

# Enhanced Fuzzing Framework for ESP32 Microcontrollers

Moritz Buhl

Department of Informatics  
Technical University of Munich  
Garching near Munich, Germany  
m.buhl@tum.de

**Abstract**—As the number of connected IoT devices increases, so does the interest in detecting flawed protocol implementations and other security vulnerabilities in the software running on these devices. One of the most popular microcontrollers in IoT applications is the ESP32. Until recently, it has not been easy to find vulnerabilities in the firmware of ESP32 devices on a large scale.

One solution for testing all exposed network endpoints and producing accurate and reproducible crashes is fuzzing. To enable easier analysis for security flaws of ESP32 software, Börsig *et al.* presented the ESP32-QEMU-FUZZ framework. The framework supports blackbox, greybox and guided whitebox fuzzing using QEMU and Honggfuzz.

This paper reproduces the previous work, describes the framework internals in detail and adds improvements. Adjustments to the framework increase the iterations per second by a factor of six utilizing parallelization. We furthermore increase the usability of the framework by reducing the steps for the initial setup and provide a way to integrate the framework into development processes. Finally, we compare the emulated fuzzing approach with other plausible whitebox approaches.

**In conclusion the framework is a proof of concept and lacks features and performance for use outside of dynamic blackbox fuzzing. It is likely that analyzing unknown ESP32 firmware with this framework will not work.**

what is that?

**Index Terms**—esp32, qemu, fuzzing

## I. INTRODUCTION

Networked appliances at home and in industrial applications are more and more common in daily life. Due to the low power requirements and adequate compute power — especially for small amounts of sensor data and infrequent communication — it is possible to add a wireless chip to electronics as an afterthought. However, this creates new challenges for network security as these new devices come online to talk various application protocols and handle diverse data formats. With the rising amount of implementations for data formats and common protocols, bugs and security threats might emerge.

One common system on a chip (SoC) used in these applications is the ESP32. The ESP32 SoC microcontroller is used in IoT and other industrial applications, commercially available products include development boards, lamps, light bulbs, LED strips, power relays, sockets, switches, monitoring devices, fans, gateways, and entertainment devices with screens [1]. Due to its IO capabilities, the microcontroller is commonly used to process sensor data and control circuits based on network input. The affordable price in combination with the

versatility of provided interfaces and the wireless connectivity ensures the ESP32 SoC is used in network-related applications. This integration of the ESP32 SoC makes it a good target for finding vulnerabilities in common file formats and network protocol implementations. One often used method to search for errors and security vulnerabilities is fuzzing.

Fuzzing is widely used in development, testing, and security research to find flaws in input validation, error handling, and program flow. This is because fuzzing can cover a breadth of inputs on multiple target locations while the initial setup is quick and easy compared to other techniques to ensure correctness [2]. Results from a fuzzer are usually very accurate and reproducible, the technique has a low rate of false positives.

However, while fuzzing an application has a low setup cost and scales very well, fuzzing IoT appliances provides new challenges. Enormous amounts of hardware targets exist due to the pairing of microcontrollers with peripheral devices in different use cases. For each target, it is necessary to buy the physical hardware and set it up for testing. This possibly introduces inconsistencies and does not scale. Furthermore, to utilize feedback-driven fuzzing it might be necessary to attach a debugging probe, or to modify the firmware as the firmware does not report application-state information.

Börsig *et al.* present a framework for fuzzing ESP32 firmware images utilizing Honggfuzz in combination with a QEMU fork to support the ESP32 microcontroller [3]. In the paper, they use blackbox fuzzing in the ESP32-QEMU-FUZZ framework, as well as coverage-guided fuzzing using greybox and whitebox techniques. This framework has the potential to simplify development, testing, quality assurance, and security research on ESP32-based products because it uses emulation, does not require any specialized hardware, speeds up the initial setup, and is highly parallelizable. Using ESP32-QEMU-FUZZ in development Continuous Integration (CI) pipelines allows for penetration testing on many application endpoints.

what is this?

We evaluate the use of the ESP32-QEMU-FUZZ framework for ESP32 security research, but also for development and testing to use the framework in a CI pipeline. To do that, we improve the user interface for easier application to a wider range of firmware images. We provide the tooling to use the fuzzing framework on new firmware images in a CI pipeline style. Finally, we implemented parallelization for

guided whitebox fuzzing to achieve more iterations per second and thus find crashes faster. The approach scales with the host systems core count.

## II. RELATED WORK

This work reproduces and extends the work on the ESP32-QEMU-FUZZ framework by Börsig *et al.* [3]. They presented various approaches for fuzzing ESP32 machines and published the framework code and a sample application [4]. In addition to the emulation framework, they also created a hook to attach a fuzzer to an actual ESP32 utilizing a JTAG debugger.

Harzer Roller [5] describes linker-based instrumentation for call-path and return-path execution tracing and crash dump extraction. A reference implementation for the ESP8266, the predecessor of the ESP32 microcontroller exists as a proof of concept. Tracing and crash detection utilizing the instrumentation also allows for guided fuzzing, but the injection described in the paper requires a hardware setup and uses the ISO-Layer 2 and 3 fuzzer BooFuzz.

