FuzzPlanner: Visually Assisting the Design of Firmware Fuzzing Campaigns

Emilio Coppa, Alessio Izzillo[*], Riccardo Lazzeretti, Simone Lenti
Sapienza University of Rome

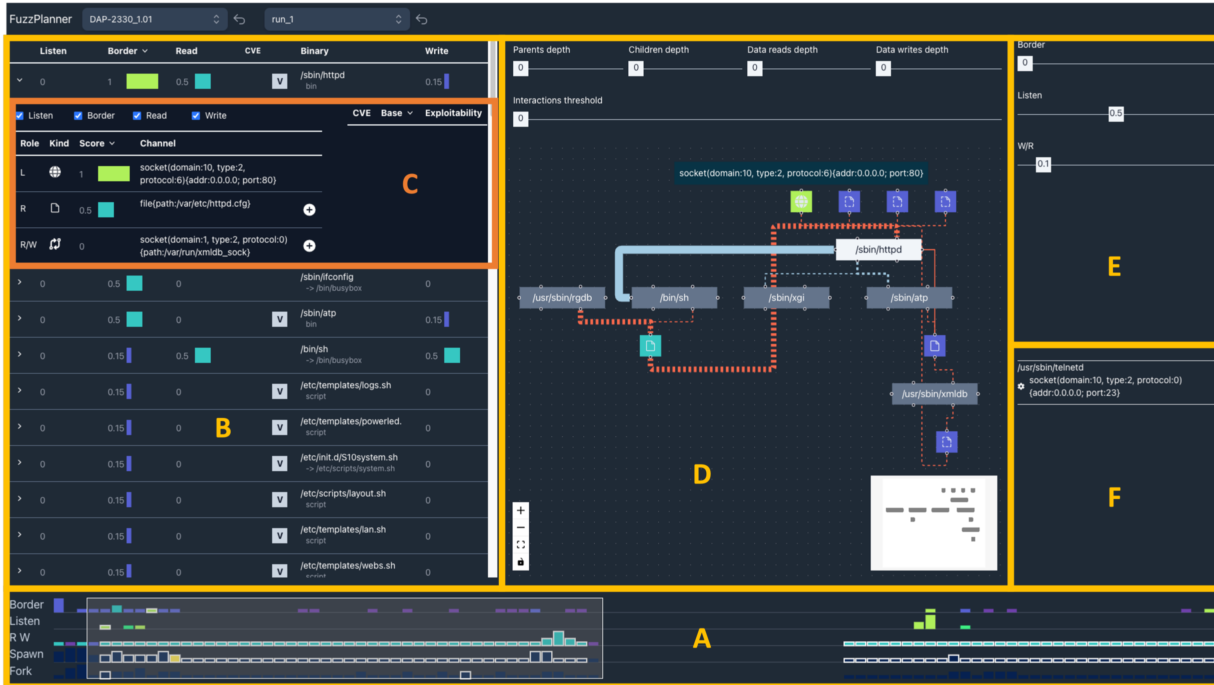

Figure 1: FuzzPlanner visual component. It comprises coordinated environments tailored to support the different phases of the analysis workflow. The *Timeline* (**A**) shows the emulation's temporal behavior representing summary information regarding the occurring interactions and enabling the narrowing of the analysis to specific time intervals. The *Binaries Table* (**B**) proposes a table-based representation of the binaries executed during the emulation, sortable according to different criteria; the selection of a binary enables its analysis in the (**C**) that shows the binary's data channel and vulnerabilities. The binary analysis also relies on the *Binary Graph* (**D**) that proposes a node-link representation of the network of interactions of the binary providing controls to extend the graph, including the neighbors' interactions, and to filter it. The *Filtering Pane* (**E**) contains additional controls to filter the analyzed interactions according to their role and the score of the data channels on which they occur. Finally, the *Experiment Pane* (**F**) contains the list of the added binary-channel tuples to be sent to the back-end to generate the fuzzing campaign execution plan.

Abstract

Embedded devices are pivotal in many aspects to our everyday life, acting as key elements within our critical infrastructures, e-health sector, and the IoT ecosystem. These devices ship with custom software, dubbed *firmware*, whose development may not have followed strict security-by-design guidelines and for which no detailed documentation may be available. Given their critical role, testing their software before deploying them is crucial.

Software fuzzing is a popular software testing technique that has shown to be quite effective in the last decade. However, the firmware may contain thousands of subcomponents with unexpected interplays. Moreover, operators may have a tight time budget to perform a security evaluation, requiring focused fuzzing on the most critical subcomponents. Also, considering the lack of accurate documentation for a device, it is quite hard for a security operator to understand *what to fuzz* and *how to fuzz* a specific device firmware.

In this paper, we present FuzzPlanner, a visual analytics solution that enables security operators during the design of a fuzzing campaign over a device firmware. FuzzPlanner helps the operator identify the best candidates for fuzzing using several innovative visual aids. Our contributions include introducing FuzzPlanner, exploring diverse analytical tools to pinpoint critical binaries, and showing its efficacy with two real-world firmware image scenarios.

Index Terms: Human-centered computing—Visualization—Visu-

[*]Corresponding author (e-mail: izzillo@diag.uniroma1.it).

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

alization techniques—Visual analytics; Security and privacy—Software and application security;

1 Introduction

Software fuzzing [29, 39, 54] is a popular technique for finding bugs in complex software. Initiatives such as OSS-Fuzz [48] from Google have proved the potential behind such a technique even when considering real-world applications deployed in critical infrastructures. In particular, Google discovered more than 28,000 bugs [31].

Academic and security researchers have proposed a large number of improvements [39] to the original technique, mixing it with more sophisticated program analysis techniques [6, 14, 15] and porting it to quite diverse application domains, including web applications, blockchains, IoT, and cloud infrastructures.

Software fuzzing is gaining traction in firmware analysis, especially for embedded devices with custom software and limited documentation [37, 52, 55, 56]. Uncertainty prevails regarding patching of known vulnerabilities (publicly tracked by the CVE system). Indeed, breaches are frequently tied to unpatched vulnerabilities due to firmware update challenges like connectivity constraints, software restrictions, and outdated systems [43].

In this scenario, governments and industries question the security of such devices [20, 49]. To tackle such concerns more systematically, they are starting to adopt two main actions. Firstly, they are devising certification schemes [21] for manufacturers to follow during device development and testing. Despite their potential benefits, the impact of these schemes could take years to manifest. The second action aims at performing security evaluations of such devices before deploying them in critical environments [36, 47]. For this task, fuzzing appears as a compelling solution given the impressive results shown in the last decade. Unfortunately, effective fuzzing of firmware images is still far from being a trivial operation. Indeed, the closed nature of most devices, the lack of proper documentation, and the large number of binaries in a firmware make fuzzing such software quite a complex endeavor [22].

Our motivation. This paper investigates how visual analytics can support a security operator interested in fuzzing a firmware image. In particular, we approach the scenario where a firmware image is available and a state-of-the-art firmware fuzzing environment, such as Firm-AFL [55], can successfully emulate the firmware. We assume the firmware source code is not fully available, and the device documentation may be incomplete. The security operator comprehends the fuzzing engine's functionalities, but uncertainty remains regarding *what* components to fuzz and *how* to fuzz them within the current firmware image context. Devices may execute numerous processes and utilize unforeseen input channels. Additionally, the operator aims to conduct a security assessment within a restricted timeframe, like a week, underscoring the need for efficient fuzzing campaign design due to the time-intensive nature of fuzzing.

Our contributions. In this paper, we propose FuzzPlanner, a novel visual analytics solution that supports a security operator in the design of fuzzing campaigns for firmware images. Our solution helps the operator understand: (a) what binaries are executed by the firmware when emulated, (b) how they interact with each other, (c) which binaries take inputs from the external world, and (d) the input channels that could be selected for fuzzing. Summarizing, the contributions of the paper are the following:

- We introduce FuzzPlanner, a novel visual analytics environment targeted at the design of fuzzing campaigns;
- We explore several analytical and visual solutions that support the user's task, pinpointing the most interesting binaries from different points of view, possibly helping the operator to consider even less immediate fuzzing setups;
- We show the effectiveness of FuzzPlanner with two usage scenarios that involve real-world firmware images.

