egele for sharing the large-scale dataset.

This material is based upon work supported by AFRL under Award No. FA8750-19-C-0003, by ONR under Award No. N00014-17-1-2011, and by NAVSEA under Award No. N00024-12-C-6404/0451. Research was also sponsored by DARPA under agreement number HR001118C0060. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the U.S. Government, or the other sponsors.

References

[1] K. R. Apt, "The essence of constraint propagation," *Theoretical Computer Science*, vol. 221, no. 1, 1999.

[2] L. Bang, A. Aydin, and T. Bultan, "Automatically Computing Path Complexity of Programs," in *Proc. of the Joint Meeting on Foundations of Software Engineering*, 2015.

[3] C. Brook, "Travel Routers, NAS Devices Among Easily Hacked IoT Devices," https://threatpost.com/travel-routers-nas-devices-among-easily-hacked-iot-devices/124877/, 2017.

[4] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communication of the ACM*, vol. 56, no. 2, 2013.

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

[6] 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 *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[7] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie, "Brick: A Binary Tool for Run-Time Detecting and Locating Integer-Based Vulnerability," in *Proc. of the Availability, Reliability and Security (ARES)*, 2009.

[8] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "PIE: Parser Identification in Embedded Systems," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[9] L. Constantin, "Hackers found 47 new vulnerabilities in 23 IoT devices at DEFCON," https://www.csoonline.com/article/3119765/security/hackers-found-47-new-vulnerabilities-in-23-iot-devices-at-def-con.html, 2016.

[10] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface Aware Fuzzing for Kernel Drivers," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[11] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A Large-scale Analysis of the Security of Embedded Firmwares," in *Proc. of the USENIX Security Symposium*, 2014.

[12] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[13] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static Detection of Vulnerabilities in x86 Executables," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[14] D. Dewey and J. T. Giffin, "Static Detection of C++ Vtable Escape Vulnerabilities in Binary Code," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[15] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Kdd*, vol. 96, no. 34, 1996.

[16] J. Feist, L. Mounier, S. Bardin, R. David, and M.-L. Potet, "Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-After-Free," in *Proc. of the Workshop on Software Security, Protection, and Reverse Engineering (SSPREW)*, 2016.

[17] J. Feist, L. Mounier, and M.-L. Potet, "Statically Detecting Use After Free on Binary Code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, 2014.

[18] M. Gyung Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[19] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proc. of the USENIX Security Symposium*, 2013.

[20] C. Heffner, "binwalk - firmware analysis tool designed to assist in the analysis, extraction, and reverse engineering of firmware images," https://github.com/ReFirmLabs/binwalk, 2014.

[21] S. Hilt, V. Kropotov, F. Mercês, M. Rosario, and D. Sancho, "The Internet of Things in the Cybercrime Underground," https://documents.trendmicro.com/assets/white_papers/wp-the-internet-of-things-in-the-cybercrime-underground.pdf, 2019.

[22] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 4, 1999.

[23] B. Insider, "Hackers once stole a casino's high-roller database through a thermometer in the lobby fish tank," https://www.businessinsider.de/hackers-stole-a-casinos-database-through-a-thermometer-in-the-lobby-fish-tank-2018-4?r=UK&IR=T, 2018.

[24] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems," in *Proc. of the ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012.

[25] M. E. Khan, F. Khan *et al.*, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *International Journal of Advanced Computer Sciences and Applications*, vol. 3, no. 6, 2012.

[26] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems," in *Proc. of the Offensive Technologies Workshop (WOOT)*, 2015.

[27] B. Krebs, "Source Code for IoT Botnet 'Mirai' Released," https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/, October 2016.

[28] D. Larochelle, D. Evans *et al.*, "Statically Detecting Likely Buffer Overflow Vulnerabilities," in *Proc. of the USENIX Security Symposium*, 2001.

[29] J. Lerch, B. Hermann, E. Bodden, and M. Mezini, "FlowTwist: Efficient Context-sensitive Inside-out Taint Analysis for Large Codebases," in *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[30] J. Leyden, "Mirai IoT botnet blamed for 'smashing Liberia off the internet'," http://www.theregister.co.uk/2016/11/04/liberia_ddos/, 2016.

