KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware

Nilo Redini*, Aravind Machiry*, Ruoyu Wang†, Chad Spensky*, Andrea Continella*, Yan Shoshtaiashvili†, 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 cybercrime 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 environment1.

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

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

#### 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.

#### 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

---

**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];
    int op;
    if (!strncmp(query, "<soap:AddRule", 13)) {
        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.

```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));
}
```
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 $\langle data\_key, data \rangle$.

### 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):

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].

2. **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).

3. **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).

4. **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).

5. **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.

### 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 by 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:

$$ps_b = \max\{\{ps_j | \forall j \in get\_functions(b)\}, \sum_{i \in \{bb, br, cmp\}} k_i \ast \#i \ast (1 + k_n \ast \#net \ast j) \ast (1 + k_c \ast \#conn \ast j)\}$$

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 Finders (or CPFes). 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

\begin{algorithm}
\begin{algorithmic}
\Function {BDG}{\text{int}_\text{locs}, B, fw} \\
\State \text{comm}_\text{info} \leftarrow \emptyset \\
\State \text{E} \leftarrow \emptyset \\
\For {each \text{b} \in B} \\
\State \text{locs} \leftarrow \text{get_locs}(\text{int}_\text{locs}, \text{b}) \\
\For {each \text{loc} \in \text{locs}} \\
\State \text{f_addr} \leftarrow \text{get_faddr}(\text{loc}) \\
\For {each \text{block} \in \text{explore_paths}(\text{f_addr})} \\
\If {\text{address}(\text{block}) == \text{loc}} \\
\State \text{buf} \leftarrow \text{get_buf}(\text{loc}) \\
\State \text{apply_taint}(\text{buf}) \\
\EndIf \\
\EndFor \\
\EndFor \\
\EndFor \\
\If {\text{matches}_\text{CPF}(\text{block})} \\
\State \text{CPF}_p = \text{get CPF}(\text{block}) \\
\State \text{k} \leftarrow \text{find_data_key_and_role}(\text{block}, \text{CPF}_p) \\
\State \text{B}_{\text{new}}, \text{int}_\text{locs}_{\text{new}} \leftarrow \text{get_new_bina}\text{ries}(\text{fw}, \text{k}, \text{CPF}_p) \\
\State \text{update binaries}(\text{B}, \text{int}_\text{locs}_{\text{new}}) \\
\State \text{comm}_\text{info} \leftarrow \text{comm}_\text{info} \cup \text{comm}_\text{info}(\text{b}, \text{block}, \text{CPF}_p, \text{k}) \\
\EndIf \\
\EndFor \\
\EndFunction
\end{algorithmic}
\end{algorithm}

(Section VI) to bootstrap a symbolic path exploration starting from the beginning of the function containing \text{loc} (function \text{explore_paths}). When the analysis reaches \text{loc}, we taint the memory location \text{buf} being referenced, i.e., the memory location being compared against the network-encoding keyword (functions \text{get_buf} and \text{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 \text{b} is sharing some tainted data \text{d}. If a \text{CPF}_p matches, i.e., it detects that the analyzed binary relies on the communication paradigm \text{p} to share some data, we leverage \text{CPF}_p to recover all of the details of the communication paradigm instance in use. More precisely, \text{CPF}_p recovers the data key \text{k} used to share data through \text{p} and infer the role (i.e., setter or getter) of the binary for \text{k} (function \text{find_data_key_and_role}) and finds other binaries within the firmware sample that might communicate through this channel (function \text{get_new_bina}\text{ries}). Newly discovered binaries are then added to the overall set of binaries to analyze. Note that, when any of these new binaries \text{B}_{\text{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 \text{b}_n, the \text{CPF}_p also retrieves the program points \text{int}_\text{locs}_{\text{new}} where the data key \text{k} is referenced, and add them to \text{int}_\text{locs}. These last two operations are performed by the function \text{update binaries}. Finally, for each analyzed binary \text{b}, we consider each CPF (cp) that matched for \text{b} over some key \text{k}, and use \text{cp} to retrieve the role of \text{b} for \text{k} (e.g., setter). Then, we create an edge between \text{b} and any other binaries that have the opposite role of \text{b} for \text{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 \text{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 \text{parse_URI} (Line 1), and propagating the taint by following the control flow of the program. When the taint analysis reaches the function call (Line 13), the Environment CPF detects that another binary is being executed, and that the \text{setenv} function is used to set the data key \text{QUERY_STRING}. Therefore, the Environment CPF establishes that the binary in analysis is a setter for \text{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 \text{QUERY_STRING} is referenced (e.g., \text{a call to the function setenv("QUERY_STRING")}).