The paper is organized as follows: Section 2 presents the domain for which the system has been developed; Section 3 discusses the related proposals; Section 4 presents the most interesting aspects of FuzzPlanner; Section 5 discusses the usage scenarios; finally, Section 6 draws the conclusions and depicts the future directions.

Release of the prototype. To facilitate extensions of our approach, we make our contributions available at:

`https://github.com/alessioizzillo/FuzzPlanner`

2 Application Domain

In this section, we introduce the application domain we are approaching. We first introduce the terminology used throughout the paper, then present binary fuzzing, and later move to firmware fuzzing. Finally, we discuss challenges related to complex firmware analyses.

Terminology. Before digging into the specific details of FuzzPlanner, we define – at least informally – a few key concepts:

- A *firmware* is an embedded device's filesystem archive containing binary code. It includes a firmware header, compressed bootloader, operating system, and root file system with critical configuration files and application binaries.
- A *binary* is any executable component included in the firmware. A binary can be *vendor-specific* - i.e., a proprietary component - or an *open-source* component.
- A *process* is the execution of a specific binary. One process can *fork* to generate a copy of itself or *spawn* a new process related to a different binary.
- The *firmware documentation* is the (possibly incomplete) knowledge base that an operator may have on a device. It could be just a high-level description of its functionalities.
- A *data channel* is the means used by a process to receive or send data. We consider the following kinds of data channels: *network socket*, such as a TCP socket; *file*, i.e., a local file in the filesystem; *virtual file*, e.g., a virtual resource exposed by Linux in `/proc` or `/sys`; *device*, e.g., a resource exposed by Linux in `/dev`; *internal*, which includes internal inter-process communication mechanisms such as UNIX sockets and pipes; and then *other* to cover other exotic (less interesting) types.
- An *interaction* is an execution event involving one or more processes. We distinguish between *data interactions* and *process interactions*. Data interaction is the (potential) use of a data channel by one or more processes within a short time window. We differentiate between *listen*, i.e., one process starts to listen on a network socket without ever receiving data, *read*, i.e., a process is reading from a data channel, *write*, i.e., a process is writing to a channel, *r/w*, i.e., a process is both reading and writing to a channel. Process interaction instead involves *spawn* and *fork* events between a parent and a child process.
- A *border* binary is a component that receives data from a channel where no other process wrote. Past works [45] restrict this term only to cases where the data is coming from the *external* world, e.g., a network socket, while we abuse this term also to capture the case where the data is coming from an *internal* resource not *generated* during the firmware execution, e.g., a configuration file that was not written by other processes. Indeed, in our working scenario, an operator may be interested in testing the firmware in different configurations, even when they may not be under the direct control of an attacker. The idea is thus to understand the reliability of the firmware when deployed in different production environments.

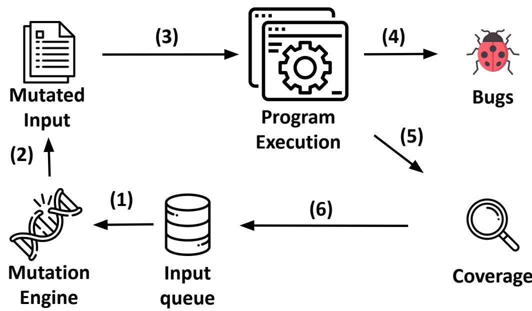

Figure 2: Workflow of a coverage-guided fuzzer.

- Given the concept of *border*, we split the data interactions of type *read* into two subtypes: (a) *border*, when the interaction is happening on a data channel where no other process has ever written to it, and (b) *non-border*, which includes the remaining *read* interactions. For the sake of simplicity, from now on, we will consider these types of data interactions: *listen*, *border*, *read* (which corresponds only to *non-border*), *write*, and *r/w*.

Several of these facts can be obtained with a static analysis over the firmware, while others (such as the processes, the data channels, the interactions, and the border property) require to be evaluated through a dynamic analysis at execution time.

Technique. Software fuzzing is a testing technique [29, 39] for discovering bugs in modern applications. In this paper, we focus on coverage-guided software fuzzing, a technique popularized by well-known projects like AFL (American Fuzzy Lop) [54] and libFuzzer [44], known for their ease of use and excellent results [31,48]. Coverage-guided fuzzing falls under grey-box fuzzing, utilizing executed path feedback to guide test-case generation. Alternatives include black-box fuzzing (easier to perform but may overlook critical paths) and white-box fuzzing (systematic path exploration but with higher overhead). Grey-box fuzzing often strikes a nice balance between these two approaches.

The main steps of a coverage-guided fuzzer, also depicted in Figure 2, are:

1. The fuzzer selects, possibly through several heuristics, one input from the *input queue*. This queue is initially populated with inputs, dubbed *seeds*, provided by a user.
2. The fuzzer applies one or more mutations to the current input. Usually, they are simple and fast input transformations that may flip a few bits, insert random values, inject interesting constants (e.g., fixed values often used in most programming languages or strings taken from a user-selected dictionary), combine the current input with other inputs from the queue, and several others. The number and type of mutations are typically randomly chosen.
3. The program under analysis is then executed over the mutated input. The execution is monitored for crashes, and the code coverage is tracked. To make it possible to track the code coverage, the code is instrumented either at compilation time (when the source code is available) or at the running time, e.g., using a dynamic binary translator such as QEMU [9].
4. Whenever the program has crashed over an input, the fuzzer stores the mutated input into a custom crashing queue, possibly performing crash deduplication to discard redundant inputs already generated in the past. A user can analyze the queue later to investigate and validate such crashes.
5. Even when a mutated input does not lead the program into a crash, the input may still be considered *interesting* by the fuzzer. In particular, a coverage-guided fuzzer evaluates the code coverage achieved by the program during the execution to assess whether the input has led to a new program behavior compared to what was observed during past executions.
6. When the input is deemed *interesting*, it is added to the queue.

This *mutate-and-run* loop effectively generates valuable inputs (possibly even crashing ones) when repeated millions of times.

From binary to firmware image. When the target of the fuzzing is a full firmware image, different strategies can be adopted. The first natural approach is to fuzz the entire firmware as a monolithic entity [42] *(system-mode emulation)*. Unfortunately, this strategy is not very effective due to two main reasons: (a) it requires emulating the entire firmware image during the fuzzing experiment, often leading to very few executions per second attempts, hurting a main requirement for the success of fuzzing, and (b) reasoning on the code coverage of an entire system makes it hard to detect specific behaviors of the firmware subcomponents and isolate non-deterministic execution aspects (which are present in most systems [22]), making the evaluation of the code coverage not valuable in several scenarios.

Hence, modern fuzzing engines [55,56] for Linux-based firmware images favor a different approach. Their main idea is to focus the fuzzing on specific subcomponents of the firmware. Indeed, emulators, such as QEMU [9], allow running a binary from a firmware in isolation as a standard user-space program on a Linux host machine *(user-space emulation)*, allowing the engine to perform a large number of executions per second attempts during fuzzing. However, whenever the binary performs a system call or accesses specific memory regions, the engine must still evaluate the results of such actions on the full system emulator to keep the execution consistent with what the actual device may do in response to such actions. Hence, these fuzzing engines mix *user-space emulation* with *system-mode emulation* to get an efficient experiment that is still accurate.

Challenges. While coverage-guided fuzzing may seem like, at least at a high level, a simple approach, in practice, it comes with several challenges when considering firmware images:

- *C1. Complexity of a firmware.* Fuzzing the entire firmware as a single unit can be ineffective due to its complexity. Instead, focusing on individual binaries and their interactions provides insights. Hence, users often prioritize fuzzing specific binaries exploiting user-space emulation. However, firmware images may contain hundreds of binaries, which may interact with each other in an unexpected and undocumented way. Hence, it could be valuable to bring some light on which binaries are executed by the firmware and which interactions are performed.
- *C2. Need of a fuzzing configuration.* When aiming to fuzz a binary, the user must identify how such binary receives the input. The input may come from a file, from a TCP socket, or even from another process, e.g., through a pipe. Unfortunately, this knowledge may not be available to the operator, requiring manual reverse engineering. Thus, it could be valuable to provide insights to the operator on how to build the configuration.
- *C3. Fuzzing needs hours to make progress.* Fuzzing incrementally builds inputs through mutations. Experimentally, it has been shown that a fuzzing experiment should last several hours, even days or weeks, to obtain effective results. Thus, when a limited time budget is given for the security evaluation, the operator can only do a few fuzzing experiments. Hence, designing a fuzzing campaign is a crucial step.