[31] G.-H. Liu, G. Wu, Z. Tao, J.-M. Shuai, and Z.-C. Tang, "Vulnerability Analysis for x86 Executables Using Genetic Algorithm and Fuzzing," in *Proc. of the International Convergence and Hybrid Information Technology*, 2008.

[32] J. Lyne, "Uncovering IoT Vulnerabilities in a CCTV Camera," https://www.rsaconference.com/videos/demo-uncovering-iot-vulnerabilities-in-a-cctv-camera, 2017.

[33] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined Symbolic Taint Analysis," in *Proc. of the USENIX Conference on Security Symposium*, 2015.

[34] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[35] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The BORG: Nanoprobing Binaries for Buffer Overreads," in *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[36] J. Newsome, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[37] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proc. of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[38] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[39] S. Rawat and L. Mounier, "Offset-Aware Mutation Based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results," in *Proc. of the Software Testing, Verification and Validation Workshops (ICSTW)*, 2011.

[40] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "BootStomp: On the Security of Bootloaders in Mobile Devices," in *Proc. of the USENIX Conference on Security*, 2017.

[41] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2010.

[42] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[43] 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 *Proc. of the IEEE Symposium on Security and Privacy (SP)*, 2016.

[44] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "FirmFuzz: Automated IoT Firmware Introspection and Analysis," in *Proc. ACM CCS Workshop on IoT Security and Privacy (IoT S&P)*, 2019.
[45] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016.
[46] R. Uehara and Y. Uno, "Efficient Algorithms for the Longest Path Problem," in *Proc. International Symposium on Algorithms and Computation (ISAAC)*, 2005.
[47] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2009.
[48] Z. Wang, Y. Zhang, and Q. Liu, "RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 7, no. 8, 2013.
[49] D. B. West, *Introduction to graph theory*. Prentice hall Upper Saddle River, NJ, 1996, vol. 2.
[50] N. Woulf, "DDoS attack that disrupted internet was largest of its kind in history, experts say," https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet, 2016.
[51] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007.
[52] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares." in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.
[53] N. Zhang, S. Demetriou, X. Mi, W. Diao, K. Yuan, P. Zong, F. Qian, X. Wang, K. Chen, Y. Tian *et al.*, "Understanding IoT Security Through the Data Crystal Ball: Where We Are Now and Where We Are Going to Be," *arXiv e-print archive*, 2017.
[54] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *Proc. USENIX Security Symposium*, 2019.
[55] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, 2011.

APPENDIX

In this appendix, we present the implementation details of our prototype of KARONTE, which is based on angr [43], and we show the details of a vulnerability discovered by our prototype.

A. Functions Identification

Our prototype of KARONTE scans all the functions in the binary under analysis to find three types of functions: (i) functions that are semantically equivalent to memory comparisons (e.g., `memcmp`), which we term *memcmp-like* functions, (ii) functions that copy the content of a memory location to another (e.g., `memcpy`), which we term *memcpy-like* functions, and (iii) functions that calculate the length of a buffer, which we term *strlen-like* functions.

Intuitively, a *memcmp-like* function f should contain a loop used to compare the memory locations pointed by different function parameters of f. To find these functions, we analyze each function f of a binary b that contains at least a loop. In particular, we linearly scan the instructions in the body of the loop, and retrieve each program point p containing a memory comparisons instruction (e.g., using an opcode from the x86 `cmp` instruction family). Then, we compute a static backward slice from p up to f's entry point, and inspect f's arguments to check whether they could affect the operands of the considered memory comparison at the program point p. If so, we consider f a candidate *memcmp-like* function. Finally, we calculate the size of f (in terms of the number of basic blocks), and adopt BootStomp's threshold [40] to filter out as many false positives as possible.

The approach to find *strlen-like* functions works in a very similar manner. The difference is that we also expect these functions to contain a counter that is incremented at each iteration of the loop.

Finally, to automatically identify *memcpy-like* functions, we adopt the same approach proposed in [40].

If a function body is not available (i.e., the function is implemented in an external library not present in the firmware sample), we apply string matching heuristics on the name of the function to detect whether it belongs to one of the three function types just described.