The Drifuzz [6] fuzzer presents a thematically similar topic, targeting wireless and Ethernet drivers. Drifuzz skips necessary initialization steps by using so-called golden seeds with concolic execution. They target device drivers and their MMIO and DMA interface using QEMU. Whereas drivers on traditional platforms are required, the ESP32 WiFi and Medium Access Control (MAC) functionalities are tightly coupled to the microcontroller and vendor provided. This lack of flexibility changes the job of a WiFi driver and makes the initialization easier. Because of this, we are not interested in this layer of the operating system. To enable more possibilities for security research and cover a broader range of applications, we focus on higher protocol layers instead.

The Frankenstein Bluetooth fuzzing framework focuses on another ISO-Layer 2 protocol, also utilizing QEMU but on different platforms and on fuzzing closed-source firmware. We did not want to restrict our research exclusively to Bluetooth and therefore did not further consider Frankenstein.

Another Bluetooth-based fuzzing framework is Braktooth, which focuses on the ESP32 hardware using OTA (over-the-air) fuzzing. It utilizes hooks, similar in style to the Harzer Roller project.

Greyhound [7] is a directed fuzzing methodology that sends WiFi frames generated from a protocol specification and mutates these to uncover incompliant responses, crashes, and security vulnerabilities. It found a vulnerability in the ESP32 implementation. The focus on the WiFi ISO-Layer 2 protocol is currently not possible with the ESP32-QEMU-FUZZ framework and instead we focus on application-layer protocols.

## III. BACKGROUND

This section discusses the three main components of the ESP32-QEMU-FUZZ project: the ESP32 microcontroller itself, the QEMU project for emulating ESP32 firmware on another computer, and the background into different fuzzing techniques that ESP32-QEMU-FUZZ applies.

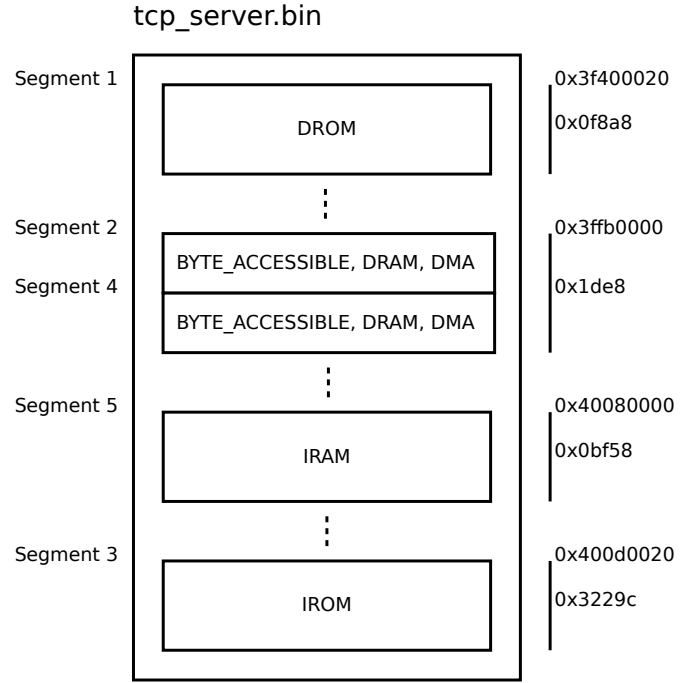


Fig. 1. Exemplary layout of the `tcp_server` firmware image.

### A. ESP32 and Xtensa

ESP32 is a low-cost microcontroller often based on Xtensa [8] LX6 cores utilizing a Harvard Architecture with separate data and instruction buses, two CPUs with symmetric address mapping, cryptographic accelerators, various external interface controllers, and most significantly integrated WiFi and Bluetooth functionality [9] [10].

Contrasting von Neumann architecture-based desktop hardware with different security features, this nonetheless does not mean findings by a fuzzer are not exploitable by design, see *Code Injection Attacks on Harvard-Architecture Devices* by Francillon and Castelluccia [11]. Recently, Lehniger et al. showed that it is possible to use Return-Oriented Programming (ROP) on the Xtensa architecture [12]. Furthermore, the restrictions of a microcontroller often come with less extensive checks for data handling in standard libraries due to memory and processing restrictions making logical programming errors potentially more damaging [13].

Figure 1 shows the different memory segments available in the later-used firmware image of the example tcp server. It shows that data (DROM and DRAM) and instructions (IROM and IRAM) are strictly separated. The data segments are below the address `0x40000000`, memory mapped peripheral devices are between `0x3FF00000` and `0x40000000`, and instructions start after `0x40000000`.

In addition to a program counter (pc) register, that tracks the next instructions, the Xtensa Instruction Set Architecture (ISA) [14] defines 16 32-Bit address registers (a0 – a15) which are used for general purpose computing, i.e. holding addresses and data. A shifts amount register (sar) holds the number

of bits the shift-related instructions shift the target register. Furthermore, the LX6 cores on the ESP32 implement the optional Loop instructions, which add the registers `lbeq`, `lend`, and `lcount`. These registers can replace the branch instruction at the top of each loop by defining the start and end locations of the loop and the count of loop executions. The `threaptr` register allows for the implementation of thread-local storage (TLS), i.e. global-like variables that exist per thread. Information on interrupts and exceptions is stored in the processor state `ps` register. To store multiple logical evaluations before branching on them, the Boolean register `br` and additional instructions are enabled on the ESP32. Because the ESP32 has two cores, the conditional store comparison data register `scompare1` synchronizes the processors and allows for inter-process communication. The multiply-accumulate registers `acclo`, `acchi`, `m0`, `m1`, `m2`, and `m3` exist for Digital Signal Processing (DSP). Lastly, the ESP32 uses the floating-point-coprocessor option, which adds the registers `expstate`, `f64r_lo`, `f64r_hi`, `f64s`, `fcr`, and `fsr`, to hold floating point values as well as control and status data.