In this paper, we tackle such challenges by proposing a novel solution to support an operator during the design of a fuzzing campaign. Our goal is to effectively choose *what to fuzz and how*, by identifying interesting binaries worthy of fuzzing, determining the appropriate data channel to feed the generated test cases, and guiding the user in configuring the fuzzer for a firmware image.

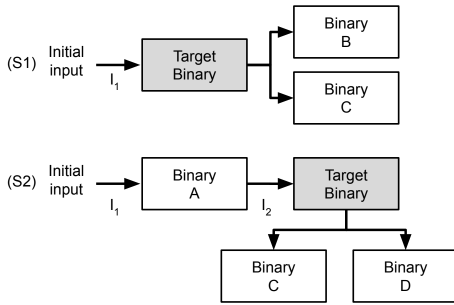

Figure 3: Two execution scenarios in a firmware.

3 RELATED WORK

This section presents the state of the art in the topics related to our proposal, i.e., software fuzzing, firmware fuzzing, and the application of visual analytic techniques to this domain. Notably, while existing works focus on the actual fuzzing stage, there is a lack of literature specifically dedicated to the planning of fuzzing campaigns.

Fuzzing. During the last decades, software fuzzing has discovered thousands of bugs and vulnerabilities in real-world software [48]. While AFL [54] and libFuzzer [44] most likely represent the two most prominent examples of coverage-guided fuzzing, the research community has proposed a large number of valuable enhancements to the original technique, including combinations with taint analysis [6, 14, 23] and symbolic execution [11, 12, 15, 17, 51], mutations driven by input format specifications [41] or language grammars [5, 50], and refinements to the internal fuzzing algorithms [10, 38]. Most of these improvements have been gradually integrated into the community-driven project AFL++ [24], the defacto successor of AFL.

Firmware fuzzing. When considering firmware fuzzing, unfortunately, there is no *Swiss army knife* that can fit all scenarios. Coverage-guided fuzzing requires running the firmware and tracking code coverage, but rehosting—running firmware on an emulator with inspection—is challenging [22]. In this paper, we focus on Linux-based firmware images where projects such as FirmAE [37] permit to emulate a large number of devices. Prior efforts fuzzed entire firmware images using system-mode emulation (e.g., TriforceAFL [42]), but scalability issues limit real-world use. Mixed approaches like FIRM-AFL [55] and EQUAFL [56] combine system-mode and user-mode emulation to fuzz specific parts. Challenges persist for customized hardware. Hybrid solutions like Avatar [53] involve emulation and on-hardware execution, but scalability concerns remain.

Visual analytics and fuzzing. Visual Analytics has been valuable in addressing research challenges in software testing and program analysis, including static and dynamic techniques [3, 8, 16, 18, 30]. Additionally, it has been applied to cybersecurity contexts, such as malware analysis [1,46] and vulnerability analysis [2,7]. However, only a few preliminary efforts have explored how visual analytics can support fuzzing. For instance, VisFuzz [57] introduces a human-assisted fuzzing approach using interactive tools to identify testing bottlenecks. FuzzSplore [25] assists users in exploring fuzzing technique evolution for specific program targets, offering insights into code coverage and input generation. Additionally, FMViz [35] sheds light on how the fuzzer modifies input bytes with mutations.

The main crucial difference between these works and our proposal is that FUZZPLANNER assists the user while designing a fuzzing campaign: our solution is valuable to choose what to fuzz and how given a complex firmware image (with hundreds of undocumented binaries). Hence, existing works are complementary to our proposal since they help analyze the results of the campaign.

4 THE VISUAL ANALYTICS SOLUTION

Software fuzzing is becoming essential for performing security evaluations of embedded devices. In this section, we describe the main aspects of FUZZPLANNER, a visual analytics solution that supports security operators during the design of a fuzzing campaign.

Scenario. In this paper, we focus on systems that are built on top of Linux but come with a minimal and vendor-specific user-space set of programs. Additionally, we consider a limited time budget for the investigation, hence requiring prioritization for specific fuzzing experiments (given that each one may last several hours or even days). In this scenario, a security operator may be interested in fuzzing the firmware based on two execution scenarios (Figure 3):

- S1 Test the security of binaries processing input data that is either potentially under the control of an attacker or that could be altered by a user when deploying the device. For instance, a natural choice is to consider vendor-specific software as the target of the fuzzing experiment since it is often unclear how it has been tested. In this scenario, the fuzzing is directly performed over the input I_1 reaching the binary.
- S2 Test the security of binaries that, although they receive preprocessed input from other binaries, likely play a pivotal role in the device functionalities due to their complex interactions with the rest of the firmware. In this scenario, the operator has two options: fuzz the target binary through the *unpreprocessed* input I_1, accepting that the preprocessing made by other binaries may make it hard to test the target binary, or fuzz directly the input I_2 of the target binary but accepting that it may not be immediately clear how easy is to generate such input for a user. The latter case may still be relevant because the operator may want to reveal bugs even when they are not readily exploitable from the current configuration[1].

Since firmware may contain and execute hundreds of binaries, FUZZPLANNER aims to support the operator in investigating these two scenarios for the most interesting targets.

High-level architecture. FUZZPLANNER is based on two main parts: (i) a back-end component that is in charge of emulating the firmware to acquire knowledge about the binaries' execution and (ii) a front-end component that is in charge of visualizing such data and supporting the operator in navigating the data deluge through novel and effective strategies.

Workflow. Figure 4 shows the workflow envisioned by our visual analytics solution. We designed this workflow based on our experience in past fuzzing campaigns for firmware images. First, in the *Emulation* phase, the operator selects a firmware image, and the emulator is spawned, allowing the operator to stimulate the emulated device. We assume that the operator knows – at least – the high-level functionalities of the device and thus can perform interesting user interactions. The back-end monitors the execution to collect data about the software behaviors. The goal of this phase is to identify what is executed by the firmware and the potential interactions among different subcomponents.

When the operator decides to stop the emulation, FUZZPLANNER moves to the *Emulation Analysis* phase, where it provides a summary of the recorded activities, pinpointing which programs, e.g., system services, were started by the system and were waiting for external input (e.g., from a network socket) but did not receive any actual data. The operator can thus decide to go back to the emulation phase to perform additional interactions. This phase is motivated by the practical observation that an operator may have partial knowledge about the internal software aspects of the firmware due to incomplete

[1]Security evaluations are often performed in environments that do not perfectly match the production environments.

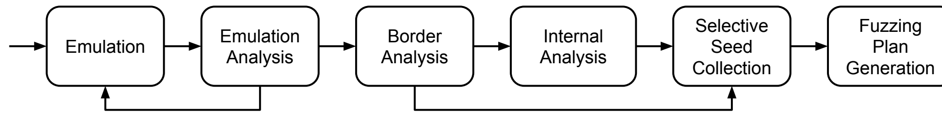

Figure 4: Main phases of the workflow supported by FUZZPLANNER.

documentation. Hence, dynamic detection of the running services may provide valuable insights. In case of need, the first two phases can be activated multiple times by the operator.

The third phase involves the *Border Analysis* designed to assist the operator interested in scenario S1, i.e., fuzzing a binary that is receiving, e.g., an external input. FUZZPLANNER allows the operator to identify the most interesting *border binaries* and decide how to fuzz them.

The next phase is dubbed *Internal Analysis*, and it is tailored for the execution scenario S2, where the operator is interested in obtaining a wide perspective about the full set of interactions happening in the device. Indeed, the operator may consider fuzzing even binaries that may receive input data from other binaries.