Furthermore, as an optimization, we abstract the *memcpy-like*, *memcmp-like*, and *strlen-like* functions by providing function summaries, which we execute every time one of these functions is encountered during KARONTE's symbolic path exploration (e.g., during the BDG algorithm). With this optimization, we alleviate the path explosion problem and speed up the overall analysis, while maintaining unaltered its precision.

B. Border Binaries Discovery

As stated in Section IV, the connection mark (i.e., $\#conn$) is used as a flag, whereas the network mark (i.e., $\#net$) is used as a counter. We made this decision as we found that, in practice, calculating the connection mark feature is computationally harder than calculating the network mark.

To calculate the network mark, we need to retrieve all the memory comparisons within a binary and consider those that might refer to hard-coded network-related strings. We found that finding memory comparisons that refer to these type of strings is computationally easy, as, in practice, the addresses of these strings are referred within the basic block containing the call to the memory comparison itself.

On the other hand, to set the connection mark, we have to determine whether any data read from a network socket (i.e., the source) is passed to a *memcmp-like* function (i.e., the sink). This would involve enumerating all the possible program paths between two arbitrary program points (i.e., a read from a socket and a call to a *memcmp-like* function), which is, in the general case, unfeasible [4]. Also, in principle, we do not know if a binary contains more sources than sinks, and, therefore, a classic forward taint analysis from a source to a sink might incur in scalability issues [29]. Therefore, to alleviate these problems and increase the chances to find a path between a source and a sink, we leverage our static taint engine and perform a combination of both forward and backward static taint analyses. In particular, we bootstrap a forward taint analysis from each program point containing a source (e.g., a `recv`), and a backward taint analysis from each program point containing a sink (i.e., a *memcmp-like* function). Also, to keep the analyses tractable, we constrain the number of functions traversed by each analysis to a fixed value n_f (set to 5 in our experiments), and limit the symbolic exploration to a time limit of 10 minutes.

Nonetheless, we might fail to find a path between a source and a sink due to, for instance, an unresolved indirect control-flow transfer. Therefore, if we detect any imprecision while analyzing a function f of a binary b, we consider the analysis for f to be *incomplete*. If the number of functions not completely analyzed overcomes a fixed threshold (set to 50% in our experiments), we take the conservative decision to set the connection mark. Also, as the connection mark is operating system (OS) dependent (i.e., the analysis should know the syscall number used to read data from a socket), if the OS is unknown (e.g., in case of a firmware blob) we simply set the connection mark.

Finally, the feature cmp in our Parsing Score (see Equation 1) represents an adaptation for binaries of the feature br_fact presented by Cojocar et al. [8], and it is calculated by incrementing its value every time we find a memory comparison operation against any string.

C. Communication Paradigm Finders

As stated in Section V-A, KARONTE provides a set of CPFes to recognize the IPC paradigms, whose specifics depend on the OS of the firmware sample under analysis. Furthermore, to maintain our prototype OS-independent, and to make it able to reason about inter-process communication paradigms when some information is missing (e.g., embedded Linux distributions whose binaries are stripped by their symbols or firmware blobs), we provide our prototype with a generic CPF called the *Semantic CPF*, which abstracts from the underlying OS.

Since OS-dependent CPFes work in a similar fashion, we describe the Environment CPF, as an example of OS-dependent CPF, and the OS-independent Semantic CPF.

Environment CPF. This CPF detects whether user data is shared through the operating system environment. Given a program path (i.e., a sequence of basic blocks) between two program points p_1 and p_2, the Environment CPF checks whether there exists a block *bb* containing marks indicating that another binary is being executed (e.g., a call to `execve`). If so, this CPF scans each basic block in the program path prior to *bb*, and collects every program point p_c that contains a call to a function setting (or getting) environment variables (e.g., `setenv` or `getenv`). Finally, the Environment CPF considers each function f containing p_c, and performs a reach-def analysis from f's entry point to p_c itself to determine the values of the arguments of the function called at *pc* (e.g., the string `QUERY_STRING` in `setenv("QUERY_STRING"))`. Finally, the Environment CPF considers these values as data keys (e.g., `QUERY_STRING`).