To emulate the ESP32 SoC, an emulator defines all the registers and implements the instructions relevant to each register.

### B. QEMU and Emulation

The QEMU machine emulator [15] supports many different CPU architectures and allows emulating other operating systems on a host machine. At the same time, QEMU allows the host system to attach a debugger to the emulated machine. A QEMU fork (ESP32-QEMU) [16] adds support for the ESP32 hardware, this way it is easy to run ESP32 firmware images on common computer hardware. It is possible to run one emulated ESP32 SoC per logical CPU core. **So on recent devices, it is possible to emulate more than eight machines at once.**

Additionally, QEMU exposes a debugging interface to the host system so it can inspect the emulated machine and application during runtime. With this, it is possible to save and restore machine states, manipulate registers and memory of the emulated system, and hook into function calls.

**One problem with emulating the ESP32 machine is that the firmware requires specific peripheral devices,** so the emulator has to emulate these devices too. QEMU allows for emulating peripheral devices, however, the number of real devices massively **exceeds the amount of peripheral devices implemented in QEMU.** **so what now?**

To support many different computer architectures, QEMU uses dynamic translation (`dyngen`) by requiring the implementation of small blocks of code specific to each architecture. At runtime, the emulator uses these micro-operations to generate all necessary machine instructions by chaining multiple operations together. The dynamic translator pools multiple instructions of the target architecture into a single Translated Block (TB). A TB ends at a jump instruction as the next CPU state depends on the current TB.

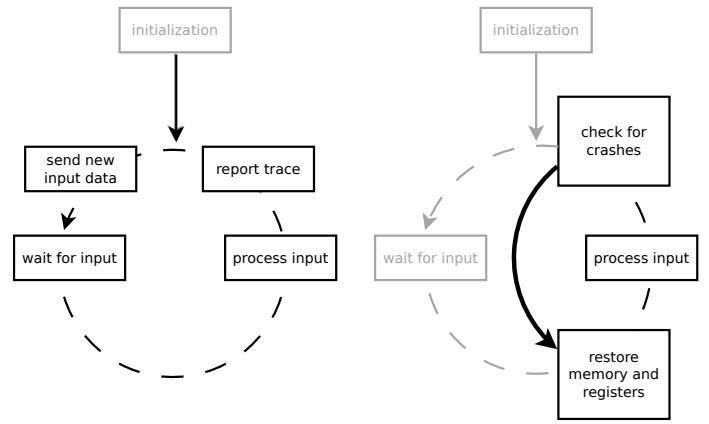


Fig. 2. *Left*: a persistent whitebox fuzzing approach. *Right*: a blackbox fuzzing approach targeting an isolated function.

### C. Fuzzing

Fuzzing [2] repeatedly inserts user input into an application until it misbehaves because unexpected data put it into an undefined state. This is usually because of unsanitized inputs, logical errors in program flow due to unforeseen combinations of inputs (often error cases), or parallelization and insufficient locking mechanisms. This undefined state usually leads to crashes due to exceptions and hangs due to lock-ordering problems or infinite loops.

To enable more data to pass basic data format checks a fixed input is padded with random data. Corpora aggregate common inputs for a specific protocol or data format and can provide a starting input for the fuzzer. The fuzzer can mutate its corpus to generate new inputs.

Blackbox fuzzing assumes no previous knowledge of the targeted application, e.g. because the source code for the application is not available. The application repeatedly accepts random data by the fuzzer until a crash occurs. The fuzzed program does not report any other information on its program state during blackbox fuzzing.

Whitebox fuzzing on the other hand assumes that the source code of the application is available and can be modified to support the fuzzer. Compilers can insert so-called sanitizers, which are runtime checks that sanitize e.g. addresses and pointers, memory accesses, or undefined behavior.

Greybox fuzzing does not assume that the source code is available but allows for modifications due to reverse engineering efforts. For example, trace points can be added after relinking the application.

Random input alone is not efficient enough to achieve sufficient coverage, especially to pass precise checks in parsing. Compilers allow inserting so-called tracepoints at points in the program where the next execution depends on data only available during execution. In guided fuzzing, the fuzzer receives information on which tracepoints are reached during execution. From this information, the fuzzer can check if the input is handled differently from the previous input and decide

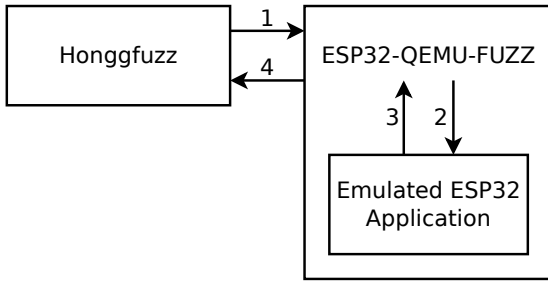


Fig. 3. Data flow between Honggfuzz, ESP32-QEMU-FUZZ, and the emulated application.

if parts of the input should be reused for future input, i.e. refine the input corpus.

Persistent fuzzing does not restart the application after each input and is used on applications that continuously listen for new input, like servers and daemons. This is useful in case the application is not expected to accumulate a lot of state during execution and saves a lot of execution time that otherwise is wasted on initialization. Figure 2 shows a server application that usually listens for input and only processes input, once it receives a request, additional points where the fuzzer interacts with the application are marked. Before the program is in its main loop, the program goes through initialization. Depending on the application, this step can be very expensive. The *left* part of the figure shows a persistent whitebox approach. After processing each input, the application reports feedback on the execution state, the program does not restart after each input.