These last two phases enable users to discern the active role of different binaries within the firmware (C1), helping the allocation of computational resources for effective fuzzing experiments (C3).

The fifth phase, *Selective Seed Collection*, is needed to collect seed inputs for the experiments. In particular, given the fuzzing targets (binary-channel pairs), emulation is repeated, automatically replaying the user interactions to collect seeds.

Finally, in the *Fuzzing Plan Generation* phase, the operator can customize the configuration of the experiments, get a summary of the fuzzing campaign, and generate the actual plan. Hence, this last phase enables the operator to obtain a customized and tailored configuration for efficiently fuzzing the target binaries (C2).

In the next sections, we first present the essential aspects of the emulation phase, and then we dig into the discussion of the visual component of FUZZPLANNER, explaining how it can effectively support most phases of the workflow.

4.1 Emulation

When the operator selects a device firmware, the back-end performs some static and dynamic analyses over the firmware image.

Static analyses. Initially, FUZZPLANNER extracts the firmware image, analyzes its structure, identifies the binaries, and then runs a few static analyses over them. In particular, it aims at detecting whether a binary: (a) is vendor-specific and (b) has indicators of security concerns. To determine whether a binary is vendor-specific, FUZZPLANNER exploits the fact that vendors may have to disclose the integration of open-source components due to software license requirements. In practice, this attribute can be heuristically derived by scanning the documentation (including the partially available source code) or deduced by analyzing each binary (e.g., by determining whether it is a well-known software project [26, 33]). For security concerns, the back-end may mark a binary as *potentially vulnerable*. Indeed, when the binary is a well-known component, the back-end can check whether it has known CVEs [27], their severity, and the available Proof Of Concepts (PoCs). A CVE is a potential flaw because the vendor may have fixed it.

Dynamic analyses. After a preliminary static analysis of the firmware, FUZZPLANNER executes the firmware to learn about its dynamic behavior. Our emulation layer is based on FirmAE [37], a QEMU-based framework able to run a broad range of firmware images for different architectures. We exploit DECAF [34] to get coarse-grained Virtual Machine Introspection (VMI) capabilities, allowing our solution to identify processes and track system call invocations. Since our visual analytics solution requires more fine-grained information about the firmware behavior, we extended this VMI layer with dynamic analyses to accurately capture each system call's arguments and return values, track the lifetime and propagation of file descriptors, and identify interactions between processes.

FUZZPLANNER assigns an interesting score ([0, 1]) to a data channel based on a device-independent heuristic approach. The score evaluates the channel's potential for fuzzing, with high scores for channels controlled by attackers, medium scores for user-modifiable files, low scores for lightly impacted virtual files, and minimum scores for others. The heuristic focuses solely on the data channel itself, rather than how it is used. Implementation includes pattern-matching rules that are effective across devices. For instance, we assigned a 1.0 score to network sockets, a 0.5 score to `.cfg` and `.conf` configuration files, and a 0.3 score to virtual files whose content can be affected by the attacker's actions, such as the number of exchanged packets. An operator may extend such rules, exploiting the additional knowledge possibly available from the documentation.

During the emulation, FUZZPLANNER also attempts to identify several general-purpose services started by the firmware, offering the operator cues for user interactions with these services. For instance, our solution can detect HTTP servers[2] and propose URLs for browsing. It also spawns a textual console that allows the operator to run commands in the emulated system. In this phase, we expect the operator to look at the firmware documentation and perform valuable user interactions. For instance, in the case of an authenticated HTTP server, we expect the operator to retrieve the credentials, perform the login procedure and then explore the Web UI functionalities.

As documentation may overlook certain features, FUZZPLANNER monitors executed firmware processes to uncover unrecorded data channels with potential interactions. Post-emulation, the *Emulation Analysis* can spotlight prospects for additional user interactions.

4.2 Visual component

The FUZZPLANNER visual component (Figure 1) is designed to support the operator in exploiting the workflow. The underlying data model is a time varying multigraph [4] where data and process interactions connect binaries, processes, data channels. The size of the graph, the cardinality, and the software execution's dynamics call for different perspectives on the data space to support the operator at different workflow phases. The main view is composed of four coordinated environments designed to support three main perspectives on the data space: the *Binaries Table* gives an overview on the binaries and supports their prioritization according to different criteria, the *Binary Pane* and the *Binary Graph* support tasks related to the inspection of individual binaries, while the *Timeline* is designed to handle the analysis of the temporal behavior of the emulation. Two further environments, *Filtering Pane* and *Experiment Pane*, contain additional controls to respectively filter the analysis data space and manage the possible fuzzing targets identified during the analysis.

Timeline. This environment shows a temporal overview of all the data and process interactions collected during the emulation. Since thousands of interactions occur over a relatively short period, the time domain is binned in a configurable number of bins that group the interactions falling into the intervals. It adopts a matrix-based representation where the columns are the time bins (see Figure 5). The first two rows show the data interactions of type *border* and *listen*, while the third row groups *write*, *read*, *r+w* interactions. The last two rows represent the process interactions,

[2]Several embedded devices can be configured through web interfaces.

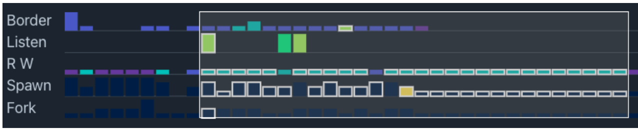

Figure 5: The *Timeline* provides a temporal summary of the interactions observed during the emulation.

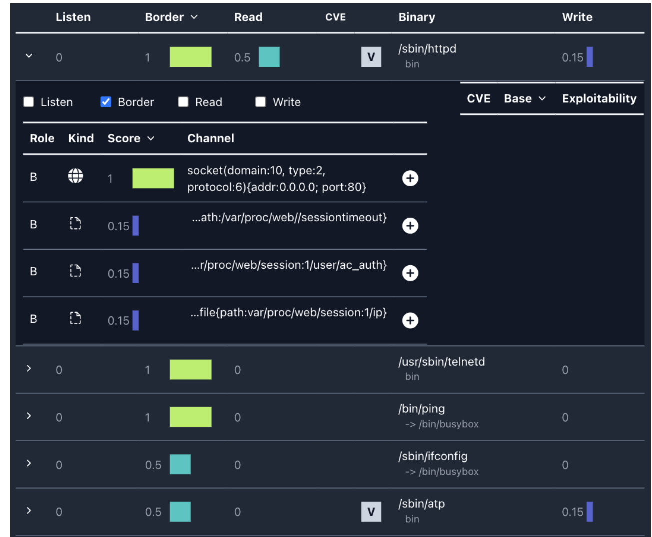

Figure 6: The *Binaries Table* lists the binaries executed during the emulation. After selecting a binary, the *Binary Pane* provides details about the different data channels used by the binary.

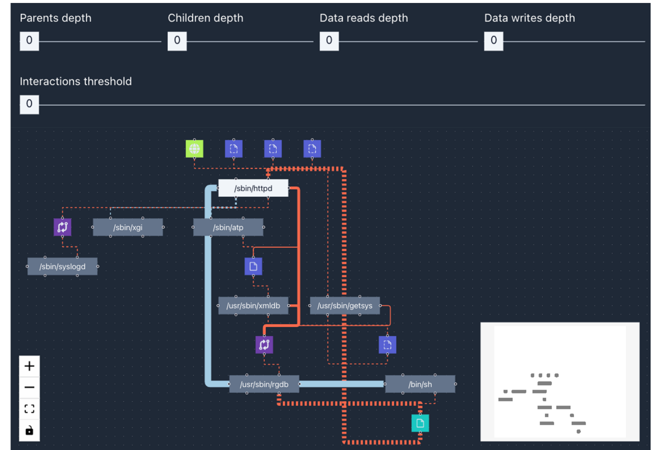

Figure 7: The *Binary Graph* shows a node-link representation of the network of interactions observed for a binary.

i.e., *fork* and *spawn*. The height of the matrix cells encodes the number of interactions of the corresponding time that occur in the time bin. For the data interactions, the color encodes the highest score of the data channels used in the interactions according to Niccoli's perceptual rainbow color scale [32], while, for the process interactions, it encodes the highest base score of the CVEs affecting the binaries spawned (or forked) in the time bin according to the Cividis color scale [40]. These two different color scales are used in all environments to encode the score of the data channels and the base score of the vulnerabilities. This encoding shows an overview of the emulation progression and highlights the occurrence of the most interesting interactions (use of data channels with a high score and execution of a process of a vulnerable binary). Furthermore, it allows the operator to narrow the analysis to specific portions of time by brushing horizontally on the matrix. Additionally, when a binary is selected, its interactions are projected on this view by highlighting the relevant bins for the binary with a white border.