The binary set magnification functionality (see Section V-A) infers the possible names of the binaries that are invoked in *bb*. To do this, we perform a reach-def analysis starting from the entry point of the function containing *bb* to *bb* itself, and we collect the strings used as arguments in the function call in *bb*. Finally, if we cannot resolve the names of the binaries being executed (e.g., because they are calculated at runtime), the Environment CPF finds all the binaries within the firmware sample that rely on the data keys previously recovered. We do this by retrieving all the strings in the binaries of the firmware sample, and selecting those that have at least one of the searched data keys.

Semantic CPF. Our key observation is that any communication among different processes must rely on the concept of data keys. That is, there must be some *known* information that is used as a *reference* to set, or get, some data d for another process to be accessed. Furthermore, as explained in Section II-C, data keys are often hard-coded in the binary itself as constant values (e.g., hard-coded strings).

The Semantic CPF leverages this intuition, and given a program path, it checks whether a constant value k is used to *index* a memory location to set (or to get) some data of interest (e.g., attacker-controlled data). If so, k is considered as a candidate data key, and the binary under analysis as a potential *setter* (or *getter*) for k. A typical example of inter-process communication detected by the Semantic CPF is given by memory-mapped I/O in embedded devices. In this setting, peripherals' input and output channels are mapped to predefined addresses in memory, which are hardcoded in the firmware components that need to access them.

Given a function f_c to analyze, this CPF applies two different approaches to infer if a data key is used as a reference (base or index) to manage data.

First, we taint each argument of f_c that points to constant data (e.g., a string in a `.ro` section of the binary), using different taint tags. Then, we examine every load (or store) in f_c to check whether tainted variables are referenced to read (or write) from a memory location m. For example, if a tainted variable is used as an address to write at location m, we consider f_c as a setter for the data key.

Second, if the first step does not yield a positive result, we check the structure of the function f_c itself. In particular, we assume that any set or get oriented functions should look for an entry point into a data structure relying on a provided key, to set, or get, some value. To achieve this, we assume that such a function contains a simple loop with a memory comparison function (e.g., *memcmp-like* functions) that has a parameter that points to tainted data. If these conditions are met, the Semantic CPF considers the function to be a set or get oriented function. To distinguish between the two, we scan the basic blocks corresponding to the true branch of the memory comparison function call and checks whether any of the f_c's arguments are set to a new value. If a new value is set, we identify the function f_c as a setter. In the case where a value is returned, we label the function as a getter.

Consider the example in Listing 4, which represents a snippet of code of a setter function found in one of the firmware samples in our dataset. The stack variable at offset `-32` (`R11` represents the base pointer) points to a hard-coded string (i.e., a sequence of ASCII characters null-terminated), which is, therefore, tainted by the Semantic CPF. Due to a function semantically equivalent to `memcpy` (Line 6), the taint gets propagated to the destination buffer (stack variable at offset `-28`). Then, after considering its length, the character "=" is appended to the destination buffer (Line 11) and a value (stack variable at offset `-36`) is appended to it through the `memcpy-like` function call (Line 20). Finally, since a hard-coded value is used as the offset (through its length) to copy *arbitrary* data into memory, the Semantic CPF considers this function as a candidate setter function. After manual verification, we found that the above example was indeed setting data to be used by another process, and that the stack variable at offset `-36` (Line 18) was the value of the data.

```
1  ; .text section
2  loc_A598:
3    LDR       R0, [R11, -28]   ; destination buffer
4    LDR       R1, [R11, -32]   ; data key pointer
5    LDR       R2, [R11, -24]   ; number of bytes
6    BL        0x9554           ; call to a memcpy-like
7                               ; function
8    LDR       R2, [R11, -28]
9    LDR       R3, [R11,-24]
10   ADD       R3, R2, R3
11   MOV       R2, 61           ; append '='
12   STRB      R2, [R3]
13   LDR       R3, [R11, -24]
14   ADD       R3, R3, 1
15   LDR       R2, [R11, -28]
16   ADD       R3, R2, R3
17   MOV       R0, R3           ; destination
18   LDR       R1, [R11, -36]   ; source (data value)
19   LDR       R2, [R11, -16]   ; number of bytes
20   BL        0x9554           ; call to a memcpy-like
21                              ; function
22   LDR       R2, [R11, -24]
23   LDR       R3, [R11, -16]
24   ADD       R3, R2, R3
25   ADD       R3, R3, 1
26   LDR       R2, [R11, -28]
27   ADD       R3, R2, R3
28   MOV       R2, 0
```

Listing 4: Snippet of code that uses a data key to set a data value into a local structure.