VI. Static Taint Analysis

\text{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 \text{Karonte} combines the taint engine with the BDG, described previously, to achieve such detection.

\text{Karonte}'s taint engine is based on BootStomp [40]. Given a source of taint \text{s} (e.g., a function returning untrusted data) and a program point \text{p}, our taint engine performs a symbolic path exploration starting from \text{p}, and, every time \text{s} is encountered, the taint engine assigns a new taint ID (or tag) to the memory location receiving data from \text{s}. \text{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 \text{p}_1 is considered to be more interesting than a path \text{p}_2 if a variable of interest is tainted in \text{p}_1, and untainted in \text{p}_2.

Consider the example in Listing 3, and assume that the variable \text{user_input} (Line 14) points to tainted data. When the function \text{parse} is invoked, the variable \text{start} (Line 1) aliases \text{user_input} (i.e., they point to the same memory location), and, therefore, it points to tainted data. The function \text{parse} contains, potentially, an infinite number of paths: If the variable \text{start} is represented by an unconstrained symbolic expression, there is always a possible path passing through the \text{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
We say that a taint tag \( t \) within a function that is to be removed. Of course, the taint tag \( t \) of \text{user_input} solution we propose is to maintain the information that the semantically constrains tainted data might not be analyzed. This behavior emerges because some functions that are not available or the call is not followed to keep its code is not available or the call is not followed to keep.

Our path prioritization strategy aims to valorize those paths 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, 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 \text{user_input} (\( t_1 \)) is different than \( n \)’s tag (\( t_2 \)), user_input is not untainted, and the call to the unsafe \text{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 \text{user_input} (i.e., \( t_1 \)) depends on the taint tag of \( n \) (i.e., \( t_2 \)), and, to untaint \text{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 \( t_1 \) 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 \text{memcpy} and \text{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 \( (b_1, \text{loc}_1, cp_1, c_1), (b_2, \text{loc}_2, cp_2, c_2) \) indicates that the data associated with the data key \( k \) can flow from the binary \( b_1 \) at location \( \text{loc}_1 \) via the communication paradigm \( cp_1 \) with the set of constraints \( c_1 \) to the binary \( b_2 \) at location \( \text{loc}_2 \) 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 \( c_1 = c_2 = \bot \) (\( \bot \) 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 \( \text{loc}_1 \). Then, we use our taint engine to explore the paths between the entry point of the function containing \( \text{loc}_1 \) and \( \text{loc}_1 \) 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 \text{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 \( \text{loc}_2 \) 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., \( \text{loc}_2 \) of edge \( e_w \)) and the location where it shares it further (i.e., \( \text{loc}_1 \) 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 \( \text{loc}_1 \) 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., \( c_1 = \bot \) for \( e_r \)) or more restrictive than \( c^* \), we substitute it with \( c^* \) and add \( e_r \) to \( \text{wset} \)—thus keeping the least restrictive constraints. We iterate this phase until \( \text{wset} \) 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 \textit{memcpy-like} functions within a binary, that is, every function that is semantically equivalent to a \textit{memcpy} (Appendix A). Then, if attacker-controlled data unsafely reaches a \textit{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 \textit{memcpy-like} functions and attacker-controlled loops with the general term \textit{sinks}.