Binaries Table. This environment adopts a table-based representation to enable the exploration of all the binaries executed during the emulation (see Figure 6). The ordering of the columns reflects their role: the first columns refer to the interactions in which the binary (potentially) reads, the columns related to the binary attributes follow, and finally, there is the column related to interactions in which the binary writes. The columns related to data interactions (*listen*, *border*, *read*, and *write*) can be interpreted as horizontal bar charts in which the width and the color of a bar encode the highest score of the channels involved in that interactions. Regarding the binary attributes, the table shows its type (and its target in the case of symbolic links), whether the binary is vendor-specific, and the highest (CVE) base score of the vulnerabilities affecting it through a colored square. During the different stages of the workflow, it is necessary to prioritize different aspects; thus, the table can be sorted according to its columns. The table can be filtered in two ways. First, it is updated when a time span is selected on the Timeline, and second, it is possible to set a minimum threshold of the data channel score for an interaction to be considered in the *Filtering Pane*, differentiating the threshold for the *border*, *listen*, and *read/write* interactions. Finally, by selecting a binary, it can be analyzed in the *Binary Pane* showing up embedded in the table and in the *Binary Graph* (see Figure 7) located to its right, where the first is devoted to inspecting the binary and its interactions in isolation while the latter shows the interactions network between it and the other binaries.

Binary Pane. The main visualization of this environment is a table-based representation of all the data channels used by the binary (see Figure 6). The columns represent the attributes of the data channel: how it was used by the binary (*border*, *read*, *write*, *r/w*), its kind (represented with an icon), and its score (encoded as a horizontal bar chart) and the description. In addition, the dedicated button adds the binary-channel pair to the list of experiments to be planned. Checkboxes placed above the table allow it to be filtered by excluding interactions based on their type. For example, to identify the possible fuzzing targets in scenario S1, it is possible to restrict the analysis only to border interactions. Additionally, the environment lists all the (CVE) vulnerabilities affecting the binary, allowing it to sort them according to their base or exploitability scores.

Binary Graph. This environment is complementary to the *Binary Pane* and shows the network of interactions that has the selected binary as its center (see Figure 7). For the graph, we adopted a node-link representation which is a common choice for representing call graphs [19] and is suitable to support tasks like following path(s) [28] needed for scenario S2. The nodes of the graph are binaries and data channels, and the edges encode their connections. To convey the flow of the interactions (from top to bottom), we used the *Dagre* layout algorithm, weighting the data interaction edges according to the score of the channel and process interaction edges according to the number of interactions. Each data channel is encoded as a square, with an icon representing its type and colored accordingly to its score. Edges related to writes on the channel are linked to the top of the node, while edges related to reads from the channel are linked to its bottom. The binaries are represented as rectangles and have six handles for the edges. The three handles on the center-left are devoted to process interaction edges, while the three on the right are dedicated to data interaction edges. Their positioning on the y-axis reflects the flow of interactions. The two handles on top are

dedicated to incoming edges (from parent binaries, read channels), the two handles at the bottom are dedicated to outgoing edges (to child binaries, write channels), while the ones halfway are related to bi-directional edges (with binaries that spawn and are spawned by the selected one and with channels that are used for both reading and writing). The unidirectional edges are dotted and animated to recall the flow of interactions. The data interaction edges are light-red while the process interaction edges are light-blue; the thickness of the edges is proportional to the number of interactions detected. The graph is enriched with additional means to filter or expand the graph. To support scenario S2, it can be useful to extend the analysis to the binaries communicating with the target one to verify if there are interesting channels in the "interactions flow" targeted to the selected binary. For this purpose, the graph can be enlarged by including the interactions of the "parent binaries" and "read channels" by configuring the depth of the visits from the proper slider. Otherwise, it is possible to investigate what happens after the interactions of the binary by including the visits of the "children binaries" and "written channels". The size of the graph can grow quickly; for this reason, we added another control that allows to exclude the edges (and eventually disconnected nodes) referring to a number of interactions below the threshold interactively set by the operator from the *Interactions threshold* slider.

In the next section, we will see how the environment supports the analysis workflow envisioned by FUZZPLANNER.

4.3 Enabling the workflow

The different workflow phases can be reduced to three main tasks: comparisons between binaries according to their attributes (supported mainly by the *Binaries Table*), inspection of a binary and its relationships with other binaries and data channels (supported by the *Binaries Table* and the *Binary Pane*), analysis of a subset of the interactions graph (supported mainly by the *Binary Graph*).

Emulation Analysis. In this phase, the operator evaluates the shreds of evidence collected in the emulation. In particular, it is crucial to understand whether there are binaries that could benefit from additional stimulations, bringing more insights into the interactions among binaries. The operator can sort the *Binaries Table* according to the *listen* score to identify which binaries are waiting for input but have not received one. It is then possible to select the binary and identify in the *Timeline* the time bins in which it starts to listen on at least one channel to eventually restrict the analysis to the time span around the most interesting listen interactions and inspect in detail its behavior at that time. At the end of this phase, the operator decides whether to resume the emulation or continue the analysis by filtering out the *listen* interactions from the *Filtering Pane*.

Border Analysis. A first natural choice for a user is to fuzz binaries receiving input data from, e.g., the external world. The *Timeline* provides an overview of the border interactions frequency and the presence of border interactions with high scores; the operator could eventually restrict the time span of the analysis if interesting patterns appear or could continue the analysis by sorting the *Binaries Table* according to the *border* score. The evaluation of the score and the other information lead to the analysis of interesting binaries in the *Binary Pane* and the *Binary Graph*. If the number of interactions is very high or the graph is complex, they can be filtered in the *Filtering Pane*, and the most promising binary-channel pairs can be added to the campaign both from the table or the graph.

Internal Analysis. Even non-border binaries could be valuable targets for a fuzzing campaign. These targets may arise for a variety of reasons, such as sequences of relevant interactions have been found by navigating the interaction graphs, or because binaries deemed interesting (e.g., because they have vulnerabilities or are vendor-specific) do not have border interactions. The *Binary Graph* is the main environment for this task allowing the operator to inspect

FuzzPlanner DAP-2330_1.01 run_0

	Listen	Border	Read	CVE	CWE	V	Binary	Write
›	1	0	0				/usr/sbin/telnetd bin	0
›	0.7	0.15	0.5				/usr/sbin/stunnel bin	0

Figure 8: DAP-2330: listening binaries after the first initial emulation.

FuzzPlanner DAP-2330_1.01 run_1

	Listen	Border	Read	CVE	CWE	V	Binary	Write
›	0	1	0.5			V	/sbin/httpd bin	0.15
›	0	1	0				/usr/sbin/telnetd bin	0
›	0	1	0				/bin/ping -> /bin/busybox	0
›	0	0.5	0				/sbin/ifconfig -> /bin/busybox	0
›	0	0.5	0			V	/sbin/atp bin	0.15

Figure 9: DAP-2330: border binaries (after setting a minimum threshold of 0.2 for *Border* in the *Configuration Pane*).

in the graph of interactions only the sub-portions that are relevant for the task excluding the extra information that could occlude the most interesting interactions.

Selective Seed Collection and Fuzzing Plan Generation. After the operator has confirmed in the *Experiment Pane* which firmware binaries to fuzz and specified which data channels to consider as their input source, FUZZPLANNER replays the emulation phase, repeating the user interactions. The main goal of this procedure is to collect the data received on the specified channels. This data is valuable since it could be used as the initial seeds for the fuzzing experiments. Notice that this seed collection phase is not performed during the initial emulation because it would mean tracking all interactions, yielding an excessive performance slowdown.