As an optimization, we leverage debugging and loading symbols (when available) to drive our Semantic CPF to *interesting* functions. For example, if a function name contains the keyword 'send' we mark it as a candidate set function, and consider it for further analysis.

D. Binary Dependency Graph Algorithm

As explained in Section V, KARONTE detects if a border binary shares user-provided data by: (i) considering the set of memory comparisons retrieved by the Border Binaries Discovery algorithm, (ii) using our taint engine to taint the involved memory locations, and, (iii) performing a taint analysis on the border binary to detect whether the binary shares some tainted data. This procedure might involve enumerating all the possible program paths in the border binary, and, therefore, it might lead to the path explosion problem. Therefore, to keep the analysis tractable, we run our taint engine up a certain time limit (set to 10 minutes in our experiments). However, as some paths might be left unexplored, our prototype might miss some valid data flows between binaries, and our BDG might not contain some valid edges. Therefore, in order to increase the path coverage within a prefixed time limit, we apply the taint to *each* function of a border binary that refers to a network-encoding string. This solution might involve more false positive edges within a BDG (thus affecting its soundness), but it decreases the likelihood of false negative edges. This heuristic gave us noticeable improvements in practice, as the program points where data is read from sockets (e.g., `recv`) might be distant (in terms of the number of instructions in an execution trace) to those where such data is shared (e.g., `setenv`). However, as network-encoding strings might be used for other purposes within a binary (e.g., as data keys), we are able to alleviate this problem by considering as a source of taint every function that refers to network-encoding strings.

E. Static Taint Analysis

Our taint engine mainly introduces two contributions: (i) taint tag dependencies, and (ii) a path prioritization strategy.

To add taint tag dependencies, we enhanced the angr's symbolic state module with an additional data structure that maps each taint tag to its dependencies. When a symbolic expression e has to be untainted, we retrieve its taint tag t_e, and all the taint tags t_{dep} that depend on t_e. Then, we consider each taint tag t_d in t_{dep}, and check whether it depends on any other taint tag other than t_e. If not, we remove the taint tag t_d (thus untainting the tainted symbolic expressions represented by t_d). Finally, we remove t_e, thus effectively untainting e. Note that, taint tag dependencies based on memory comparisons (as explained in Section VI) are created automatically.

Our path prioritization strategy aims to prioritize those paths within a function that *potentially* return tainted variables. Given a function f to symbolically explore, we build its control flow graph (CFG), and we retrieve all the *exiting* basic blocks, that is, those containing a return statement r. For each of these basic blocks, we perform a static reach-def analysis from f's entry point up to r, and collect all the possible returning values. We then prioritize those paths that *do not* always return constant values.

F. Multi-binary Data-flow Analysis

The cornerstone of the multi-binary data-flow analysis module is to estimate the size of the buffers used to send (or receive) attacker-controlled data. Our prototype provides two sub-modules for this task: the *stack-size finder* and the *heap-size finder* to detect the size of buffers allocated on stack and heap, respectively.

Given a function fs and a buffer b allocated at offset bs on fs stack, the stack-size finder scans fs body, and collects the offsets of the variables allocated on fs stack. Then, this sub-module sorts the stack offsets in ascending order, and it picks the offset bz right after bs (remember that the stack grows downward). Finally, the stack-size finder considers the buffer b as big as $|bz - bs|$.