The Insecure Interactions Detection phase works as follows. First, we consider every edge \( e_f \) in a BFG, and for each node \((b, \text{loc}, cp, c) \in e_f\), we leverage the static taint engine to bootstrap a symbolic path exploration from the function \( f \) containing \( \text{loc} \). Then, when we encounter the location \( \text{loc} \), we rely on the provided CPF \( cp \) to retrieve the address of the buffer \( buf \) that references attacker-controlled data at location \( \text{loc} \) (e.g., the memory location returned by \text{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 \textit{memcpy-like} function, we retrieve the address of the destination buffer \( \text{bdst} \). Then, we retrieve the allocation point of \( \text{bdst} \) (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 \( \text{bdst} \), 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, \textsc{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., \textsc{angr}), as shown in Section X. This might result in incomplete BDGs, and, therefore, some security vulnerabilities might be left undiscovered. However, \textsc{Karonte} alleviates this problem by generating taint tag dependencies (see Section VI).

Though by default, \textsc{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 \textsc{Karonte}’s algorithm on several of the latest firmware samples available at the time of writing. Then, we evaluate \textsc{Karonte}’s performance using a dataset from related work [5]. We implemented a prototype of \textsc{Karonte} on top of \textsc{angr} [43], and, in particular, our taint engine on top of \textsc{BootStomp} [40].

A. DATASETS

We evaluated our prototype of \textsc{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: \textsc{Netgear}, \textsc{TP-Link}, \textsc{D-Link}, and \textsc{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 \textsc{BootStomp} [40] dataset, which provides us with the ground truth for our approach. \textsc{BootStomp}’s dataset is composed of 5 firmware samples. In particular, it contains two versions of Qualcomm’s Little Kernel (or \textsc{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 *.

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.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Device Series</th>
<th># Firmware Samples</th>
<th># Single Binary</th>
<th># Multi Binaries</th>
<th># Binaries</th>
<th>Avg # Border Binaries</th>
<th># Alerts</th>
<th>Avg Time [h:mm:ss]</th>
<th># Bugs</th>
<th>Single Binary Vulnerabilities</th>
<th>Multi-binary Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETGEAR</td>
<td>R/XR/WNR</td>
<td>17</td>
<td>12</td>
<td>5</td>
<td>4,773</td>
<td>7</td>
<td>36</td>
<td>17:13:45</td>
<td>23</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>D-Link</td>
<td>DIR/DWR/DCS</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>1,290</td>
<td>5</td>
<td>24</td>
<td>14:09:12</td>
<td>15</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>TP-Link</td>
<td>TD/WA/WRTX/KC</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>1,769</td>
<td>5</td>
<td>2</td>
<td>1:36:16</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Tenda</td>
<td>AC/WH/FH</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>734</td>
<td>5</td>
<td>12</td>
<td>1:01:22</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Huawei</td>
<td>ALE-L23</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4:04:37</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Nvidia</td>
<td>Nexus 9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0:23:01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Qualcomm</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1†</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5:03:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Qualcomm³</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1†</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5:03:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>53</td>
<td>38</td>
<td>15</td>
<td>8,565</td>
<td>279</td>
<td>87</td>
<td>49:09</td>
<td>51</td>
<td>17</td>
<td>34</td>
</tr>
</tbody>
</table>

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

Large-scale Dataset. To measure the scalability of KARONTE, we obtained Firmadyn’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 CPFs matched 11 and 32 times respectively, whereas the remaining CPFs 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
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. 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.

Table II shows the number of alerts generated for each step of KARONTE. For each vendor, we report the average values. The 12 confirmed vulnerabilities are being fixed as well as those bugs affecting samples sharing similar codebases (at least an additional 20).

![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.]

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

4CVE-2017-14948, PSV-2017-3121
(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 $7.85 \times 10^5$ 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 $7.15 \times 10^6$ 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).
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, ARONTE 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 ARONTE 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, ARONTE 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 ARONTE 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, ARONTE 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