After the replay, FUZZPLANNER returns the initial seeds to the visual component. For each binary-channel pair, the operator can then configure its experiment by choosing: (a) which seeds to use (including possible PoCs found for the CVEs), (b) which user dictionaries to use, and (c) the time budget. Our current implementation can generate execution plans for the fuzzing engine FirmAFL [55].

5 USAGE SCENARIOS

In this section, we report our experience using FUZZPLANNER on firmware images from two real-world devices that are commercially available and lack detailed documentation. The devices we considered are the firmware image v1.01 for DAP-2330, a wireless router from D-Link, and the firmware image v1.01 for DSL-2740R, another popular wireless router from D-Link.

5.1 DAP-2330

Emulation analysis. After starting the emulation, we generate traffic on the emulated network. We then use the HTTP web interface of the router to configure several internal features. After terminating the initial emulation, we need to know which binaries are actively listening on data channels to conduct a more in-depth investigation. FUZZPLANNER pinpoints through the *Binaries Table* (Figure 8), after sorting by the *Listen* column, that the two binaries, `stunnel` and `telnetd`, are listening on some data channels: the first one is a standard TLS proxy, which mainly relays data to the

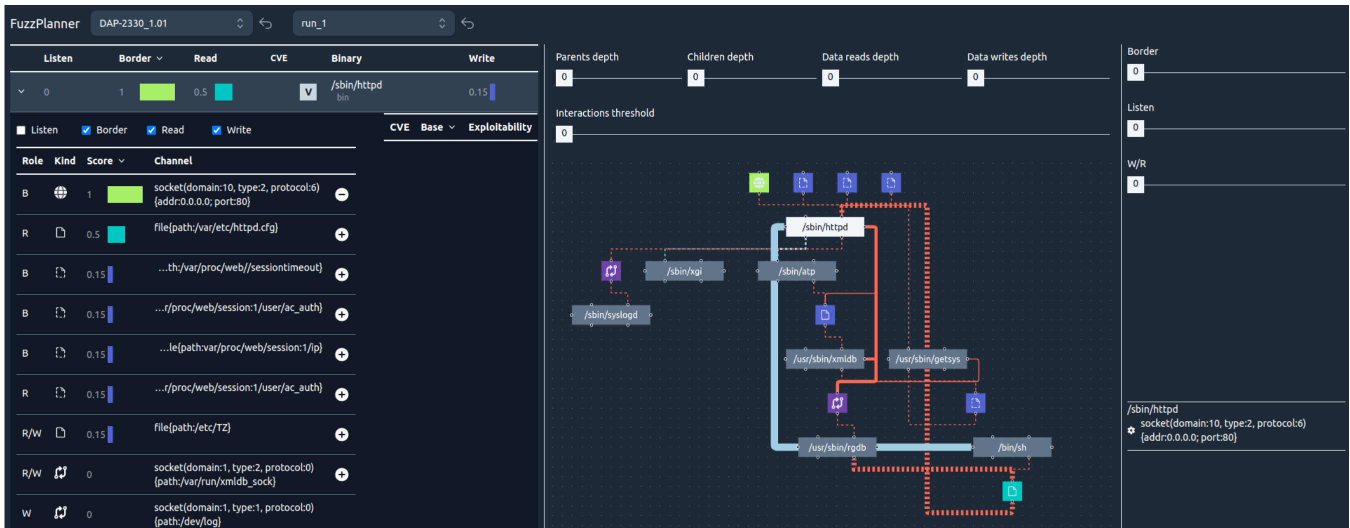

Figure 10: DAP-2330: *Border Analysis* for the binary `httpd`.

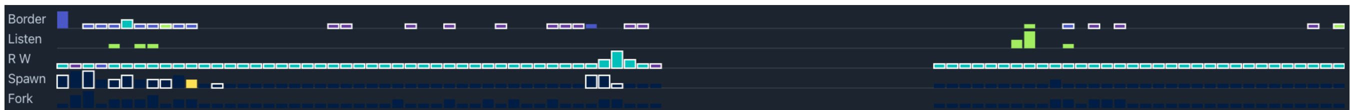

Figure 11: DAP-2330: *Timeline* where highlighted events are related to the binary `xmldb`. The *empty space* in the middle represents the time when the emulation was paused while performing the *Emulation Analysis*.

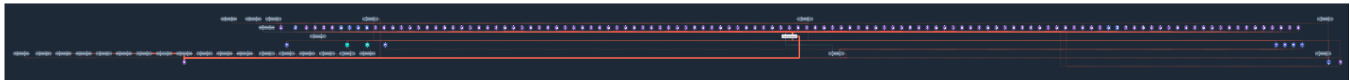

Figure 12: DAP-2330: The large and complex *Binary Graph* for the binary `xmldb`.

HTTP server and thus we skip it. In contrast, the second one is more interesting since it is a well-known virtual terminal for remote systems. We hence perform a second round of emulation where we interact with the `telnetd` console, which, however, requires authentication before allowing us to run any command. Overall, during the emulation, FuzzPlanner identifies that 87 binaries are executed out of 814 executable objects available in the firmware.

Border analysis. We then start to analyze the results to spot interesting fuzzing opportunities. We first focus on the border binaries, inspired by the execution scenario S1 (Section 4). Initially, the *Binaries Table* lists 54 distinct binaries. In order to narrow down the fuzzing candidates effectively, we leverage the *Configuration Pane* and sort the table by the *Border* column. We then set a minimum threshold of 0.2 for *Border* in the *Configuration Pane*, allowing us to restrict our investigation to 5 border binaries (see Figure 9): `httpd`, `telnetd`, `ping`, `ifconfig` and `atp`, respectively.

The vendor-specific binary `httpd` uses 9 data channels (Figure 10) but only 4 are marked with the role *B* (first column in the *Binary Pane*), meaning that they are the reason why `httpd` is a border binary. The most interesting data channel is the TCP port 80, which is likely the best candidate for direct fuzzing: indeed, the HTTP web server is often heavily customized by the vendor, and past works [13,37,55,56] have shown that it may contain critical vulnerabilities (that in several cases permits to bypass authentication). Hence, we select it for fuzzing (see Experiment Pane in Figure 10). The three other border data channels are related to virtual files from `/var/proc/web`. They appear interesting, but – with no additional documentation – it is not immediately clear whether it could make sense to fuzz them, assuming a tight time budget for the fuzzing campaign. When looking at the other data channels read by `httpd`, we notice two configuration files, `httpd.cfg` and `TZ`, respectively. However, by using the *Binary Graph*, we gain valuable insights into the relationships and interactions among binaries. We can see[3] that they are generated by other binaries (since their nodes have incoming and outcoming edges). Finally, we observe that `httpd` is communicating through a UNIX socket with the binary `xmldb`, which appears to have interactions with other binaries.

We then move our investigation to the binary `telnetd`. Being a remote shell, this binary requires authentication and then allows a user to run several commands. Since `telnetd` is likely a standard implementation of the telnet protocol, we feel that it would make more sense to invest time into fuzzing specific binaries (possibly executed from the shell) instead of testing the remote shell itself. Moreover, since this binary is not identified as potentially vulnerable from our back-end, we decide not to fuzz it, but we may reconsider our choice in a future campaign.

The binaries `ping` and `ifconfig` perform a few (although interesting) interactions but they are standard Linux command-line tools and we decide to not consider them for fuzzing.

Finally, the binary `atp` is marked as border binary because is reading from the virtual file (`/proc/net/arp`), which our heuristics marked with an *interesting score* equal to 0.5. We do not select it for fuzzing because it is not clear how a user may directly affect it. Nonetheless, in the presence of a second campaign, it could

[3] When doing a *mouse over* on a node, FuzzPlanner provides additional info, e.g., the name of a data channel, through a popup.