One more technique used in fuzzing is the isolation of smaller parts of a program, e.g. single function calls or memory areas that are known to contain data. Especially with emulation or virtualization, it is possible to save a state right before the execution of the isolated area and reset the application to that state again to save on initialization time. Figure 2 (*right*) utilizes this in combination with unguided blackbox fuzzing. Only the call to the target function is executed with different inputs.

We will further discuss the blackbox and whitebox approaches from Figure 2 in section IV-C.

#### IV. FUZZING ON ESP32-QEMU-FUZZ

This section first explains how the honggfuzz-qemu [17] component works in combination with the Honggfuzz [18] fuzzer to utilize guided fuzzing. Later, we show the application of the ESP32-QEMU-FUZZ framework utilizing a blackbox and a whitebox approach. Finally, we present our improvements to the framework including a speedup due to parallelization and improvements to the command-line interface.

##### A. ESP32-QEMU-FUZZ and honggfuzz-qemu Internals

ESP32-QEMU-FUZZ combines two QEMU forks: the honggfuzz-qemu fork that allows injecting data into the emulated application and a fork to support the ESP32 machine architecture. Figure 3 shows the relevant communication between the different components for each fuzzing iteration.

Once the application is set up and running, ESP32-QEMU-FUZZ requests input data from the fuzzer (1). Honggfuzz generates this data from its fuzzing corpus. Next, ESP32-QEMU-FUZZ injects this data into the emulated application (2) and the application generates tracing data while processing the input. As soon as the request is answered or the application crashes (3), ESP32-QEMU-FUZZ reports the tracing data back to the fuzzer (4). Honggfuzz now updates its fuzzing corpus based on the tracing points reached. By default, Honggfuzz does up to six mutations to the previous input.

ESP32-QEMU-FUZZ allows for two ways of injecting data into an application. The simpler method is forwarding a port from the host system to the emulated machine. To do this, the `-nic` command-line flag accepts the `hostfwd` parameter which is followed by ISO Layer 3 protocol (TCP or UDP), the optional host IP address, the host listening port, the optional guest IP address of the emulated system, and the target port of the forwarding. Now the emulated application receives all packets addressed to the host IP and host port.

The other method allows injecting a state into a running application by loading a previous memory dump of the data segments (`0x3F400000 - 0x3FFFFFFF`) and general purpose registers (`a0 - a15`). The idea is to dump memory and register content at one point of execution, the `hfuzz_qemu_entry_point`. Optimally, this is right before the call to the target function. Later, ESP32-QEMU-FUZZ sets the `pc` register to this address after loading the memory and register state. To ensure that all instructions are loaded from the firmware image and the SoC is ready to run the application, it is necessary to define a point at which the framework can successfully restore the previously saved state, the `hfuzz_qemu_setup_point`. Once the application is in a state right before executing the target function, the emulator adjusts the input by changing the input buffer and input length before resuming emulation. It is necessary to pass the location of the input buffer `hfuzz_input_data_pointer` and the length register `hfuzz_length_register` to the fuzzing framework. The exit points of our target instructions (`hfuzz_qemu_exit_points`) usually are all return instructions of our target function. After the emulator reaches one of the exit points, it pauses the execution and resets the state back to the call to our target function.

As Honggfuzz is a feedback-driven fuzzer, the emulator must generate trace information. Honggfuzz supports utilizing hardware registers on Intel CPUs, a subsystem for the Linux kernel, compile-time instrumentation, and sanitizer instrumentation. Because these methods are not available on an ESP32 Soc, the honggfuzz-qemu project can insert instrumentation during emulation, i.e. every change to the `pc` register and every time a `cmp` micro-operation is executed. Furthermore, calls to the libc functions `strncasecmp`, `strcasecmp`, `strncmp`, and `strcmp` are intercepted and additional trace data is generated from the string comparisons. The addresses of the string comparison functions are hard coded in ESP32-QEMU-FUZZ.

To reduce the length of a trace, it makes sense

why those



```

void processData(int len, char *inp) {
    if (len > 5 || inp[0] != 'G' || inp[1] != 'E'
        || inp[2] != 'T' || inp[3] != '_'
        || inp[4] != '/' )
        return;

    char stack[20], *heap = malloc(20);

    switch (inp[5]) {
    case 's':
        for (int i = 0; i < inp[6]; i++)
            stack[i] = 0xDE;
        break;
    case 'h':
        heap[inp[6]] = 0xAD;
        break;
    case 'n':
        printf("%s", NULL);
        break;
    case 'd':
        free(heap);
        break;
    case 'p':
        printf(inp);
        break;
    }
    free(heap);
}

```

Fig. 4. Faulty `processData` function from the `tcp_server` application by Börsig *et al.* [4], modified for printing

to reduce the address range in which the emulator traces the `cmp` micro-operations. This is possible by setting the `hfuzz_qemu_start_code` and `hfuzz_qemu_end_code` variables and allows the fuzzer to find better inputs that relate to new instructions reached during the execution of the target function. Similar restrictions to the `pc` register do not exist.

In case the start and end code are chosen such that the execution covers too many instructions, the fuzzer will take a lot longer to cover states that relate to the target function, due to state explosion. On the other hand, if the start and end code do not cover the target function, no useful trace information is reported and adjustments to the input data are then independent of the state of our target function.