<table>
<thead>
<tr>
<th>Vendor</th>
<th># Firmware Samples</th>
<th># Multi Binary (%)</th>
<th># Binaries</th>
<th># Border Binaries</th>
<th>BDG Size</th>
<th>Subgraph Cadinality</th>
<th>Subgraph Depth</th>
<th># Basic Blocks</th>
<th>Paths</th>
<th>Explored Paths</th>
<th>Time</th>
<th># Alerts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airlink101</td>
<td>1</td>
<td>(100.0%)</td>
<td>94</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>9×10^4</td>
<td>1×10^5</td>
<td>68.58K</td>
<td>3:55:44</td>
<td>13</td>
</tr>
<tr>
<td>Belkin</td>
<td>6</td>
<td>(16.7%)</td>
<td>184</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2×10^5</td>
<td>3×10^4</td>
<td>4.12K</td>
<td>0:49:46</td>
<td>1</td>
</tr>
<tr>
<td>Buffalo</td>
<td>3</td>
<td>(0.0%)</td>
<td>301</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2×10^6</td>
<td>3×10^14</td>
<td>43.00</td>
<td>0:17:01</td>
<td>0</td>
</tr>
<tr>
<td>Cisco</td>
<td>21</td>
<td>(62.8%)</td>
<td>142</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4×10^5</td>
<td>2×10^22</td>
<td>173.27K</td>
<td>5:36:15</td>
<td>4</td>
</tr>
<tr>
<td>D-Link</td>
<td>306</td>
<td>(64.1%)</td>
<td>103</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>7×10^5</td>
<td>3×10^10</td>
<td>41.64K</td>
<td>21:51:27</td>
<td>1</td>
</tr>
<tr>
<td>Foscam</td>
<td>5</td>
<td>(100.0%)</td>
<td>115</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4×10^5</td>
<td>5×10^15</td>
<td>52.20K</td>
<td>18:01:00</td>
<td>7</td>
</tr>
<tr>
<td>Immsrat</td>
<td>2</td>
<td>(0.0%)</td>
<td>640</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2×10^5</td>
<td>9×10^5</td>
<td>3.10K</td>
<td>11:05:06</td>
<td>0</td>
</tr>
<tr>
<td>Linksys</td>
<td>12</td>
<td>(8.3%)</td>
<td>404</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>8×10^5</td>
<td>2×10^103</td>
<td>23.20K</td>
<td>3:32:36</td>
<td>1</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>304</td>
<td>(27.1%)</td>
<td>115</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5×10^5</td>
<td>4×10^107</td>
<td>82.83K</td>
<td>3:54:00</td>
<td>1</td>
</tr>
<tr>
<td>OpenWrt</td>
<td>12</td>
<td>(8.3%)</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3×10^4</td>
<td>4×10^15</td>
<td>24.14K</td>
<td>1:06:16</td>
<td>0</td>
</tr>
<tr>
<td>Polycom</td>
<td>7</td>
<td>(0.0%)</td>
<td>130</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1×10^6</td>
<td>2×10^12</td>
<td>1.01M</td>
<td>31:49:22</td>
<td>8</td>
</tr>
<tr>
<td>Supermicro</td>
<td>26</td>
<td>(11.5%)</td>
<td>209</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>4×10^5</td>
<td>2×10^13</td>
<td>12.16K</td>
<td>1:54:03</td>
<td>5</td>
</tr>
<tr>
<td>Synology</td>
<td>44</td>
<td>(25.6%)</td>
<td>679</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5×10^5</td>
<td>3×10^104</td>
<td>4.55K</td>
<td>33:12:01</td>
<td>1</td>
</tr>
<tr>
<td>TP-LINK</td>
<td>3</td>
<td>(0.0%)</td>
<td>200</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>7×10^5</td>
<td>1×10^12</td>
<td>2.09K</td>
<td>2:53:15</td>
<td>1</td>
</tr>
<tr>
<td>TRENDnet</td>
<td>55</td>
<td>(27.3%)</td>
<td>156</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>6×10^5</td>
<td>2×10^118</td>
<td>14.52K</td>
<td>22:59:12</td>
<td>1</td>
</tr>
<tr>
<td>Tenda</td>
<td>4</td>
<td>(25.0%)</td>
<td>332</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>6×10^5</td>
<td>2×10^13</td>
<td>13.04K</td>
<td>5:39:25</td>
<td>1</td>
</tr>
<tr>
<td>Tomato</td>
<td>51</td>
<td>(21.6%)</td>
<td>223</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>7×10^5</td>
<td>1×10^28</td>
<td>90.36K</td>
<td>9:40:55</td>
<td>6</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>15</td>
<td>(76.6%)</td>
<td>68</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1×10^5</td>
<td>3×10^104</td>
<td>11.61K</td>
<td>3:06:21</td>
<td>2</td>
</tr>
<tr>
<td>Verizon</td>
<td>1</td>
<td>(0.0%)</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5×10^5</td>
<td>5×10^100</td>
<td>2.49K</td>
<td>0:19:02</td>
<td>1</td>
</tr>
<tr>
<td>Zyxel</td>
<td>19</td>
<td>(47.4%)</td>
<td>153</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>3×10^5</td>
<td>4×10^16</td>
<td>260.87K</td>
<td>4:46:38</td>
<td>3</td>
</tr>
<tr>
<td>forceware</td>
<td>2</td>
<td>(0.0%)</td>
<td>173</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2×10^5</td>
<td>2×10^103</td>
<td>3.00</td>
<td>0:30:18</td>
<td>0</td>
</tr>
</tbody>
</table>