Figure 13: DSL-2740R: Excerpt of the *Binary Pane* for binary `dnsmasq`, showing (a subset of) the CVEs potentially affecting such binary.

be worth an experiment. Interestingly, when observing the *Binary Graph* for `atp`, we notice that this binary, similarly to `httpd`, is communicating with `xmldb`.

Overall, during *Border Analysis*, we select for fuzzing only the binary `httpd` through the TCP port 80. Nonetheless, we already have insights about the role of `xmldb` in several interactions, making it a natural candidate for investigation during the *Internal Analysis*.

Internal Analysis. We start again from the *Binaries Table*, sorting it by the column *Read*, trying to identify non-border binaries that could be worthy of a fuzzing experiment. One of the first binaries emerging in the list is `xmldb`, which we decide to analyze in detail. After selecting it, the *Timeline* (Figure 11) immediately shows that this binary is involved in a large number of interactions. We then look at its *Binary Graph*, which, as shown by Figure 12, contains an impressive number of interactions. Overall, it takes part in 129 interactions. After slightly zooming in, we can easily identify one data channel related to the UNIX socket `xmldb_sock`, that is used by more than 20 binaries. We hypothize that `xmldb` is an essential binary for the firmware, which is used as a central bridge by several other binaries for many tasks and is in charge of performing several low-level tasks. Without more extended documentation and without performing reverse engineering of such binary, it is hard to confirm this hypothesis. However, we decide to take the risk and select it for fuzzing through the data channel `xmldb_sock`. Continuing the analysis, we do not find additional promising candidates and thus we go on with the remaining phases envisioned by FUZZPLANNER.

Selective Seed Collection and Fuzzing Plan Generation. FUZZPLANNER repeats the user interactions performed during the initial emulation to collect the seeds relevant for the fuzzing of `httpd` and `xmldb`. During the *Fuzzing Plan Generation*, we decide to use the collected seeds, enabling the use of a HTTP *dictionary* when fuzzing `httpd`, and then assign a time budget of 84 hours to each experiment (1 week in total). We then obtain the actual plan.

5.2 DSL-2740R

Emulation analysis. Similarly to the other firmware, we generate traffic on the emulated network and interact with the web interface, setting firewall rules and changing different system configurations. After terminating the emulation, FUZZPLANNER shows that the firmware is listening on several network sockets, including UDP port 53 with the binary `dnsmasq`. We thus decide to resume the emulation and stimulate this service with a few DNS requests. Overall, FUZZPLANNER identifies that 34 binaries are executed out of 256 executable objects available in the firmware.

Border analysis and Internal Analysis. Differently from the first usage scenario, we decide to start our investigation by considering binaries that are potentially vulnerable. Exploiting the *Binaries Table* (Figure 13), we sort the binaries by the *CVE* column. We immediately notice that two well-known open source binaries, `busybox` and `dnsmasq`, have several vulnerabilities potentially affecting them.

The binary `busybox` is a crucial component in an embedded system, as it combines tiny versions of several common UNIX utilities (such as `cp`, `ls`, `mv`, etc.) into a single small executable. This makes it potentially pivotal in several interactions. However, while the CVEs primarily refer to the single binary `busybox` in their reports, the actual vulnerabilities are only exploitable when executing `busybox` for a specific utility (e.g., CVE-2018-1000517 targets the `wget` implementation from `busybox`). Without a better (parsable) description of the CVEs for our back-end, we decide to not fuzz it for this campaign and maybe reconsider it in the future.

We then move our analysis to `dnsmasq`, that our back-end detects as potentially vulnerable to a large number of CVEs with high severity. Moreover, this is a border binary; hence it is crucial to evaluate whether it could be exploitable by an attacker sending, e.g., a simple DNS request. Therefore, we select it for fuzzing, picking the UDP port 53 on IPv4 as data channel.

Selective Seed Collection and Fuzzing Plan Generation. FUZZPLANNER repeats the user interactions performed during the emulation to collect the seeds relevant for the fuzzing of `dnsmasq`. During the *Fuzzing Plan Generation*, we decide to use the collected seeds plus a few PoCs that our back-end was automatically able to obtain from the CVEs, as the initial set of inputs. Finally, we assign a time budget of 168 hours to the experiment and ask for its plan.

6 CONCLUSIONS

Security evaluations of embedded devices may often involve firmware fuzzing. However, one of the first obstacles to such practice is understanding *what* should be fuzzed and *how* within a firmware. Indeed, firmware may contain hundreds of software components, performing unexpected and undocumented behaviors. FUZZPLANNER has been specifically designed to support a security operator during the design of a fuzzing campaign. Through novel visual analytics aids, it may provide valuable insights to identify the best candidates for the fuzzing experiments. In the remainder of this section, we discuss possible directions for future work.

Multi-stage fuzzing. FUZZPLANNER could integrate the idea of a *multi-stage* fuzzing campaign, where after starting a first round of experiments, the operator can monitor the current results and exploit them to design additional fuzzing experiments. We believe that existing works (Section 3) may be integrated to improve the feedback given to the operator.

Different fuzzing engines. The fuzzing plan generation could be extended to support different engines. FirmAFL currently struggles at indirectly fuzzing a binary, which requires to carry on the execution of more than one binary. For this reason, when selecting this feature, we rely on (the less performant) full system-mode emulation. Also, we would like to exploit a fuzzing engine that integrates AFL++, an improved version of AFL, allowing us to expose to the operator more complex (but potentially valuable) fuzzing configurations during the *Fuzzing Plan Generation* phase.

Advanced planning strategies. The *Fuzzing Plan Generation* phase may be improved to support the design of experiments running in parallel on different cores of the same machine, or even in parallel on different machines. We believe this direction shares traits with other computer science problems requiring planning and scheduling strategies. Hence, we plan to look at such existing proposals.

Tool Evaluation. We plan to perform a more extended evaluation of the tool. From one side, we would like to perform a user study to get feedback about the usefulness of the functionalities from the tool. On the other side, we would to evaluate whether

FUZZPLANNER can indeed help the user designing the optimal experiments given a specific firmware and a limited time budget.

ACKNOWLEDGEMENTS

This work is supported, in part, by: Project ECS 0000024 Rome Technopole (CUP B83C22002820006), Project PRIN 2022 FARE (202225BZJC, CUP D53D2300838006), Sapienza University of Rome project "Secure ANd PrivatE Information sharing (SANPEI)", and project "SERICS" (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

REFERENCES

[1] M. Angelini, L. Aniello, S. Lenti, G. Santucci, and D. Ucci. The Goods, the Bads and the Uglies: Supporting Decisions in Malware Detection through Visual Analytics. In *Proceedings of the 14th IEEE Symposium on Visualization for Cyber Security*, VizSec '17, 2017. doi: 10.1109/VIZSEC.2017.8062199

[2] M. Angelini, G. Blasilli, P. Borrello, E. Coppa, D. C. D'Elia, S. Ferracci, S. Lenti, and G. Santucci. ROPMate: Visually Assisting the Creation of ROP-based Exploits. In *Proceedings of the 15th IEEE Symposium on Visualization for Cyber Security*, VizSec '18, 2018. doi: 10.1109/VIZSEC.2018.8709204

[3] M. Angelini, G. Blasilli, L. Borzacchiello, E. Coppa, D. C. D'Elia, C. Demetrescu, S. Lenti, S. Nicchi, and G. Santucci. SymNav: Visually Assisting Symbolic Execution. In *Proceedings of the 16th IEEE Symposium on Visualization for Cyber Security*, VizSec '19, 2019. doi: 10.1109/VizSec48167.2019.9161524

[4] D. Archambault, J. Abello, J. Kennedy, S. Kobourov, K.-L. Ma, S. Miksch, C. Muelder, and A. C. Telea. Temporal Multivariate Networks. In *Multivariate Network Visualization*. 2014. doi: 10.1007/978-3-319-06793-3_8

[5] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. NAUTILUS: fishing for deep bugs with grammars. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS '19, 2019.

[6] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS '19, 2019.