### B. TCP-Server Example Application


Börsig *et al.* present a small example application including bugs classified according to *What you corrupt is not what you crash: Challenges in fuzzing embedded devices* by Muench *et al.* [19].

The server listens on a TCP socket and accepts all packets on port 80. Each packet passes to the `processData` function where the content is inspected, listing 4 shows the relevant parts of the faulty function. The first five bytes are checked to contain `GET /` using byte-by-byte comparison.

It is important to point out that comparing each byte individually generates different tracing data than a call to `strncmp`. Each byte comparison generates feedback via the `cmp` micro-operation. This means that the fuzzer can reverse the `GET /`-string byte-by-byte. A `strncmp`-call on the other

hand gives one data point that contains the top three Bits of the `pc` register and the lower 29 Bits of the `a0` register, the previous TB, the two string parameters, and the length parameter. The string length and the corresponding strings are truncated to a length of 64.

The function jumps to different errors depending on the next byte. In case the request starts with `GET /s`, it causes a stack overflow, and `GET /h` causes a heap overflow. With `GET /n` a NULL-pointer dereference occurs, `GET /d` causes a double-free, and with `GET /p` user input passes to `printf` as the first argument.

Because `processData` requires the first 5 Bytes of input to exactly match, it is not reasonable, to blindly try all  $2^{40}$  possible inputs to pass the check. It might make sense to prefix any fuzzing string with the required string, and for some protocols, this might make sense. However, it is not generalizable for security research and quality assurance and therefore we do not proceed with this idea. 

### C. Reproducing the Framework Methodologies

The ESP32-QEMU-FUZZ repository [4] comes with instructions on how to build the framework, an example application, and instructions on how to run the different fuzzing methods against the sample application. The ESP32 Integrated Development Framework (IDF) is required for building the example application. **It is not mentioned which version of the IDF is used in the original paper.**

To reproduce the results, we use Ubuntu 20.04 LTS [20] and ESP32 IDF v.4.0 [21] as those versions were available at the publication date of the paper by Börsig *et al.*

Following the instructions for building the included Honggfuzz and QEMU projects, both projects compile without any errors.


Instructions for generating the firmware image file exist but result in compilation errors due to `incompatible-pointer-types` and `format-overflow` compiler warnings. The compiler warns about the input buffer as the first argument to `printf`. Adding a simple wrapper function around `printf` results in effectively the same bug but the compiler does not spot it. A cast to the expected pointer type fixes the other warning. However, it is not necessary to compile the `tcp_server` application because a firmware image comes with the project. 

Figure 5 shows constants and their locations in instruction memory which ESP32-QEMU-FUZZ requires before fuzzing the `tcp_server` application. Whitebox fuzzing uses `hfuzz_qemu_start_code`, and `hfuzz_qemu_end_code`. Blackbox fuzzing uses the variables `hfuzz_qemu_setup_point`, `hfuzz_qemu_entry_point`, and `hfuzz_qemu_exit_point`. The entry and exit locations used in the blackbox approach are exactly pointing to the first address and last address of the `processData` function. In contrast to that, the start and end locations used in the whitebox approach only roughly point to the function

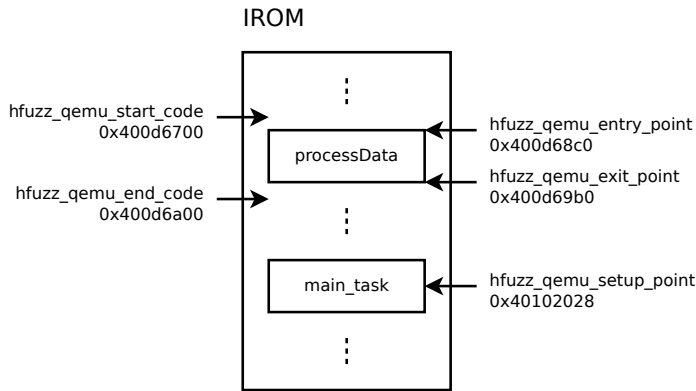


Fig. 5. Memory layout of the `tcp_server` application and addresses used by ESP32-QEMU-FUZZ during blackbox and whitebox fuzzing.

location. Once the application passes the setup point, fuzzing can begin.

1) *Whitebox Fuzzing within QEMU*: the instructions yield results due to programming errors in the framework code. The included source code has a hard-coded value for `hfuzz_qemu_setup_point` which then causes a call to `setup_fuzzing_entrypoint` once the program counter has the value of the defined setup point. This function sets the `pc` register to the value of `hfuzz_qemu_entry_point`, which is 0. Further on, the function passes the NULL-initialized variable `hfuzz_dump_file` to `open`, does not check the return value and passes the potentially returned file descriptor to `lseek`, casts that unchecked return value to a `size_t` and passes it to `malloc`. Again, the program does not check the return value, and the NULL-pointer ends up in a `read` call where the program finally crashes.

After minor adjustments to the control flow by returning from `setup_fuzzing_entrypoint` in case `hfuzz_dump_file` is a NULL-pointer, and additionally setting `hfuzz_qemu_entry_point`, the example would crash within a few minutes.