| 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).
(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, ARONTE 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 ARONTE 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., ARONTE’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. RPfuzz [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], integer overflows [47], 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 ARONTE, 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 ARONTE, an approach to detect insecure interactions among components of embedded firmware. ARONTE 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 ARONTE 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 ARONTE 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


functions that are semantically equivalent to memory comparisons (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. Also, 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. 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 striped 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` (`811` 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 tainted gets propagated to the destination buffer (stack variable at offset `-28`). Then, after considering its length, the character “e” 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.

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

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

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 f and a buffer b allocated at offset bs on f’s stack, the stack-size finder scans f’s body, and collects the offsets of the variables allocated on f’s stack. Then, this sub-module sorts the stack offsets in ascending order, and it picks the offset {bs} right after bs (remember that the stack grows downward). Finally, the stack-size finder considers the buffer b as big as [bs−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
 WiFi Gigabit routers, and it is composed of 129 different 
(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

\[
\text{while } (\text{raw_data} && *\text{raw_data}) \\
\text{add_data_key(e, "CONTENT_TYPE", r->content_type);}
\]

is recognized by our Semantic CPF to be a setter for httpd. 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.

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

Attacker-controlled data. This memory comparison is returned by our Border Binary Discover module (Section IV).

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 (through 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 "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.

\[\begin{align*}
&\text{void add_data_key(e, key, data) } \\
&\text{int nk = strlen(key); } \\
&\text{int nd = strlen(data); } \\
&\text{char * tmp = (char *) malloc(nk + nd + 3); } \\
&\text{memcpy(tmp, key, nk); } \\
&\text{tmp[nk] = ";\}; } \\
&\text{memcpy(tmp[nk + 1], data, nd); } \\
&\text{e->n_vars = realloc(e->vars, e->size + nk + nd + 3); } \\
&\text{e->vars[e->n_vars] = tmp; } \\
&\text{e->n_vars ++; } \\
&\text{// ... } \\
&\text{do_serve(r); } \\
&\text{parse_req(raw_data, r); } \\
&\text{serve_request() } \\
&\text{add_data_key(e, "CONTENT_TYPE", r->content_type); } \\
&\text{exec_bin(e, "fileaccess.cgi"); } \\
&\text{int do_serve(r) } \\
&\text{env_struct* e; } \\
&\text{usr_req* r; } \\
&\text{add_data_key(e, "CONTENT_TYPE", r->content_type); } \\
&\text{exec_bin(e, "fileaccess.cgi"); } \\
&\text{void parse_req(char* raw_data, usr_req* r) } \\
&\text{while (raw_data && *raw_data) } \\
&\text{char * s = get_next_field(raw_data); } \\
&\text{if (!strncmp(s, "Content-Type", 12) ) } \\
&\text{set content type info in r } \\
&\text{// ... } \\
&\text{void serve_request() } \\
&\text{usr_req* r; } \\
&\text{char* raw_data; } \\
&\text{raw_data = get_req_socket(); } \\
&\text{parse_req(raw_data, r); } \\
&\text{do_serve(r); } \\
&\text{int uploadfile_handler() } \\
&\text{char* buff[256]; } \\
&\text{get_content_type(buff); } \\
&\text{// ... } \\
&\text{Listing 5: Decompiled snippet of code of httpd.}\]

3https://github.com/Z3Prover/z3
6CVE-2017-14948