On the other hand, given the address bh of a heap-allocated buffer b, and a function fh allocating b, the heap-size finder leverages our static taint engine to taint bh, and bootstraps a symbolic path exploration from fh's entry point. For each basic block encountered during the symbolic path traversal, this sub-module detects whether the basic block contains a call to a heap allocation function fa (e.g., `malloc`). If any of fa's arguments is tainted, or fa's returning value gets assigned to a tainted memory location, the heap-size finder considers the call to fa

```
1  void add_data_key(e, key, data) {
2    int nk = strlen(key);
3    int nd = strlen(data);
4    char* tmp = (char*) malloc(nk + nd + 3)
5    e->vars = realloc(e->vars, e->size + nk + nd + 3);
6    memcpy(tmp, key, nk);
7    tmp[nk] = "=";
8    memcpy(tmp[nk + 1], data, nd);
9    e->vars[e->n_vars] = tmp;
10   e->n_vars ++;
11   // ...
12 }
13
14 int do_serve(r){
15   env_struct* e;
16   add_data_key(e, "CONTENT_TYPE", r->content_type);
17   // ...
18   exec_bin(e, "fileaccess.cgi");
19 }
20
21 void parse_req(char* raw_data, usr_req* r){
22   while (raw_data && *raw_data) {
23     char *s = get_next_field(raw_data);
24     // ...
25     if ( !strncmp(s, "Content-Type", 12) ) {
26       // set content type info in r
27     }
28     // ...
29   }
30 }
31
32 void serve_request() {
33   usr_req* r;
34   char* raw_data;
35   raw_data = get_req_socket();
36   parse_req(raw_data, r);
37   do_serve(r);
38 }
```

Listing 5: Decompiled snippet of code of `httpd`.

for further inspection. In particular, it considers the symbolic expression of the fa's argument that represents the allocated size (e.g., the first argument in `malloc`), and leverages the z3 [5] theorem solver to concretize its value, thus retrieving the buffer b's allocated size. If the symbolic expression can be concretized to multiple values, we conservatively consider the greatest value.

G. Vulnerability Example

We provide the details of one of the vulnerabilities discovered by KARONTE [6] for the D-Link 880 firmware sample. This firmware is used on the D-Link Wireless AC1900 WiFi Gigabit routers, and it is composed of 129 different binaries executing on a Linux-based filesystem.

Two of the binaries involved in handling user's requests are the binary `httpd` and a binary called `fileaccess.cgi`. The former receives user's data from the network, whereas the latter uses such data to perform file operations.

A simplified code of `httpd` is shown in Listing 5.

First, `httpd` calls the function `get_req_socket` (Line 35) to receive user requests from the network, and stores them in the `raw_data` variable.

The content of the request is parsed by the function `parse_req` (Line 36), which also properly sets an internal data structure `r` (Line 26). Note that, the memory comparison contained in function `parse_req` (Line 25) refers to attacker-controlled data. This memory comparison is returned by our Border Binary Discover module (Section IV).

[5]https://github.com/Z3Prover/z3

[6]CVE-2017-14948

```
1  void get_content_type(char* dst) {
2    const char *haystack = getenv("CONTENT_TYPE");
3    char *haystacka;
4    haystacka = strstr(haystack, "boundary=");
5    // ...
6    strcpy(dest, haystacka + 9); // buffer overflow
7  }
8
9  int uploadfile_handler() {
10   char buff[256];
11   get_content_type(buf);
12   // ...
13 }
```

Listing 6: Decompiled snippet of code of `fileaccess.cgi`.

Then, `httpd` calls the function `do_serve` (Line 37), which prepares the execution environment for `fileaccess.cgi` and executes it. In particular, `do_serve` (Line 14) uses the function `add_data_key` (Line 16) to set the local variable `e` with attacker-controlled data. Note that, `add_data_key` (Line 1) does not impose any constraints on the size of the attacker-controlled data: it allocates a buffer `tmp` (Line 4) to accommodate arbitrarily long data. In our prototype, the function `add_data_key` was recognized by our Semantic CPF to be a setter for `httpd`.

Finally, the binary `fileaccess.cgi` is executed (trough `exec_bin`), and the variable `e` is used as its execution environment.

When `fileaccess.cgi` is executed (Listing 6), if the user's request involves uploading a file, the function `uploadfile_handler` is executed (Line 9). This function allocates a buffer of 256 bytes on the stack (Line 10), and then calls the function `get_content_type` to retrieve the content type of the user's request (at Line 1).

Unfortunately, this function contains a bug. In fact, if the variable `haystack` (which points to the environment variable identified by the data key `CONTENT_TYPE`) contains the string `"boundary="` followed by at least 257 characters, the `strcpy` function call (Line 6) will provoke a buffer overflow. KARONTE automatically identified this bug, and we reported it to D-Link, which promptly fixed the issue.