For whitebox fuzzing, ESP32-QEMU-FUZZ injects the fuzzing data by forwarding the application port and opening a TCP socket. After reaching `hfuzz_qemu_setup_point`, the emulator requests input data from the Honggfuzz fuzzer and writes the data to the socket. The `pc` register of the `tcp_server` application eventually points to the location of the `processData` function. From that point on, all `cmp` micro-operations are recorded while the program counter is in between `hfuzz_qemu_start_code` and `hfuzz_qemu_end_code`. Calls to the string comparison functions and the values of the `pc` register are also recorded. Once the emulator receives a response to the input, it reports the tracing information back to the fuzzer. Honggfuzz processes the information and adjusts its corpus to generate better fuzzing data for future requests.

2) *Blackbox Fuzzing within QEMU*: the addresses mentioned in the instructions do not work with the provided firmware image. As soon as the emulator loads the provided

memory dumps into the application, a *Guru Meditation Error* occurs.

The addresses from the instructions also do not correspond to the checked-in `tcp_server.img`. The provided address for `processData` instead points to an initialization function, which is before the application listens on the specified port.

By creating a new `tcp_server` firmware image, it is possible to start the blackbox fuzzer. It expects seven additional parameters: setup, entry and exit points, the address of the data buffer, the register containing the buffer length, a path to the memory dump file, and a path to the register dump file. This means that the application state during blackbox fuzzing is reset after each request.

The application state is not reset during whitebox fuzzing. This means that the whitebox fuzzing approach has a lot more state changes that are kept in between inputs, which in turn results in more crashes due to e.g. heap corruption. Furthermore, it is unlikely to find a crash using unguided blackbox fuzzing, as it is unlikely a random input matches `GET /`. It is still useful to attempt blackbox fuzzing to infer the overhead of whitebox fuzzing.

#### D. Fixing, Enhancing Utility, Ease of Application, and Speed

For the blackbox approach, we replaced a call to `snprintf` with `memcpy` to ensure that the whole fuzzing input is copied to the input buffer. Börsig *et al.* purposefully avoided a `memcpy` because the `tcp_server` application does not require NUL-Bytes to crash. Nevertheless, this is not generalizable for other protocols and applications. good

The `-fuzz` option as well as the `setup_fuzzing_entrypoint` did not properly validate input and file content and did not report errors to the caller. Additionally, we now check the return values of the syscalls.

In addition to the necessary changes to make the framework operate as promised, we contribute the following enhancements.

We added a script to easily generate a memory and register dump and write the start and end address of a target function to a file. The same script allows quickly dropping into a debugger to inspect the program state before the call to the target function.

Previously it was only possible to provide one register which contains the length of the input buffer. We now allow a list of registers in this form: `aX+aY+aZ`. We did this because, in the `tcp_server` application, multiple registers contain the length of the buffer. To verify this, we set a break-point at the location of the target function (`processData`) and print all registers. We repeat this with a different input and determine multiple registers contain the length of the data buffer.

The workflow as presented in the instructions requires us to first find the addresses to the tracing locations. After that, these constants are inserted into the QEMU source code and we rebuild the QEMU project. Finally, fuzzing Honggfuzz can start.

The middle step of recompiling the binary can be skipped by providing separate flags depending on the fuzzing method-

ology, and setting the required constants via the command line interface. It now suffices to pass the `-whitebox init:start:end` parameters to `ESP32-QEMU-FUZZ`. Which are stored in the `hfuzz_qemu_setup_point`, `hfuzz_qemu_start_code`, and `hfuzz_qemu_end_code` variables. Furthermore, we rename the `-fuzz` argument to `-blackbox`. This enables us to use the `save-restore` method in combination with `whitebox` fuzzing instead of persistent fuzzing. It is important to specify the `-whitebox` option before the `blackbox` option as both write to the `hfuzz_qemu_setup_point` variable.

The port forwarding described in the previous section limits the framework to a single instance of firmware running, which in turn reduces the iterations per second significantly. It is possible to simultaneously start a second instance of Honggfuzz on a different host port but this in turn increases the complexity of coordinating multiple fuzzer instances running. Since the request with the fuzzing data is sent from the same process that is also forwarding that request to the emulated machine, it is not necessary to use a TCP socket. However, it is a lot easier to enhance the forwarding code instead of creating another special-purpose subsystem. Instead of always using port 8081 like in the instructions for reproducing the previous results, we allow the user to provide a ? instead of a port number. The QEMU slurp subsystem will then open an ephemeral port on the host system and use that for forwarding. This random port number is available to the function that sends the request with the fuzzing data. This simple hack allows running multiple `whitebox` fuzzing instances in parallel and drastically speeds up the iterations per second.

## V. EVALUATION

For the following statements regarding performance, we use an AMD Ryzen 5 5600G processor with 6 cores and 12 threads. If not stated otherwise, we use twelve QEMU instances in parallel.

With the adjustments<sup>1</sup> both approaches work on the provided firmware image and are reproducible. The `whitebox` setup finds crashes and the input strings start with `GET /` followed by the necessary letters to trigger a crash.

For `whitebox` fuzzing, replacing the byte-by-byte comparison of the `GET /`-string with a call to `strncmp` increases the time to crash by a factor of three, from 90 seconds to 270 seconds. The string comparison generates fewer trace points per execution that the fuzzer can use to learn about the required input format.