[7] H. Assal, S. Chiasson, and R. Biddle. Cesar: Visual representation of source code vulnerabilities. In *Proceedings of the 13th IEEE Symposium on Visualization for Cyber Security*, VizSec '16, 2016. doi: 10.1109/VIZSEC.2016.7739576

[8] I. Bacher, B. Mac Namee, and J. D. Kelleher. On using tree visualisation techniques to support source code comprehension. In *Proceedings of the Fourth IEEE Working Conference on Software Visualization*, VISSOFT '16, 2016. doi: 10.1109/VISSOFT.2016.8

[9] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, ATEC '05, 2005.

[10] M. Boehme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2019. doi: 10.1109/TSE.2017.2785841

[11] L. Borzacchiello, E. Coppa, and C. Demetrescu. FUZZOLIC: Mixing fuzzing and concolic execution. *Computers & Security*, 2021. doi: 10.1016/j.cose.2021.102368

[12] L. Borzacchiello, E. Coppa, and C. Demetrescu. SENinja: A symbolic execution plugin for Binary Ninja. *Elsevier SoftwareX*, 2022. doi: 10.1016/j.softx.2022.101219

[13] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium*, NDSS '16, 2016. doi: 10.14722/ndss.2016.23415

[14] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, S&P '18, 2018. doi: 10.1109/SP.2018.00046

[15] J. Choi, J. Jang, C. Han, and S. K. Cha. Grey-box concolic testing on binary code. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*, ICSE '19, 2019. doi: 10.1109/ICSE.2019.00082

[16] E. Coppa. An Interactive Visualization Framework for Performance Analysis. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '14, 2014. doi: 10.4108/icst.valuetools.2014.258172

[17] E. Coppa, H. Yin, and C. Demetrescu. SymFusion: Hybrid Instrumentation for Concolic Execution. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, 2022. doi: 10.1145/3551349.3556928

[18] S. Devkota and K. E. Isaacs. CFGExplorer: Designing a Visual Control Flow Analytics System around Basic Program Analysis Operations. *Computer Graphics Forum*, 2018. doi: 10.1111/cgf.13433

[19] S. Devkota and K. E. Isaacs. CFGExplorer: Designing a Visual Control Flow Analytics System around Basic Program Analysis Operations. *Computer Graphics Forum*, 2018. doi: 10.1111/cgf.13433

[20] ENISA. Guidelines for securing the Internet Of Things. `https://www.enisa.europa.eu/publications/guidelines-for-securing-the-internet-of-things/@@download/fullReport`, 2020. [Online; accessed 12-Jul-2023].

[21] European Union. The Cybersecurity Act. `https://digital-strategy.ec.europa.eu/en/policies/cybersecurity-act`, 2023. [Online; accessed 12-Jul-2023].

[22] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson. Sok: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ACM ASIACCS '21, 2021. doi: 10.1145/3433210.3453093

[23] A. Fioraldi, D. C. D'Elia, and E. Coppa. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '20, 2020. doi: 10.1145/3395363.3397372

[24] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++ : Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies*, WOOT '20, 2020.

[25] A. Fioraldi and L. P. Pileggi. Fuzzsplore: Visualizing feedback-driven fuzzing techniques. *CoRR*, 2021. doi: 10.48550/arxiv.2102.02527

[26] Firmware Analysis and Comparison Tool (FACT). Plugin for detecting software components. `https://github.com/fkie-cad/FACT_core/tree/master/src/plugins/analysis/software_components`, 2023. [Online; accessed 12-Jul-2023].

[27] Firmware Analysis and Comparison Tool (FACT). Plugin for software version to CVEs. `https://github.com/fkie-cad/FACT_analysis-plugin_CVE-lookup`, 2023. [Online; accessed 12-Jul-2023].

[28] M. Ghoniem, J.-D. Fekete, and P. Castagliola. On the Readability of Graphs Using Node-Link and Matrix-Based Representations: A Controlled Experiment and Statistical Analysis. *Information Visualization*, 2005. doi: 10.1057/palgrave.ivs.9500092

[29] P. Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 2020. doi: 10.1145/3363824

[30] J. R. Goodall, H. Radwan, and L. Halseth. Visual analysis of code security. In *Proceedings of the Seventh International Symposium on Visualization for Cyber Security*, VizSec '10, 2010. doi: 10.1145/1850795.1850800

[31] Google. Taking the next step: OSS-Fuzz in 2023. `https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html`, 2023. [Online; accessed 12-Jul-2023].

[32] M. Hall, E. Bianco, and M. Niccoli. How to assess a colourmap. In *52 Things You Should Know about Geophysics*, 52 Things You Should Know About... Agile Libre.

[33] I. U. Haq and J. Caballero. A survey of binary code similarity. *ACM Computing Surveys*, 2021. doi: 10.1145/3446371

[34] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*, 2017. doi: 10.1109/TSE.2016.2589242

[35] A. Hussain and M. A. Alipour. Fmviz: Visualizing tests generated by AFL at the byte-level. *CoRR*, 2021. doi: 10.48550/arXiv.2112.13207

[36] Italy's National Cybersecurity Authority. National Assessment and Certification Centre. https://www.acn.gov.it/en/agenzia/organizzazione/cvcn, 2023. [Online; accessed 12-Jul-2023].

[37] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim. FirmAE: Towards large-scale emulation of iot firmware for dynamic analysis. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, 2020. doi: 10.1145/3427228.3427294

[38] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium*, USENIX Security '19, 2019.

[39] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019. doi: 10.1109/TSE.2019.2946563

[40] J. R. Nuñez, C. R. Anderton, and R. S. Renslow. Optimizing colormaps with consideration for color vision deficiency to enable accurate interpretation of scientific data. *PLOS ONE*, 2018. doi: 10.1371/journal.pone.0199239

[41] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019. doi: 10.1109/TSE.2019.2941681

[42] N. G. Plc. TriforceAFL. https://github.com/nccgroup/TriforceAFL, 2023. [Online; accessed 12-Jul-2023].

[43] Ponemon Institute. Costs and Consequences of Gaps in Vulnerability Response. https://www.servicenow.com/lpayr/ponemon-vulnerability-survey.html, 2019. [Online; accessed 12-Jul-2023].

[44] L. project. LibFuzzer. https://llvm.org/docs/LibFuzzer.html, 2023. [Online; accessed 12-Jul-2023].

[45] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *Proceedings of the 41th IEEE Symposium on Security and Privacy*, S&P '20, 2020. doi: 10.1109/SP40000.2020.00036

[46] G. R. Santhanam, B. Holland, S. Kothari, and J. Mathews. Interactive visualization toolbox to detect sophisticated android malware. In *Proceedings of the 14th IEEE Symposium on Visualization for Cyber Security*, VizSec '17, 2017. doi: 10.1109/VIZSEC.2017.8062197

[47] S. Schmitz-Berndt and P. G. Chiara. One step ahead: mapping the italian and german cybersecurity laws against the proposal for a nis2 directive. *International Cybersecurity Law Review*, 2022. doi: 10.1365/s43439-022-00058-7

[48] K. Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany, 2017. [Online; accessed 12-Jul-2023].

[49] UK HCSEC. Annual report on Huawei cyber security evaluation centre (hcsec) oversight board. https://www.gov.uk/government/publications/huawei-cyber-security-evaluation-centre-oversight-board-annual-report-2020, 2020. [Online; accessed 12-Jul-2023].

[50] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*, ICSE '19, 2019. doi: 10.1109/ICSE.2019.00081

[51] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, USENIX Security '18, 2018.

[52] J. Yun, F. Rustamov, J. Kim, and Y. Shin. Fuzzing of embedded systems: A survey. *ACM Computing Surveys*, 2022. doi: 10.1145/3538644

[53] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium*, NDSS '14, 2014. doi: 10.14722/ndss.2014.23229

[54] M. Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2023. [Online; accessed 12-Jul-2023].

[55] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation. In *Proceedings of the 28th USENIX Conference on Security Symposium*, USENIX Security '19, 2019.

[56] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun. Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '22, 2022. doi: 10.1145/3533767.3534414

[57] C. Zhou, M. Wang, J. Liang, Z. Liu, C. Sun, and Y. Jiang. Visfuzz: Understanding and intervening fuzzing with interactive visualization. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, 2019. doi: 10.1109/ASE.2019.00106