Using the previously described example `tcp_server` application, Figure 6 shows that the fuzzing scales very well with the number of available threads for both `blackbox` and `whitebox` fuzzing. The slight drop in scaling between 11 and 12 threads is likely because the twelve QEMU instances completely exhaust the available CPU resources and the Honggfuzz process bounces between the cores. This means the scheduler stops one

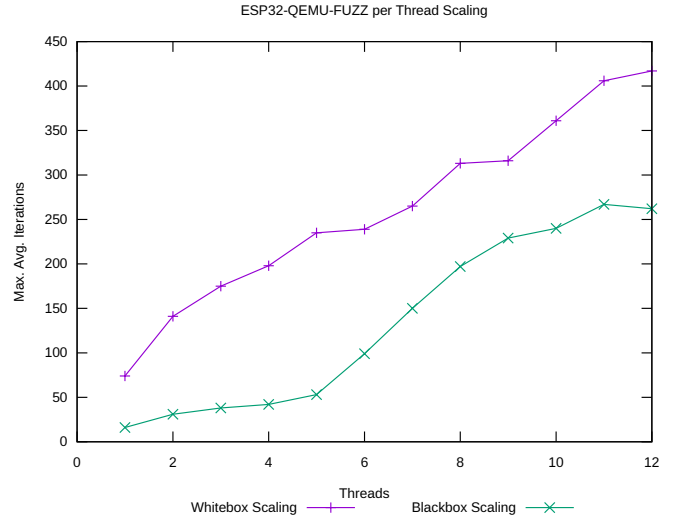


Fig. 6. Per thread scaling of whitebox and blackbox fuzzing.

of the emulators each time the fuzzer receives tracing data. The per-core scaling is sub-linear with an average increase of 31 iterations per added thread. Except for the case of 12 threads, no decrease in scaling is visible. However, the differences in scaling from 5 to 6 threads and 8 to 9 threads are negligible. This means there is reason to believe that the parallelization will continuously scale with higher thread-count CPUs.

In addition to this, we compare the `ESP32-QEMU-FUZZ` framework to Honggfuzz running on a native binary. Because the source code of the `tcp_server` application is quite simple and uses normal C Standard Library and POSIX functions, it is possible to compile a slightly modified version on Linux. As only the `whitebox` use case supports feedback-driven fuzzing, slight modifications to the source code are possible during development and testing to run the application on different hardware. The instructions on how to use persistent fuzzing with Honggfuzz are easy to follow, and the necessary modifications to compile and fuzz the application on the host machine took less than an hour. Furthermore, this approach achieved over 4000 iterations per second in a single-threaded application and roughly 23000 iterations per second with 12 threads. The gradient drawn from single-threaded performance to the performance achieved utilizing all threads is similar in both cases:  $417/74 \approx 23000/4000$ . However, this approach achieves 55 times as many iterations per second.

Our adjustments reduce the steps needed to fuzz an ESP32 application, as finding the constants in the program and then recompiling `ESP32-QEMU-FUZZ` is no longer necessary. It is possible to transfer the automation to other `whitebox` fuzzing applications quickly due to the automated calls to the debugger. This means that it is now possible to integrate the tooling into development and testing processes for ESP32 development.

Applying blackbox or whitebox fuzzing no longer requires

<sup>1</sup>available at <https://github.com/moritzbuhl/esp32-fuzzing-framework>

additional undocumented changes to ESP32-QEMU-FUZZ and it is easy to switch the desired fuzzing method. Not only does this reduce potential manual labor time, but it also reduces the number of disruptions during the analysis required for greybox fuzzing.

## VI. DISCUSSION

In this section, we describe the impact of our work and argue for certain design decisions before listing limitations, shortcomings, and possible future research.

### A. Impact of the Enhanced ESP32-QEMU-FUZZ Framework

Using whitebox fuzzing with ESP32-QEMU-FUZZ still requires manual work and an understanding of a target program before the advantages of guided fuzzing are exhausted. It is easy to automate and integrate this work into the development process.

The speedup due to the enabled parallelization is significant and changes parts of the evaluation in the initial evaluation by Börsig *et al.* as the whitebox approach achieves more iterations per second than the JTAG hook. `good`

The described modifications to ESP32-QEMU-FUZZ allow for more extensive use of the framework and integration of the framework into development pipelines.

### B. Design Decisions

The modifications to the string comparison functions in QEMU to generate trace points based on the string parameters cuts the strings at 64 characters. This is a technical limitation and negligible as a search for all strings in the C source code of the ESP32-QEMU-FUZZ repository shows that less than 0.3% of all strings are longer than 64 characters.

Utilizing ephemeral ports for parallelization instead of another IPC mechanism or Unix domain sockets is not well thought out and a slight chance for collisions exists. However, the slurp sub-project of QEMU which enables TCP port forwarding does not support any other socket type than IPv4 on the host site. Because of that, we use ephemeral ports.

### C. Limitations

The honggfuzz-qemu project was not significantly updated in 3 years whereas ESP32-QEMU [16] receives regular contributions. It might make sense to update the ESP32-QEMU part of ESP32-QEMU-FUZZ. Additionally, the implementation of the fork server in ESP32-QEMU-FUZZ slightly differs from the one in the honggfuzz-qemu repository [17].

While it is now theoretically possible to use the save-restore injection with whitebox fuzzing, it is not fully tested. Determining the difference in speedup would allow for an easier comparison of the approaches. ~~But the lack of a way to crash earlier on undetected corruptions means that the persistent fuzzing approach is more reliable.~~

The save-restore injection method only resets the register state of the general purpose registers and the program counter. Usually, the callee should not rely on the state of some registers at the beginning of a function call, but the thread

pointer, processor state, and some floating point control data (e.g. exceptions, rounding direction modes) should also be restored. It might also make sense to allow the fuzzer to jump to an arbitrary instruction and not just to a function prologue.

It is only possible to inject a single buffer and modify registers to encode the length of the buffer for the save-restore injection method. The framework could profit from allowing the adjustment of more registers and stack data, especially if arbitrary restore locations are implemented.

The start and end location for whitebox fuzzing to track `cmp` micro-operations only allow a single interval, which means it is not necessarily possible to cover all callees of the target function without covering other nearby functions.

The addresses of the string comparison functions are hard-coded into the framework. It would make sense to make them adjustable via the command line.

### D. Future Research

Utilizing modified instrumentation from Harzer Roller [5] it should be possible to generate runtime checks to crash earlier while fuzzing the ESP32 platform. This works without requiring the source code of the firmware. It should be possible to implement instrumentation that works similarly to the various sanitizers by the LLVM project [22] [23] [24]. This would allow for faster more reproducible crashes, and make the framework a lot more useful for security research, especially if this approach is fully automated.

It would be interesting to emulate more peripheral devices and fuzz the interfaces to apply the approaches of the drifuzz project [6] to find vulnerabilities in lower levels of the network stack and other hardware-related interfaces.

The open-source firmware Tasmota [25] currently does not boot in QEMU, it would be interesting if it was possible to apply the whitebox approach to the project. Especially since the source code is available and the project is widely used.

## VII. CONCLUSION

We use the blackbox and whitebox modes of the framework on a simple application, extended the framework to enable a speedup of six on common computer hardware, and conclude that the framework is still more like a proof of concept than something that could find application in the development industry.

The use case for this framework is very limited due to its capabilities. From the evaluation, it is easy to conclude that using this framework during development and testing is not useful in case the application also compiles on common x86 hardware. If the firmware only runs on the ESP32 or requires peripherals that are supported by QEMU, it may make sense to use ESP32-QEMU-FUZZ.

For security research where source code is not available and emulation is the only option to inspect the firmware at runtime, this is the only fuzzer that fulfills the requirements for ESP32 firmware. However, we show that it is likely that the framework needs further modification before applying it to a use case.



For the future of this framework, it is necessary to focus on its distinguishing features of combining real ESP32 operating systems and libraries with customized applications for the platform. **Less focus on the use of ESP32 development and instead focus on reducing the efforts of ESP32 security research is necessary.**

## REFERENCES

- [1] Blakadder, "Esp32 based devices," December 2022. [Online]. Available: <https://templates.blakadder.com/esp32.html>
- [2] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.
- [3] M. Börsig, S. Nitzsche, M. Eisele, R. Gröll, J. Becker, and I. Baumgart, "Fuzzing framework for esp32 microcontrollers," in *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*, 2020, pp. 1–6.
- [4] "Esp32-qemu-fuzz source code repository," November 2020. [Online]. Available: <https://github.com/MaxCamillo/esp32-fuzzing-framework/>
- [5] K. Bogad and M. Huber, "Harzer roller: Linker-based instrumentation for enhanced embedded security testing," in *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*, 2019, pp. 1–9.
- [6] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting bugs in device drivers from golden seeds," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1275–1290.
- [7] M. E. Garbelini, C. Wang, and S. Chattopadhyay, "Greyhound: Directed greybox wi-fi fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 817–834, 2020.
- [8] R. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.
- [9] *ESP32 Series Datasheet*, Espressif Systems, 2022, v. 4.0. [Online]. Available: [https://www.espressif.com/sites/default/files/documentation/esp32\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf)
- [10] *ESP32 Technical Reference Manual*, Espressif Systems, 2022, v. 4.7. [Online]. Available: [https://www.espressif.com/sites/default/files/documentation/esp32\\_technical\\_reference\\_manual\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf)
- [11] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 15–26. [Online]. Available: <https://doi.org/10.1145/1455770.1455775>
- [12] K. Lehniger, M. J. Aftowicz, P. Langendorfer, and Z. Dyka, "Challenges of return-oriented-programming on the xtensa hardware architecture," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 154–158.
- [13] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 31–46.
- [14] *Xtensa® Instruction Set Architecture (ISA) Summary*, Cadence Design Systems, Inc., April 2022. [Online]. Available: [https://www.cadence.com/content/dam/cadence-www/global/en\\_US/documents/tools/ip/tensilica-ip/isa-summary.pdf](https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ip/tensilica-ip/isa-summary.pdf)
- [15] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.
- [16] Ivan Grokhotkov, "Qemu fork to support esp32 machine," Espressif Systems, December 2022. [Online]. Available: <https://github.com/espressif/qemu>
- [17] "honggfuzz-qemu source code repository," thebabush, December 2022. [Online]. Available: <https://github.com/thebabush/honggfuzz-qemu>
- [18] "Honggfuzz source code repository," Google Inc., December 2022. [Online]. Available: <https://github.com/google/honggfuzz>
- [19] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *NDSS*, 2018.
- [20] Canonical Ltd., "Ubuntu 20.04.5 lts (focal fossa)," February 2023. [Online]. Available: <https://releases.ubuntu.com/20.04/>
- [21] Espressif Systems, "Espressif iot development framework," February 2023. [Online]. Available: <https://github.com/espressif/esp-idf/>
- [22] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.
- [23] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 46–55.
- [24] "Clang 16.0.0 documentation undefinedbehaviorsanitizer," The Clang Team, December 2022. [Online]. Available: <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
- [25] Theo Arends, "Tasmota: Open source firmware for esp devices," February 2023. [Online]. Available: <https://tasmota.github.io/>